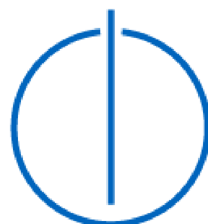
Technische Universität München
Fakultät für Informatik

Incremental Language Independent Static Data Flow Analysis

Fabian Streitel

Master's Thesis in Computer Science



Technische Universität München
Fakultät für Informatik

Incremental Language Independent Static Data Flow Analysis

Inkrementelle Sprachunabhängige Statische Datenflussanalyse

Fabian Streitel

Master's Thesis in Computer Science

Supervisor: Prof. Dr. Dr. h.c. Manfred Broy

Advisors: M.Sc. Maximilian Junker
Dr. Benjamin Hummel
Dr. Martin Feilkas

Submission Date: April 22, 2014

Declaration

Ich versichere, dass ich diese Master's Thesis selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

I assure the single handed composition of this master's thesis only supported by declared resources.

Munich, April 22, 2014

....

Fabian Streitel

Abstract

Data flow analysis is a useful technique to find bugs in a software system but tool support only exist for a handful of programming languages. Today, many software projects are written in more than one language and it is costly and time-consuming to implement a separate data flow tool for each of them. In this thesis we develop an approach to language independent data flow analysis which requires only a small overhead to add support for a new language to the system. Thus we can detect problems in the source code of an application in a uniform way in a wide variety of programming languages. In a case study on several systems written in three different languages we show that our approach has high precision and recall and compare it to FindBugs and FxCop, established data flow tools for Java and .NET.

Acknowledgements

I would like to thank my advisers Dr. Benjamin Hummel, Maximilian Junker and Dr. Martin Feilkas for their guidance during my entire research, for their helpful insights on complicated topics, their tireless proof-reading of this thesis and for supporting my work and ideas. You made writing this thesis a great experience for me.

I also thank Prof. Dr. Manfred Broy for allowing me write this thesis under his supervision and for all his interesting software engineering lectures I attended, which were one of the reasons I became interested in this research subject.

A big thank you also goes to the CQSE GmbH and all my colleagues there for allowing me to use your expertise and tools – especially ConQAT and Teamscale – without which this thesis would not have been possible. Thank you for the friendly and productive work environment and all the tea, coffee and dried fruits that kept me going through the days.

My friends and family must also be mentioned here. Thank you for supporting me – not only while writing this thesis but throughout my entire studies. And thank you for dragging me away from my computer from time to time to see the real world and get some perspective.

And finally: my girlfriend, Fangyi Zhi. Thank you so much for your love and support. For being there for me. For being my anchor. For brightening up my day. I can't even imagine where I would be without you. I love you BaoBao.

Table of Contents

1 Introduction **1**
1.1 Problem 1
1.2 Contribution 2
1.3 Outline 2

2 Related Work **3**
2.1 Language Independent Algorithms for Quality Analysis 3
2.2 Data Flow Analysis for Software Quality 4
2.3 Language Models 5

3 Preliminaries **7**
3.1 Control Flow Graph 7
3.2 Data Flow Analysis 7
3.3 Static Single-Assignment Form 8
3.4 Language Independence 9
3.5 Shallow Parser 10
3.6 False Positive and False Negative 11
3.7 Precision and Recall 11
3.8 F_β Score 12

4 Language Model **13**
4.1 Source Structure 13
4.2 Data Reference 15
4.3 Data Transfer 15
4.4 Conditions 17
4.5 Worked Example 18

5 Data Flow Framework **21**
5.1 Implementation 21
5.2 Language Dependent Parts 21
5.3 Language Independent Parts 25
5.4 False Positive Filters 28

6 Evaluation **33**
6.1 Research Questions 33
6.2 RQ1: Integrating another language into our framework 34
6.3 Studied Systems 35

6.4 Reference Tools 35
6.5 RQ2a: Precision and recall compared to other tools 37
6.6 RQ2b: Execution time compared to other tools 41
6.7 RQ3a: Precision and recall on different languages 42
6.8 RQ3b: Execution time on different languages 44
6.9 Threats to Validity 44
6.10 Discussion 46

7 Conclusion **49**
7.1 Future Work 49

1 Introduction

Data flow analysis is a static analysis technique that is applied to source code. It models the flow of data throughout a program, e.g. from one variable to another and across branches and loops. Although it was originally devised for optimisation tasks in compilers [21], it can also be used in the software quality context to find bugs and maintenance problems in software programs [10]. Certain anomalies in the data flow graph are identified and reported as possible problems, e.g. variables that are declared but never used. This is helpful both during the initial development of a software to reduce the number of bugs in the program, as well as during its maintenance phase to identify parts of the source code that might contain problems. Over the years, many programs were created that perform such an analysis for a certain language, e.g. FindBugs [12] for Java or mygcc [23] for the GCC compiler framework.

Today, many software projects use not only a single language but instead combine many different languages for different purposes. A good example are web applications that often use several languages like Java, Ruby or C# on the server side and a combination of JavaScript, Adobe Flash and Java applets on the client side. For some of these languages, data flow analysis tools are available, but their quality and the range of analyses they offer differ widely: for Java, FindBugs provides many different, in-depth analyses, while for languages like JavaScript or ABAP, little to no tool support exists.

Furthermore, there are languages which are not widely used and often undocumented, e.g. proprietary languages that are tied to a software solution of one specific vendor. For these languages, implementing data flow analysis from the ground up can be cumbersome, time-consuming and cost-intensive. Since the analyses offered by most tools are very similar in nature, it would be desirable to have a framework available that allows a simple and fast integration of these languages into a set of existing, language independent analyses.

1.1 Problem

The problem is therefore threefold:

1. We want to have a language independent framework to support data flow analysis on a wide range of languages.
2. We want the effort necessary to develop a language back-end for this framework to be minimal.
3. We want the results of the analyses implemented on this framework to be useful to the developer.

1.2 Contribution

In this paper we present the following contributions to this problem:

1. A language model that captures the information necessary for a wide range of data flow analyses.
2. A language independent data flow framework.
3. Several analyses based on this framework.
4. An evaluation of this approach
    - in terms of the ease of adding new languages to the framework.
    - by comparing it to established tools.
    - by comparing its performance on different languages.

1.3 Outline

We will first present related work in chapter 2, followed by the definition of important terms and concepts in chapter 3. In chapter 4 we present our chosen language model. This is followed by a description of the data flow framework and its analyses in chapter 5, which we created on the basis of the model. In chapter 6 we evaluate our approach and compare it to existing data flow tools. Finally, we will give a summary of our work in chapter 7 and show possibilities for further research on this topic.

2 Related Work

Other work related to this thesis can be classified into three categories: Language independent algorithms that have a quality assessment as their goal, other data flow analysis techniques used for software quality control and other language models as the basis of language independent algorithms.

2.1 Language Independent Algorithms for Quality Analysis

Several other quality aspects of source code have been measured successfully using language independent algorithms.

2.1.1 Clone Detection

Language independent algorithms have been proposed for the problem of detecting clones in source code, i.e. snippets of code that have been copied, optionally with minor alterations. These clones are generally seen as an indication of poor quality as they complicate maintenance of the program.

There are several approaches to detecting code clones which are language independent. These algorithms usually rely on a language dependent pre-processing of the source code. The transformed code is then processed in a language independent manner. Such pre-processing ranges from simple string manipulation (removing comments and white space) as in [9] to more complicated parsing of the language, e.g. to identify interesting structural features as in [5].

Our approach is similar in that we also apply a language dependent pre-processing and then execute language independent algorithms on the obtained data structures. It differs from clone detection since more in-depth knowledge about the source code is required: Data flow analysis needs to understand the semantics of the code to some extent, e.g. the interrelationship between different variables or the meaning of conditionals. Thus it also identifies completely different quality problems.

2.1.2 Source Code Metrics

A plethora of metrics have been proposed to measure the quality of source code. Some of these are very specific and tied to a certain language, but most can be formulated in a language independent way. The FAMOOS project, for example, created a meta-model for object-oriented languages and used this to extract a variety of metrics from source code written in different languages [16].

Other approaches define metrics on graph-based meta models to make them language independent and consistent in their definition. [18] provides such a model and shows how several existing metrics can be translated into this representation.

Our approach is similar to this since we also create a model of the languages we support. Yet, our model is more detailed as it needs to take into account not only the syntactical information in the source code, but also the semantics of some of its statements.

2.1.3 Data Flow Patterns

mygcc is an extension to the GNU Compiler Collection[1] (GCC) that allows specifying data flow violations via a pattern language, first presented in [23]. These patterns – called *condates* – are specified in a special pattern language and evaluated on the so-called GIMPLE representation – a language independent version of the abstract syntax tree of the analysed program that has undergone several simplifying transformations.

Since this pattern matching occurs in a compiler, the exact grammar of the analysed language must be specified correctly. Furthermore, the GIMPLE representation puts several non-trivial restrictions on the writer of such patterns, which may or may not have an influence on the analysis results, depending on the check that is being performed and the language of the analysed source code [23]. Finally, what differentiates mygcc most from this thesis is its intent: it does not aim to create language independent analyses but to enable users to create custom analyses for a single language and project setting.

2.2 Data Flow Analysis for Software Quality

A variety of tools exist that use data flow analysis to measure software quality.

2.2.1 FindBugs

FindBugs [12] is an open source tool created at the University of Maryland that performs various checks (called *bug patterns*) on Java bytecode to detect problematic and error-prone areas in the source code. Unlike our approach, it is therefore limited to languages that can be compiled to this representation. Some of its bug patterns use data flow analysis. Most notably, the tool features a null pointer analysis that has been optimised to produce a low number of false positives [13].

2.2.2 Commercial Data Flow Analysis Tools

There exist other, commercial products that claim to perform data flow analysis on different languages, e.g. Parasoft JTest[2] for Java or Virtual Forge CodeProfiler[3] for ABAP. For most of these tools there is little information available as to what kind of

[1] Website: `http://gcc.gnu.org/` last accessed 2013-12-02
[2] Website: `http://www.parasoft.com/jtest` last accessed 2014-04-01.
[3] Website: `https://www.virtualforge.com/en/portfolio/codeprofiler.html` last accessed 2014-04-01.

analyses are actually performed. All of them are again designed to work with only one language and it can be assumed that they use a full parser to facilitate their analyses.

Several other commercial tools claim to be able to execute data flow analyses on C/C++, C# and Java code, for example Klockwork Insight[4] or Coverity Code Advisor[5]. Unfortunately, no information is available on whether they were actually designed to be language independent and could be extended easily or just exploit the high similarity between these C-like languages. Furthermore, no estimate can be given on the amount of work necessary to adapt them to a new language.

2.3 Language Models

Other projects have created models of programming languages to create language independent tools.

2.3.1 LLVM

The low level virtual machine (LLVM) is a project that aims to create "a compiler framework designed to support transparent, life- long program analysis and transformation for arbitrary programs" [17]. It provides a low-level code representation into which all source code is transformed. The compiler then only works on this representation, making all optimisation and translation work language independent. In order for this to work, Lattner and Adve had to create a language model that encapsulates a wide range of languages while still enabling the compiler to create valid binaries. This model includes, among other things, a specification of a language independent type system and exception handling. Several compilers have been built on the basis of the LLVM, including ones for C/C++ and Java.

Our language model differs from LLVM's mainly in the granularity of the represented information. We don't aim to fully understand the semantics of all statements, which would make implementing new back-ends for our analyses too time-consuming and error-prone. Instead, we rely on a select few heuristics to extract only the information necessary to drive our analyses.

2.3.2 FAMOOS

As part of the FAMOOS project, which was researching in the area of software re-engineering, a language model called FAMIX [8] was created. It was used as an information exchange format and as the basis of their re-engineering platform Moose. Unlike our model, it is only applicable to object-oriented systems and thus models parts of the source code that are not of interest for our data flow analyses and may not even be present in all the languages we aim to support, e.g. the inheritance relationships between classes. Furthermore, the model is very detailed on its lower levels, e.g. when modelling method

[4]Website: `http://www.klocwork.com/products/insight/` last accessed 2014-04-01.
[5]Website: `http://www.coverity.com/products/code-advisor/` last accessed 2014-04-01.

invocations. As we expect that this level of detail is neither necessary nor helpful for our framework and would only increase the time it takes to develop a new back-end for it, we chose to not use FAMIX.

3 Preliminaries

In this section, we define several important terms that are used throughout this thesis.

3.1 Control Flow Graph

Data flow analysis needs information about every statement in a program and how control is passed between them at runtime. For this purpose we define a *control flow graph* (CFG) – similar to Allen in [2] – as a directed graph in which the nodes represent statements and the edges represent transfer of control between these statements.

We furthermore define a *basic block* as a linear sequence of statements that does not contain branches. In the CFG these are subgraphs where every node except the first has exactly one predecessor and every node except the last has exactly one successor. Thus, there is only one path leading from the entry node of the subgraph to the exit node. If control enters such a basic block, all nodes of its subgraph receive control in order exactly once, until its exit node is reached. Figure 3.1 shows the CFG for the pseudo code in listing 3.1 with all basic blocks marked in blue.

3.2 Data Flow Analysis

In [10], Fosdick and Osterweil describe *data flow analysis* as follows:

> The program is scanned in a systematic way and information about the use of variables is collected so that certain inferences can be made about the effect of these uses at other points of the program.

More formally speaking, we add edges to a CFG that connect the definition (or assignment) of a variable to all its uses and a use to all possible definitions. Based on this graph, we can execute analyses that use this so-called def-use information to find anomalies in the graph, such as a use of a variable that has not yet been defined or a definition that has no uses.

Data flow analyses may fall into one of two categories: *Flow-sensitive* analyses only care about the order of the statements as given by the CFG. *Path-sensitive* analyses in addition care about the conditional information at branch points and how they influence the possible values carried by a variable, e.g. whether a variable may be `null` in a certain branch or not.

Listing 3.1: Example pseudo code.

```
1 a = 1
2 while (a < 10) {
3   a++
4   write(a)
5 }
6 write("end")
```

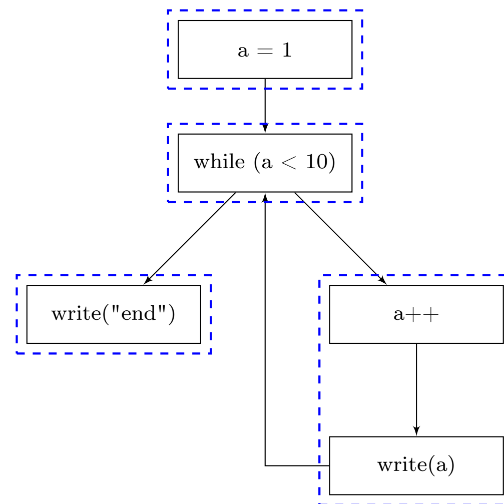


Figure 3.1: A CFG with all basic blocks marked with dashed blue lines.

3.3 Static Single-Assignment Form

Such a def-use graph can be quite complex: Every definition may have several uses and each use may also have several possible definitions, depending on the control flow of the program. This makes writing analyses quite complex. We therefore introduce *static single-assignment form* (SSA form), which is a transformation of the original def-use graph that simplifies it.

Andrew Appel describes it as follows [3]:

> In SSA, each variable in the program has only one definition - it is assigned to only once. [...] To achieve single-assignment, we make up a new variable name for each assignment to the variable.

A variable called a in the original program would therefore be split into several versions a_1, a_2, a_3, etc. at each point in the program where it is assigned. This is straight-forward in a program that contains no branches. After such a branch point, however, at the point where both branches meet, a variable may have more than one possible point of definition, e.g. one in the `then` branch of an `if` statement and one in the `else` branch.

To get around this problem, we introduce the imaginary ϕ-function, which selects the correct definition of a variable based on the branch that was taken to enter the ϕ-node.

Listing 3.2: Example code to be transformed to SSA form.

```
1 a = 1
2 if (a > 0) {
3   a = 2
4 }
5 use(a)
```

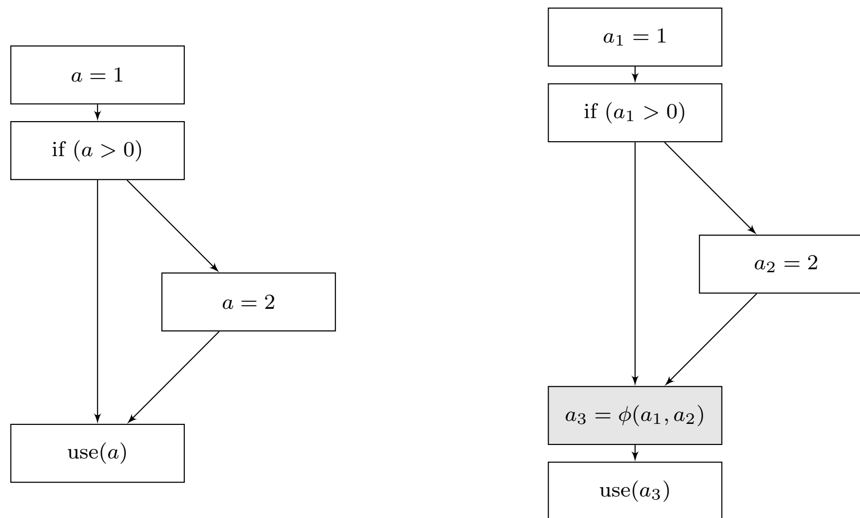


Figure 3.2: The original control flow graph (left) and the control flow graph in static single-assignment form (right). The SSA form contains versioned variables and a ϕ-node (grey background) was inserted where the two branches meet.

An expression such as $a_3 = \phi(a_1, a_2)$ would therefore assign the value of a_1 to a_3 if the statement received control from the first branch. If control was passed from the second branch, the value of a_2 would be assigned to a_3 instead.

To illustrate this transformation, consider the pseudo code in listing 3.2. Figure 3.2 shows how the CFG of this code snippet would be transformed to SSA form.

3.4 Language Independence

We define an algorithm as *language independent* if it can be executed on an abstract representation of the source code of a program that is not specific to a single language. A system that executes such an algorithm therefore needs a two-layer architecture: The lower layer is language specific and transforms the original program text into the abstract representation while the upper layer is language independent and only works on the abstract representation.

Listing 3.3: Example Java code.

```
1  class A {
2    void a(int x, int y) {
3      int z = x + y;
4      if (z > 2) {
5        return 1;
6      } else {
7        return z;
8      }
9    }
10 }
```

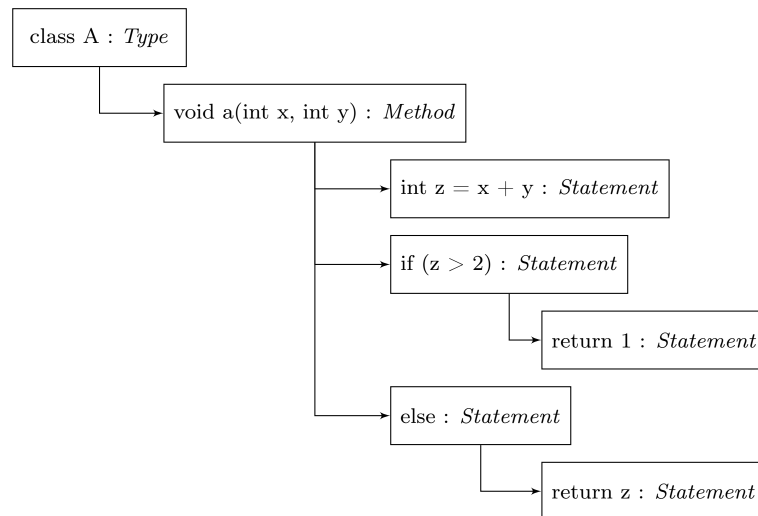


Figure 3.3: A tree of shallow entities, the result of shallow-parsing the code of listing 3.3. Each entity has the format "code : *type*."

3.5 Shallow Parser

Since we are working with source code, we need a way to parse it. It should be easy to create such a parser for a new language. The parsers should be fast and robust. For this we use the approach of shallow parsing. Unlike a full-fledged parser, *shallow parsers* do not create an abstract syntax tree (AST) that contains all the information present in the source code. Instead, they only parse structural information, e.g. which classes and methods a file contains. The most fine-granular structures a shallow parser can recognise are single statements in a method and their nesting within each other. Listing 3.3 and fig. 3.3 show how a Java file is transformed with a shallow parser into a tree of so-called *shallow entities*.

For our specific problem, a shallow parser offers several advantages over a full parser:

- For many languages there is no pre-built parser available that can be used by analysis tools and developing a shallow parser for a language needs significantly less

work than developing an accurate grammar for it.

- For many languages there is no specification available, which makes constructing a correct and complete grammar nearly impossible.
- A shallow parser is more robust than a full parser. As Ducasse et al. put it in [9]: Parsers are "technology that has proven to be brittle in the face of different languages and dialects". While a parser will break if the language's grammar is altered (e.g. from one release to the next) a shallow parser might still work or only need minor, trivial modifications.
- A shallow parser is usually significantly faster than a full parser as it does not need to understand the content of complex statements and constructs.

Shallow parsers have been used successfully in other projects in the software engineering context, e.g. to facilitate an incremental code search [15].

3.6 False Positive and False Negative

Data flow analysis is one example of a classification problem. Statements in the source code are classified as either "problematic" or "not problematic". To evaluate such a classifier, a reference classification is needed to which it can be compared, e.g. a human's opinion. This means that every statement in the code is rated twice as either *positive* or *negative*: once objectively and once by the classifier that is being evaluated. The rating of the classifier is then assessed as either *true* if it is consistent with the objective rating, or *false* if it is not.

The classifier may make two kinds of mistakes: First, it may wrongly categorise a correct statement as problematic. This is called a *false positive*. Second, it may wrongly categorise an incorrect statement as not problematic. Analogously, this is called a *false negative*.

Both types of errors have an effect on the usefulness of the classifier. Having too many false positives means that the user of the classifier will feel irritated, having to sort through a lot of false warnings. Having too many false negatives on the other hand means that many real problems are missed by the tool. Since for most problems, optimizing both false positive and false negative rates is impossible, any classifier thus faces a trade-off between the two.

3.7 Precision and Recall

The pure false positive and false negative numbers alone are usually not sufficient to judge a classifier's effectiveness. Instead, *precision* and *recall* are most commonly used for this purpose.

The precision of a classifier is the ratio of true positives to all items rated positive by the classifier, i.e. it measures how correct the classifier's assessment of the positive items

was:

$$precision = \frac{tp}{tp + fp}$$

where tp are the true positives and fp are the false positives.

The recall on the other hand is the ratio of true positives to all objectively positive items, i.e. it measures how complete the classifier's assessment of the positive items was:

$$recall = \frac{tp}{tp + fn}$$

where fn are the false negatives.

3.8 F_β Score

As with false positives and false negatives, there exists a trade-off between precision and recall. For most problems, optimizing one of them will decrease the other value. In order to contrast precision and recall, we can calculate the F_β-score. This score ranges from 0 (worst value) to 1 (best value) and "measures the effectiveness of [a binary classifier] with respect to a user who attaches β times as much importance to recall as precision" [22]. It is calculated with the following formula:

$$F_\beta = (1 + \beta^2) \cdot \frac{precision \cdot recall}{(\beta^2 \cdot precision) + recall}$$

By changing the value of β, we can thus see how well a classifier performs for different users.

4 Language Model

To be able to create a data flow framework that works for many different languages, we need a unified model of these languages. This model must contain structural requirements for the source code so it can be parsed into a CFG and describe the way data is referenced and transferred within the program. Furthermore, it must provide input for path-sensitive analyses based on the conditions that occur in the source code.

Figure 4.1 shows the entire language model. Each node in this figure represents a concept of the model. In the following sections, its different parts will be explained.

The goal of our model is to be as simple as possible, in order to allow an easy integration of new languages into the system. On the other hand, this means that we have to sacrifice analysis precision in some cases, since not all relevant information can be encoded in the model. We highlighted these instances throughout this chapter.

To make up for this loss, we will prioritise precision over recall wherever possible. In our experience this is justified as developers tend to lose interest in tools that have a high false positive rate, since that requires them to manually filter the reported findings.

4.1 Source Structure

In this thesis, we focus on intra-procedural data flow analysis, i.e. we perform the analysis on each method separately, without any knowledge of other methods or functions in the system. We are therefore only interested in the executable parts of the source code. Language constructs such as classes and their fields or package declarations etc. are not interesting to an intra-procedural data flow analysis. Thus we impose no restrictions or requirements on these parts of the source code.

This means that our analyses are potentially less accurate, because they lack information about the functions, classes, fields and methods the analysed function interacts with. On the other hand, this significantly simplifies creating a back-end for our framework as it does not have to parse classes and fields and does not have to infer type information. Especially the last part can be a complicated job in modern programming languages due to the use of generics or type inference. Furthermore, it is trivial to turn such a purely intra-procedural approach into an incremental analysis as it already operates locally. This means that if one wants to analyse several versions of the same source code, only the changed files have to be re-analysed when switching from one version to the next. This significantly decreases the analysis time and allows real time analysis.

For the executable parts, e.g. functions and methods, we require that a CFG must be constructible from the source code, i.e. the flow of control must be explicit in it. This excludes most functional languages as the flow of control is only given implicitly in the

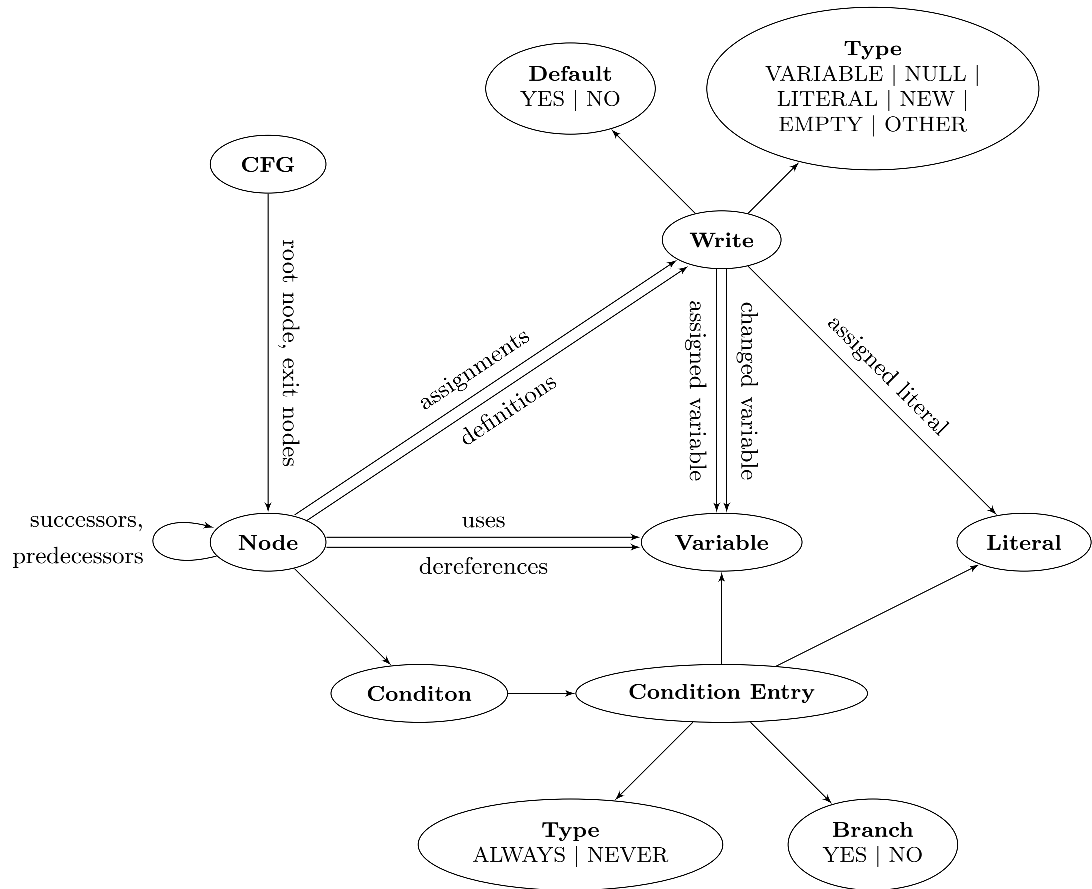


Figure 4.1: The language model, its concepts and their relationships. If only certain values are allowed for a concept they are listed under the concept's name.

source code through the use of higher order functions or closures which receive control from one another. [19] gives a more detailed explanation of the problems associated with functional control flow analysis as well as possible solutions and approximations.

We can still fit semi-functional languages, e.g. Ruby or JavaScript, into this model though, as control flow is apparent for most language constructs, with higher order functions being the most prominent exception. These will simply have to be ignored during the data flow analysis, which might lead to a higher false positive/negative rate.

When the flow of control is highly complicated or cannot be accurately predicted, we allow it to be approximated. This can be the case, for example, with error handling code. Often it is not easily possible to determine, which method invocations or operations cause a certain exception to be thrown and where it would be caught. Listing 4.1 shows such a situation in Java. Here, a `RuntimeException` might be thrown by any of the two statements inside the `try` block or by none. One of those methods might even throw such an exception unconditionally. It would be caught by the `catch` block and the statement in line 8 might never be reached. Because we want the model to allow an easy integration of

Listing 4.1: An example of complex control flow caused by exception handling in Java.

```
1 try {
2   method1();
3   method2();
4 } catch (RuntimeException e) {
5   handle(e);
6   return;
7 }
8 afterwards();
```

new languages into the framework, we do not require that this control flow is represented 100% accurately. Each language back-end may choose its own approximations in such situations. This will, of course, potentially affect the accuracy of our analyses, but will also significantly simplify writing a back-end for a new language.

Note also that we do not require that the source language be structured, i.e. languages that support the infamous `goto` or similar constructs are supported by our model as a CFG can be constructed even under such circumstances.

The CFG in our model is thus represented by a directed graph of nodes, where each node represents a single statement. The graph has exactly one root node and one exit node. Each node may have a number of successors and predecessors.

4.2 Data Reference

Data may be referenced in the source code only through variables. A variable can either directly or indirectly reference the data. The difference between these two styles of reference is best explained with an example from C/C++: direct reference is a variable on the stack while indirect reference is a pointer to data on the heap. No matter how the data is referenced, though, we treat all variables the same: as containers of data.

This means that we cannot track the address value of a pointer, but only the data at the memory location it points to. Thus, we cannot detect if two pointers point to the same memory location, which may reduce the quality of our analyses, but also simplifies them considerably.

4.3 Data Transfer

To be able to track the transfer of data between variables, we must track certain operations on them:

- The definition of a variable, including a possible assignment of an initial value
- Any assignments to a variable, i.e. the transfer of a value to it
- Any uses of a variable, i.e. any read of the value stored in it

- Any dereferences of pointer/reference variables, i.e. any access to the memory location they point to

We call definitions and assignments a *write* to the variable and reads and dereferences a *use*.

Each node within a CFG must thus be annotated with this def-use information. To understand the semantics of writes to variables requires an understanding of the values that may be transferred in these writes. It is hard to consistently model value types across multiple languages due to the differences between them. For example, an integer number may be 32 bit, 64 bit, signed, unsigned, etc. depending on the language or machine architecture. We therefore restrict ourselves to a simple model that does not care about the exact semantics of values or types. Instead we concentrate on several key aspects of value transfer that are important for our analyses.

For simplicity's sake we define any write to consist of exactly one left side and one right side, where the value obtained from evaluating the expression on the right is transferred to the variable on the left. More complex forms of assignment that are allowed in some languages – e.g. chain assignments (`a = b = c`), incrementing operators (`a += 2`), multi-assignments (`a, b, c = foo()`), etc. – must be transformed into this normal form. Based on the right side, we distinguish several types of writes, depending on the expression that is being assigned:

LITERAL These are assignments of literal, constant values that appear in the source code, such as numbers or literal strings. These values are simply represented as strings as we are not interested in their exact semantics.
Java example: `a = 12;`

VARIABLE These are assignments of a single variable to another variable, i.e. the value stored in the variable on the right is transferred to the variable on the left. The transferred value, which we cannot know without executing a data flow analysis, is therefore represented by the name of the right side variable (called the *assigned variable*).
Java example: `int a = b;`

NULL Some languages support the concept of a null-pointer, i.e. one that points to an invalid memory address, often represented as `0` or `null`. This category tracks the assignment of literal null values to pointer variables.
Java example: `a = null;`

NEW These are assignments of newly created objects to a variable:
Java example: `a = new String();`

EMPTY These are used to represent declarations of variables that do not assign any value to them.
Java example: `int a;`

OTHER Any other expression on the right side of the assignment that does not fall into one of the above categories is categorised here. These expressions are too complicated to resolve in the data flow analysis, e.g. a function call or some complex arithmetic expression.
Java example: `a = quicksort(b, c + 1, true);`

In addition to these categories, we store another flag for each data transfer: Some languages allow a variable to be initialised with a default value. Which value that is usually depends on the type of the variable. We want to be able to track both the actual value assigned to the variable and the fact that it was not assigned explicitly by the programmer. We therefore need a *default* flag for each assignment that is set whenever the variable was implicitly assigned a default value.

4.4 Conditions

In order to enable path-sensitive analyses, the model must include information about the conditions that guard the branches in the CFG, e.g. `if` statements or loops.

Fully understanding such a condition is in most cases neither possible nor necessary for the data flow analyses. Constructs such as type casts and function calls make such a full analysis of the condition impractical and complicated without significantly improving analysis results. Instead, we are only interested in whether or not the variables that occur in the condition will always or never have a certain value in a certain branch.

As an example, consider the Java statement `if (a != null)`. Understanding the condition, we can assume that in its yes branch, where the condition is `true`, `a` will never have the value `null`, while in the no branch it will always contain `null`.

Such information is thus represented as a list, where each entry contains the following information:

- The branch to which the information applies (YES branch or NO branch)
- The variable that is affected
- A value which is interesting for an analysis, e.g. `null`
- Whether that variable will ALWAYS or NEVER contain that value in that branch

Listing 4.2 shows an example of a condition and the list of condition entries that is stored for it. Each language heuristic may decide itself what kind of conditions it will parse. Very complex conditions may be ignored completely, which will of course cause the number of false positives/negatives to increase for some analyses. At the same time, however, it will greatly decrease the development time for a language back-end.

There are also other language elements that may contain information that is important for path sensitive analyses. One common example are assertions. A developer may have context knowledge that the analysis does not possess, for example that a certain list will always contain at least one item or that certain events can only happen in a particular

Listing 4.2: A simple condition in Java and the condition entries stored for it in our model.

```
1 if (a == null && b instanceof String && c != null)
```

```
1 YES branch, a, null, ALWAYS
2 YES branch, b, null, NEVER
3 YES branch, c, null, NEVER
```

Listing 4.3: A Java method from the ConQAT class `XMLUtils`.

```
1 /**
2  * Determines the index (starting at 0) of the given element relative to
3  * other element nodes for the same parent.
4  */
5 public static int getElementPosition(Element element) {
6   int num = -1;
7   Node node = element;
8   while (node != null) {
9     if (node.getNodeType() == Node.ELEMENT_NODE) {
10      ++num;
11    }
12    node = node.getPreviousSibling();
13  }
14  return num;
15 }
```

order. This knowledge may be encoded in assertions, e.g. `assert a != null`. Our model does not support such annotations because they would complicate the path-sensitive analyses. This may increase the number of false positives, since some information is missing. On the other hand, assertions are in our experience used sparingly. We therefore prefer a simpler model and simpler analyses to having a slightly higher precision.

4.5 Worked Example

In the following we will give an example of how this model would be applied to the Java code shown in listing 4.3. First of all, the CFG would consist of the nodes shown in fig. 4.2, with `Element element` being the root node of the CFG and `return num;` being the only exit node. This figure only shows the successor relationship between the nodes.

Each of these nodes has some def-use annotations:

Element element This node has a definition of the variable `element` with type OTHER.

int num = -1; This node has a definition of the variable `num` with type LITERAL and the assigned literal `-1`.

Node node = element; This node has a definition of the variable `node` with type VARIABLE and the assigned variable `element`. It also has a read of `element`.

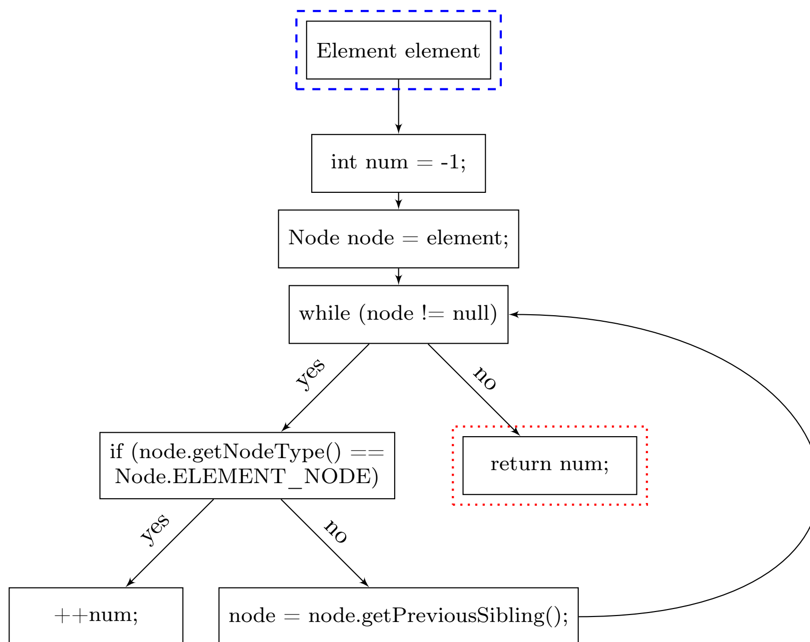


Figure 4.2: The CFG nodes of our model created for the code in listing 4.3. The entry node of the CFG is marked with a blue dashed line and the exit node with a red dotted line.

while (node != null) This node has a read of the variable `node`. It also has a condition with two entries:

```
node, YES branch, null, NEVER
node, NO branch, null, ALWAYS
```

if (node.getNodeType() == Node.ELEMENT_NODE) This node has one read and one dereference of variable `node`, as well as a condition with no entries.

++num; This node has an assignment to the variable `num` with type OTHER. It also has a read of `num`.

node = node.getPreviousSibling(); This node has an assignment to the variable `node` with type OTHER. It also has a read of `node`.

return num; This node has a read of the variable `num`.

5 Data Flow Framework

In this chapter we describe the general framework we created based on the language model we detailed in the previous section. It processes the source code, extracts all necessary information from it and executes the data flow analyses to obtain findings that describe data flow anomalies, i.e. possible problems. The framework is structured into several distinct parts, which are executed in series. This architecture is depicted in fig. 5.1. As the diagram shows, the entire framework is separated into language dependent and language independent parts.

5.1 Implementation

We based the implementation of our framework on ConQAT [7] and its incremental extension Teamscale [11], a quality analysis engine written in Java, which allows a user to determine quality metrics of a software system using a pipes and filters approach. These pipes and filters are implemented as Java classes called *processors*. For the implementation of our approach we used the existing infrastructure provided by the ConQAT framework, including several lexers and shallow parsers.

5.2 Language Dependent Parts

These components transform the original source code into internal data structures derived from our language model that capture all necessary information for the data flow analyses. The first step in this process is to tokenise the source code, i.e. split it into a sequence of coherent units like keywords, operators, literals and identifiers. This is done by the so-called *lexer* using a simple grammar that describes these tokens.

Following this, the stream of tokens generated by the lexer is passed to the shallow parser, which transforms it into a tree of shallow entities. These parsers build on a common framework that makes it easy to create parsers for new languages. We were able to reuse the existing lexers and shallow-parsers of ConQAT. From this shallow entity tree, we extract all sub-trees that represent executable code, i.e. methods and functions.

In the next step, the shallow entities of each method are converted into a CFG that is annotated with def-use information as described in our language model. We created a framework that simplifies this step. Implementers only have to specify rules for each control flow structure the language offers and can reuse existing code if these structures behave similarly to other languages. ABAP, C# and Java for instance all share source code to transform `if` and `while` statements.

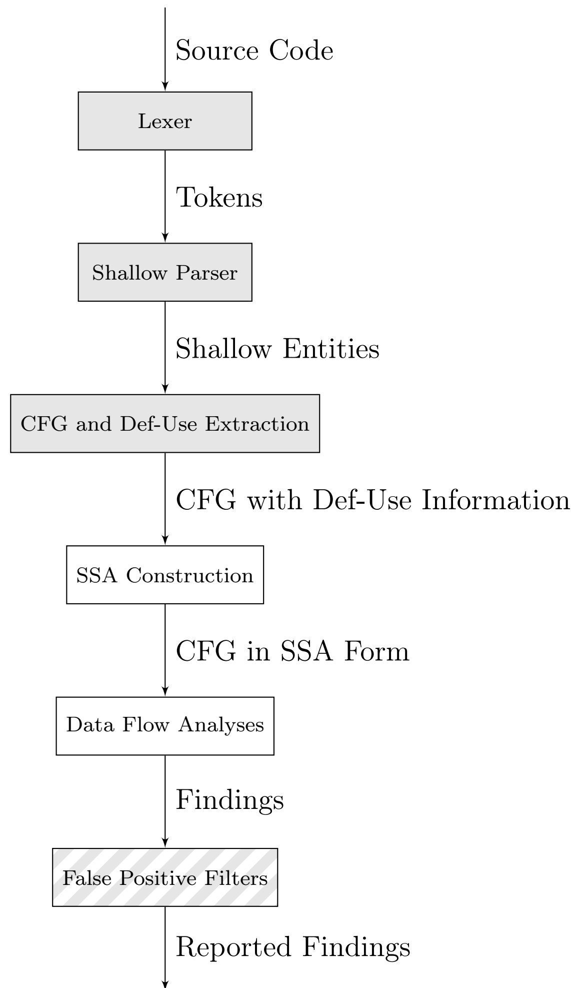


Figure 5.1: The architecture of the data flow framework. The white boxes represent language independent parts of the framework, while coloured boxes are language dependent. The false positive filters can be both.

The extraction of the def-use information on the other hand differs widely between languages and reuse is much lower since the syntactical representation of this information varies a lot. Assignments in Java for example are always represented using the `=` operator, e.g. `a = b + 1`, while in ABAP assignments are often keyword-based, e.g. `MOVE b + 1 TO a`. To obtain this information, the heuristics work on the tokens associated with a shallow entity, since the entity itself only contains structural information. The nodes in the CFG are then annotated with this information.

In the following sections we give a short overview over how the def-use heuristics for different languages work. We implemented back-ends for three programming languages: Java, C# and ABAP. We chose them to have two languages that are very similar to each other (Java and C#) as well as languages that are very different (Java/C# vs. ABAP).

5.2.1 Java and C# Heuristic

The Java and C# def-use heuristics work very similarly. This allows them to share most of their code. In order to extract the def-use information for a statement, they keep track of all variables that are currently in scope. This allows them to easily identify variables in the token stream. When the current scope, e.g. an `if` statement, is closed, all variables declared therein are removed from the running list of known variables by the heuristic.

Each statement in the method body is processed in several stages:

1. It is scanned for variable definitions. The defined variables are then added to the scope.
2. It is scanned for variable assignments with one of the many assignment operators, e.g. `=`, `+=`, `*=`, etc. Such assignments can be arbitrarily complex. Not only are chain-assignments allowed, such as `a = b = c = 12`, but it is also possible to write multiple such chains in one line, separated by a comma: `a = b = c, d = e = 12`. Assignments may also happen inside arbitrary expressions, e.g. method calls: `a.foo(b = c = 12)`.
3. It is scanned for increment and decrement operations, i.e. `++` or `--`.
4. It is scanned for dereferences of known variables, i.e. accessing an object's fields and methods (e.g. `a.foo()`) or by reading a value from an array (e.g. `a[12]`).
5. It is scanned for reads of known variables, i.e. any occurrence of a variable that is not a simple assignment to it with the `=` operator.
6. Finally, for branching statements such as `if` or `while`, the branch condition is analysed. A simple heuristic detects interesting terms in the condition, e.g. `a == null`, and their con- or disjunction and negation.

The C# and Java heuristics both incorporate logic to handle constructs specific to the respective language, e.g. `synchronised` blocks in Java or `out` and `ref` parameters in C#.

5.2.2 ABAP Heuristic

In contrast to Java and C#, the ABAP heuristic is pattern-based. This is due to the fact that ABAP features a plethora of keywords, which can be combined in complex statements, many of which write results to one or more variables. Listing 5.1 shows several such statements. The heuristic also does not keep track of scopes like the Java and C# heuristics, since ABAP only has one shared scope for each method.

Each statement is thus processed in several steps:

1. It is scanned for variable definitions with the `data` or `field-symbol` keywords. The defined variables are added to the list of known variables.
2. It is scanned for in-line variable definitions, e.g. the variable `wa` in the statement `LOOP AT itab INTO DATA(wa)`. These variables are also added to the list of known variables.

Listing 5.1: Several ways to assign a value to the variable a in ABAP.

```
1 find first occurrence of 'a' in 'some string' results a.
2 a = 2.
3 a ?= 2.
4 move 2 to a.
5 write 'foo' to a.
6 clear a.
```

3. It is matched against a list of 72 patterns. The first matching pattern identifies the variable writes that occur in that statement.

4. It is scanned for variable dereferences, e.g. field symbol uses or reference accesses with the `->` operator.

5. It is scanned for variable reads, i.e. every use of a known variable that has not been matched as a write by the pattern.

6. Finally, for branching statements the branch condition is analysed. The heuristic used for this is very similar to the one used for C# and Java.

5.2.3 Parsing Conditions

In almost all languages, branches in the CFG are guarded by conditions. The syntax of these conditions is often very similar, even for languages that are otherwise very different. Thus, the parsing of such conditions is an ideal candidate for code reuse within our framework.

We therefore created a base class that handles the overall parsing of the condition – such as identifying single terms and the operators between them – while leaving the parsing of single terms to the language specific subclasses. Our base class can recognise, for example, that `if (a == null && isUpperCase(b) && c instanceof String)` is a logical conjunction of three terms, while its Java-specific subclass can tell that if the condition is true, `a` must be `null` and `c` will never be `null`. All three of our def-use heuristics use this base class.

5.2.4 Exception Handling

As mentioned in section 4.1, we made a trade-off between the simplicity of creating a new language back-end and the accuracy of the CFG. The way we create the CFG for exception handling constructs exemplifies this. In the following, we will detail these approximations for Java, but the other language back-ends we implemented handle this similarly.

Theoretically, almost any line within a `try` block may throw an exception. If the `try` block is long or contains complex control flow, e.g. loops, this leads to a plethora of possible paths from within the `try` block to the `catch` and `finally` blocks. Instead of modelling this accurately, we chose to only model an exception being thrown in the very

first statement. This means that in our model, either the entire `try` block is executed or control is passed directly to one of the `catch` or `finally` blocks, without executing any code within the `try`.

Furthermore, we approximated the behaviour of `finally` blocks. At the end of such a block, control will either go to the statement after the exception handling or it will leave the method, depending on whether the `try`/`catch` block that passed control to the `finally` block ended in a `return` statement or not. This means that if some blocks end in a `return` and some don't, we would have to duplicate the `finally` subgraph to separate those two cases. We chose not to do this to make the implementation simpler. Both of these approximations make developing a new back-end for a language easier, whilst sacrificing some analysis precision.

5.3 Language Independent Parts

The results obtained from the language dependent parts are passed to the language independent SSA construction algorithm that transforms the CFG into its SSA form using the annotated def-use information. We use static single-assignment form because it greatly simplifies many of the data flow analyses we implemented, since every variable only has a single definition.

5.3.1 SSA Construction

In order to transform the annotated CFG obtained from the language dependent heuristics into its SSA form, we employ the simple algorithm described by Aycock and Horspool in [4]. It first finds all basic blocks and creates a new version of every variable at every basic block boundary. Afterwards it inserts a ϕ-node at each of those boundaries for every SSA variable. These are obviously too many ϕ-nodes. Therefore the algorithm includes a minimisation phase where useless nodes are removed iteratively. This makes sure that we get all necessary ϕ-nodes, but not necessarily a minimal amount. It is a much simpler algorithm than, e.g. the iterated dominance frontier algorithm of Cytron et. al. [6], which will always place a minimal number of ϕ-nodes.

We also implemented the first two improvements as presented in Aycock and Horspool's paper. Furthermore, we added some index data structures to the final SSA form to speed up its construction and the analyses run on it. The final SSA form consists of a directed graph of basic blocks, each of which contains a list of statements that in turn contain writes and uses of variables.

5.3.2 Analysis Framework

This SSA form is passed to the data flow analysis framework, which executes all analyses in turn and gathers their results. The analysis process is iterative in nature. Akin to [1] we separate it into two components: a driver and the analysis logic. The driver is the same for all analyses and simply executes the analysis repeatedly for every basic block in the SSA form, until it has obtained a stable result. Some analyses, like the self assignment

analysis, only need to be executed once for every basic block, while others, e.g. the null pointer analysis, need to propagate information between basic blocks and therefore might need several iterations to complete.

The final result is a list of findings that describe possible problems in the source code with a message, e.g. "the value written to variable 'foo' is never read", and information about the location of the problem in the code.

5.3.3 Analyses

For each analysis we created, we give a brief overview over its purpose, followed by a description of how we implemented it. As a guideline while choosing which analyses to implement, we used the list of FindBugs detectors[1]. This list contains all analyses that FindBugs is able to execute. From it, we first selected all detectors that perform a data flow analysis. These we sorted according to how easy they would be to implement. From this list we chose three: two flow-sensitive analyses that were straight-forward to create and a more complex path-sensitive analysis.

All analyses were designed so they can be reused, i.e. one analysis can be run on many methods. The resulting findings are collected by our framework. This allows a whole system to be analysed in one go. Furthermore, we optimised the precision of our analyses by taking deliberate steps to remove false positives – sometimes at the cost of increasing the number of false negatives.

Dead Store Analysis

This flow-sensitive analysis tries to identify locations in the source code where a value is written to a variable but that value is never read – e.g. because it is immediately overwritten with another or neither overwritten nor read until the end of the method.

It takes into account assignments of default values by the compiler, e.g. in ABAP. For such variables, it is assumed that overwriting the default value without reading it in between is allowed as this is standard practice and cannot be avoided in some cases.

The implementation of the analysis on the SSA form of a method is achieved rather easily: Since all variables have exactly one definition location, we may simple enumerate all used variables and subtract that set from the set of all variables. The resulting set contains all unused variables. From this we then subtract all default initialised variables that were overwritten and thus obtain the set of all dead stores.

The only complication arises when ϕ-nodes are present in the SSA form, e.g. $a_3 = \phi(a_1, a_2)$. In this case, if there is a use of variable a_3, the variables in the ϕ-node must be considered as used as well. This transfer of the *used* property must be executed repeatedly as variables in ϕ-nodes may in return reference other ϕ-nodes.

There are several limits to the dead store analysis: Since it does not know about branch feasibility, it may wrongly categorise variables as alive when they can actually never be overwritten since all branches that do so are infeasible. This would lead to an increase

[1] Available at: `http://findbugs.sourceforge.net/bugDescriptions.html` last accessed 2013-10-17

in false negatives. Furthermore, due to the limits of our language specific heuristics, an incorrect SSA form may be constructed, which may lead to an increase in false positives.

Self Assignment Analysis

This flow-sensitive analysis tries to find statements that assign a variable to itself, e.g. `a = a`. Implementing it is simple: We look at all assignments of one variable to another in the SSA form and those that contain a variable with the same name on the left and on the right side are reported as findings. This simplistic detector is likely to produce false negatives as there are many more syntactically different but semantically equivalent ways of writing such a self assignment. In Java for example, `a = a++` is also a self assignment (as the increment is overwritten by the assignment). Furthermore, self-assignments via temporary variables are also not tracked, e.g. `b = a; a = b`.

Null Pointer Analysis

The null pointer analysis is a path sensitive analysis that tries to find locations in the source code where a variable containing a null value is dereferenced. These dereferences are faults as that memory location is not valid and a dereference will result in an exception being thrown or the termination of the program.

In order to track null values throughout the source code, we need to define a notion of *nullness*, i.e. whether a variable may hold a null value at a specific point in the CFG or not. Since the answer to this depends on the exact path taken throughout the CFG to reach that point, this cannot be answered with a simple yes or no.

We therefore define the nullness of a variable at a certain point in the source code as one of the following values:

always The variable will always be null at this point, no matter which path is taken to reach it.

never The variable cannot be null at this point, no matter which path is taken to reach it.

sometimes There is at least one path to this point that makes the variable null, but not all paths will make it null.

unknown There was no indication of whether the variable can be null at this point and we therefore cannot assume any of the other options.

We furthermore apply a heuristic: If a variable is checked for null, we assume that, since the developer checked for it, the variable may also be null, i.e. its nullness is changed to *sometimes* if it would otherwise be *unknown*.

Our analysis tries to determine the nullness of every variable within every basic block in which it is dereferenced. Since in SSA form every variable is assigned to only once, it is enough for us to track the nullness of a variable relative to a basic block. The information used to compute the nullness does not change within a linear sequence of statements as

only branch conditions can influence the nullness of a variable after its definition. If the nullness of the variable at its dereference location is *always* or *sometimes*, we report a finding as we are sure that there is at least one path through the CFG which will make the variable null. Note that in both cases this path may be infeasible due to some information we did not consider in our analysis, thus making the finding a false positive.

In order to determine the nullness of a variable within a basic block, we need to have the following information:

- Information about the value assigned to the variable at its definition point in SSA form.
- Information about the paths through which the basic block may be reached and the null-checks performed along these paths.

In order to obtain the definition information, we need to decide the variable's nullness within the basic block in which it is defined. If a ϕ-node or another variable is assigned to the variable in that block, this may require deciding the nullness of other variables within the definition block. This is therefore done in a recursive manner until all necessary variables have been resolved to either literals or expressions. Infinite recursions are possible when resolving these variable dependencies, since loops in the CFG may cause a variable assignment loop. To prevent this scenario, we simply assume a value of *unknown* when we encounter a variable for the second time while resolving.

In order to obtain the path information, we need to look at conditional checks for nullness, e.g. `if (a == null)` in Java. For these, we need to know along which of the outgoing CFG edges the variable may be null. For example, a dereference of the variable `a` in the then branch of the above `if` statement would surely be a null pointer dereference while in the else branch it can never be one.

This information is then combined to produce the final nullness value of the variable at the dereference location. For example, in the code in listing 5.2, the nullness of variable `a` in line 6 is *never* since if it were null in line 1, it would be set to a non-null value in line 4. Variable `b` on the other hand has a nullness of *sometimes* in line 7. At its declaration site in line 2 it receives a nullness of *unknown*, since we do not know anything about `getB()`. Then a null-check is performed on it in line 3, which increases its nullness to *sometimes*. Since no other value is assigned to it anywhere, this is the nullness that reaches the dereference location of `b`. Therefore, a null pointer finding is created for variable `b` at line 7, but not for variable `a`.

5.4 False Positive Filters

A special case are the false positive filters, which can be both language independent (filters that work for exactly one analysis, regardless of the underlying language) and language dependent (filters that take special features of a language into account). Each finding produced by the data flow analyses, along with detailed information about the location in the SSA form where they occurred, is passed to these filters. For each such

Listing 5.2: A Java example of obtaining the nullness value of a variable.

```
1 SomeClass a = getA();
2 SomeClass b = getB();
3 if (a == null || b == null) {
4   a = new SomeClass();
5 }
6 a.dereference();
7 b.dereference();
```

finding, the filter may indicate that it is a false positive. In this case, the finding is not reported to the user.

We implemented several such filters, all of which are described here shortly. Please note that since we optimised our analyses for precision, these filters prefer filtering out too much over leaving possible false positives in the source code.

5.4.1 Dead Store Analysis

We implemented a filter for dead store findings on statements that contain more than one increment of the same variable, e.g. `method(i++, i++)`. Such double-increments are always going to be reported as dead stores by our analysis. Due to the way our SSA conversion algorithm works, such a statement will be converted to:

$$i_1 = i_0 + 1$$

$$i_2 = i_0 + 1$$

As can be seen, the variable i_2 incorrectly references i_0 instead of i_1. All subsequent uses of i will therefore reference the variable i_2 and i_1 will never be used.

5.4.2 Null Pointer Analysis

For the null pointer analysis, a common false positive is a condition that first checks if a variable is null and dereferences it only if that check succeeds, e.g: `if (a == null || a.isEmpty())` These types of expression work due to their use of short-circuit logic, i.e. the second part of the condition is only evaluated if the first part does not already determine the value of the entire expression. Since this is a very common idiom in many languages, we implemented a general filter for it that can be used regardless of the programming language.

The filter identifies all conditions that both dereference a variable and check for its nullness. Findings for such variables are ignored. This, of course, is a very coarse filter and there are several types of false negatives it may produce, most notably:

- Using the wrong boolean operator, e.g. `if (a == null && a.isEmpty())` instead of `if (a == null || a.isEmpty())`

Listing 5.3: An ABAP report with variables that are only used in its forms.

```
1 report Z_REPORT.
2
3 data a type i.
4 a = 12.
5 perform main.
6
7 form main.
8   write: 'a=', a.
9 endform.
```

- Wrong order of the checks, e.g. `if (a.isEmpty() || a == null)`

It also misses some false positives, e.g. conditional expressions inside a ternary operator or as arguments to functions.

5.4.3 Java

For Java, 5 different filters were implemented, which filter the following types of findings:

- Null pointer findings on variables that are checked with `instanceof` in the same statement in which they are being dereferenced, e.g:
  `if (a instanceof String && a.length > 0)`
- Dead store findings on the arguments parameter of a `main` method
- Dead store findings on parameters of methods annotated with `@Override`
- Dead store findings on parameters annotated with `@SuppressWarnings("unused")`
- Dead store findings on parameters of template methods, i.e. methods that may be overwritten by subclasses

5.4.4 ABAP

For ABAP, a common false positive reported by the dead store analysis are variable declarations in a report that also contains forms. A report is a program that can be executed. Forms are subroutines that may be called from the report and that may also access the variables declared in the report. See listing 5.3 for an example. Most of the dead store findings in such reports are false positives.

To remove them, we created a false positive filter that checks if the method that contains the finding is a report and if that report furthermore has at least one form. If that is the case, all dead store findings are simply ignored. This behaviour will, of course, produce a number of false negatives since it does not make sure that the variables for which all findings are ignored are actually used inside the forms.

5.4.5 C#

C# allows so-called `out` and `ref` method parameters. These pass their argument by reference instead of by value. Thus, a method call with such a parameter may write to its argument. To prevent false positives in the dead store analysis for these parameters, which are very common, we filter out all findings produced for them in the statements that invoke such a method.

Furthermore, 5 other filters were implemented, which filter the following types of findings:

- Null pointer findings on variables that are checked with the `is` operator in the same statement in which they are being dereferenced, e.g:
  ```
  if (a is string && a.length > 0)
  ```
- Dead store findings on the arguments parameter of a `Main` method
- Dead store findings on parameters of methods annotated with `override`, `virtual` and `new` keywords
- Dead store findings on parameters of template methods, i.e. methods that may be overwritten by subclasses
- Dead store findings on parameters of event handler methods

6 Evaluation

In this section, we will describe how we evaluated our approach. We will first present the research questions that guided the evaluation and then show how we designed and executed a case study to answer these questions. Finally, we will discuss its results and show the limits of our approach.

6.1 Research Questions

The design of the evaluation was driven by the following research questions:

RQ1. How easy is it to integrate another language into our framework? For our tool to be useful in a context where many different, possibly undocumented languages have to be analysed, a high amount of reuse in the language specific parts of our framework is desirable. If developing the back-end for a new language is easy and can build on the experience gathered in implementing other language specific heuristics, we expect that our tool is useful in such scenarios.

RQ2. How well do our implemented analyses perform in comparison with established tools? In order to be useful to a developer, our tool has to produce accurate results within a reasonable amount of time. Therefore, we want to see how our analyses perform when compared to language specific tools that have been optimised for a low number of false positives and are widely used today. We will look at two specific criteria for each tool:

a) precision, i.e. how many false positives do they generate, and recall, i.e. how many of the actual problems do they find
b) execution time, i.e. how long does it take to analyse a system

RQ3. Do our analyses perform equally well on different languages? The intent of our tool is to provide the same analyses on a variety of different languages. Consequently, the results produced by these analyses should have the same quality, independent of the analysed language. We therefore want to know whether our analyses perform similarly for three different languages based on two criteria:

a) precision, i.e. how many false positives do they generate
b) execution time, i.e. how long does it take to analyse a system

6.2 RQ1: Integrating another language into our framework

In this research question we determine how easy it is to create a new back-end for our framework to support another programming language.

6.2.1 Study Design

To get an estimate of the effort necessary to implement a back-end for another language in our tool, we chose to measure the code reuse between the three already implemented heuristics, assuming that a higher amount of reuse means less effort when developing a new back-end. We therefore obtained the number of source lines of code (SLOC) of our tool and split these into three categories:

- The code that is specific to Java.
- The code that is specific to ABAP.
- The code that is specific to C#.
- The code that is shared between the languages. This includes any classes that are used by more than one language specific heuristic, e.g. common base classes and utilities. The language independent parts of the system and some of the false positive filters are also part of this category.

We then calculated the percentage of code reuse that is possible when implementing a new language back-end.

For many languages it is possible to reuse existing lexers and parsers. To also get a feeling for how much reuse we can expect in such a situation, we calculated the same values again, this time leaving out the lexer and parser code. Furthermore, we took a look at what kind of code is shared between the heuristics and what has to be reimplemented for each new language.

6.2.2 Results

Table 6.1 summarises the measured SLOC for the different parts of our system. The shared code is between 46 and 58% of the entire system. This means that for a new language, most of the system can be reused and only around 400 to 2400 SLOC have to be implemented.

Code that is shared between the different language back-ends includes the CFG construction framework, CFG construction rules for commonly used languages elements (e.g. `if` statements, `while` or `for` loops, `switch` statements, etc.) and code for condition parsing. The code that is specific to only one language includes mostly parsing and CFG construction for special language elements, e.g. annotations, complicated type expressions or special condition terms like `instanceof` in Java or `is initial` in ABAP. For ABAP, there is also special parsing for the plethora of its keywords and the statements that can be constructed with them. Thus, the framework allows the developer to focus on the

	Without Lexer/Parser		With Lexer/Parser	
	SLOC	Percent	SLOC	Percent
ABAP Heuristic	1112	14%	2393	23%
Java Heuristic	435	6%	962	9%
C# Heuristic	1645	21%	2188	21%
Shared Code	4487	58%	4766	46%
Entire System	7679	100%	10309	100%

Table 6.1: The amount of SLOC measured for different parts of our tool, once including the respective lexer and shallow parser and once without.

specialities of a language, as they can reuse code for common elements that are present in many languages with minimal effort.

6.3 Studied Systems

For research questions 2 and 3 we used several different systems on which we executed our analyses. Table 6.2 shows all of these systems and their respective size and programming language. We excluded generated and test code from the analyses.

6.4 Reference Tools

To compare our tool against the state of the art in RQ2, we chose FindBugs 2.0.3[1] and FxCop 12.0[2] as a reference. FindBugs is a well-established tool that performs (among other things) a diverse set of data flow analyses on Java bytecode. Several papers have been written about the tool and its null pointer analysis has been given special attention to reduce the number of false positives it produces [13].

FxCop is a tool created by Microsoft that analyses .NET code which has been compiled to its intermediate language format. FxCop offers much less data flow analyses than FindBugs. It mainly relies on simpler checks.

Unfortunately, all data flow tools for ABAP are, to our knowledge, not available freely. We were therefore not able to evaluate the precision and recall of our approach against a reference ABAP tool.

[1] Available at: `http://findbugs.sourceforge.net/` last accessed 2014-04-02
[2] Available at: `http://msdn.microsoft.com/en-us/library/bb429476(v=vs.80).aspx` last accessed 2014-04-02

System	Language	SLOC	Domain	OS	Available at
jEdit	Java	108298	Text editor	✓	http://www.jedit.org
Lucene Core	Java	89516	Search engine library	✓	https://lucene.apache.org/core
JabRef	Java	83773	Reference manager	✓	http://sf.net/p/jabref
SweetHome 3D	Java	78267	Interior design	✓	http://sf.net/p/sweethome3d
FreeMind	Java	55396	Mind mapping	✓	http://sf.net/p/freemind
SAPlink	ABAP	21463	Data exchange	✓	https://code.google.com/p/saplink
SAPChess	ABAP	2694	Chess program	✓	https://code.google.com/p/sapchess
ABAP Exporter	ABAP	2166	Data export		
SharpDevelop	C#	583774	.NET development environment	✓	http://sf.net/p/sharpdevelop
NHibernate	C#	159574	Object relational mapper	✓	http://sf.net/p/nhibernate
iTextSharp Core	C#	93685	PDF manipulation library	✓	http://sf.net/p/itextsharp
Firebird .NET Back-end	C#	31003	Database back-end	✓	http://sf.net/p/firebird
Wurfl	C#	24287	Device Detection Library	✓	http://sf.net/p/wurfl
Robocode	C#	14821	Programming game	✓	http://sf.net/p/robocode
Midi Sheet Music	C#	9220	Sheet music tool	✓	http://sf.net/p/midisheetmusic
Mono Calendar	C#	5183	Calendar application	✓	http://sf.net/p/monocalendar

Table 6.2: The systems which we used to evaluate our analyses, sorted by language and size. OS stands for Open Source.

6.5 RQ2a: Precision and recall compared to other tools

In this research question we compare our approach to FindBugs and FxCop in terms of its precision and recall.

6.5.1 Study Design

To be able to compare our analyses to those of FindBugs, we needed to create a mapping between the two. Table 6.3 shows the FindBugs detectors that correspond to our analyses. Each FindBugs detector is identified by a code, which can be looked up in the FindBugs detector reference[3].

For FxCop, we chose the two rules CA1801 and CA1804[4], which find unused parameters and unused local variables, respectively. They correspond to our dead store analysis. While FxCop does offer rules for detecting null pointer dereferences (C6011 and C28182), these unfortunately only work for C/C++ code.

In order to judge the usefulness of our approach to developers, we ran FindBugs and our tool on all Java systems and FxCop and our tool on NHibernate, Mono Calendar and Wurfl[5]. We obtained all findings the tools produced on these systems. For each language, we fairly sampled one third of the findings. We classified all sampled findings as belonging either to one specific tool or to both, depending on whether only one or both tools had reported that finding. The resulting finding counts per tool and analysis are shown in table 6.4 for Java and table 6.5 for C#. The self assignment analysis did not produce any findings on any system and was thus excluded from this research question.

We then manually assessed all findings as either correct or incorrect. With this information we were able to calculate the number of true and false positives produced by all tools and thus their precision per analysis.

To demonstrate that our tool does not over-optimise the precision, we furthermore approximated the recall of the tools. Since we cannot list all places in the source code where a particular analysis should have reported a finding, we assume that the set of true positives identified by both our tool and the reference tool together are a suitable approximation of this number:

$$recall = \frac{tp_1 + tp_{12}}{tp_1 + tp_2 + tp_{12}}$$

where tp_1 and tp_2 are the true positives identified exclusively by tool 1 and tool 2 respectively and tp_{12} are the true positives identified by both tools. To contrast precision and recall, we calculated the F_β score of each tool per analysis. We chose $\beta = 0.5$, since

[3]Available at: `http://findbugs.sourceforge.net/bugDescriptions.html` last accessed 2013-10-17

[4] A full list of all FxCop rules is available at `http://msdn.microsoft.com/en-us/library/ee1hzekz.aspx` for managed code and at `http://msdn.microsoft.com/en-us/library/a5b9aa09.aspx` for C/C++ code. Last accessed 2014-03-25.

[5]Since could not get the other C# systems to compile without errors, we were unable to use them in this research question.

Our Analysis	Corresponding FindBugs Analyses
Dead Store	IP_PARAMETER_IS_DEAD_BUT_OVERWRITTEN
	DLS_DEAD_LOCAL_STORE
	DLS_DEAD_LOCAL_STORE_OF_NULL
	SA_LOCAL_DOUBLE_ASSIGNMENT
	SF_DEAD_STORE_DUE_TO_SWITCH_FALLTHROUGH
	DLS_DEAD_LOCAL_STORE_IN_RETURN
	DLS_OVERWRITTEN_INCREMENT
	DLS_DEAD_LOCAL_INCREMENT_IN_RETURN
	DLS_DEAD_STORE_OF_CLASS_LITERAL
	DLS_DEAD_LOCAL_STORE_SHADOWS_FIELD
	SF_DEAD_STORE_DUE_TO_SWITCH_FALLTHROUGH_TO_THROW
Self Assignment	SA_LOCAL_SELF_ASSIGNMENT
	SA_LOCAL_SELF_ASSIGNMENT_INSTEAD_OF_FIELD
Null Pointer	NP_ALWAYS_NULL
	NP_ALWAYS_NULL_EXCEPTION
	NP_ARGUMENT_MIGHT_BE_NULL
	NP_GUARANTEED_DEREF
	NP_NULL_ON_SOME_PATH
	NP_GUARANTEED_DEREF_ON_EXCEPTION_PATH
	NP_NULL_ON_SOME_PATH_EXCEPTION

Table 6.3: Our analyses and the corresponding FindBugs detectors.

	FindBugs		**Our Tool**		**Both**	
	Total	Sample	Total	Sample	Total	Sample
Dead Store	3	0	518	127	65	23
Self Assignment	0	0	0	0	0	0
Null Pointer	6	2	120	41	5	0

Table 6.4: The number of findings produced and sampled per tool and analysis on Java systems.

	FxCop		**Our Tool**		**Both**	
	Total	Sample	Total	Sample	Total	Sample
Dead Store	58	22	126	50	97	21

Table 6.5: The number of findings produced and sampled per tool and analysis on C# systems.

	FindBugs		**Our Tool**		**Both**	
	TP	FP	TP	FP	TP	FP
Dead Stores	0	0	164	8	23	0
Null Pointer	2	0	14	27	0	0

Table 6.6: The number of true (TP) and false positives (FP) per analysis for FindBugs and our tool on Java systems.

we are interested in a high precision at the cost of a lower recall, and contrasted this with $\beta = 1$. $\beta = 0.5$ means that we value precision twice as much as recall and $\beta = 1$ means that we value both the same.

Finally, we took a deeper look at the causes of false positives for our tool.

6.5.2 Results

Table 6.6 shows the number of true and false positives per tool and analysis on Java systems, while table 6.7 shows them on C# systems. From these values, we calculated the precision, recall and F_β values of each tool per analysis, as shown in table 6.8.

It is easy to see that the dead store analysis consistently performs very well compared to both reference tools, with very high precision, recall and F_β values. Our null pointer analysis on the other hand has a much higher recall, but a lower precision and $F_{0.5}$ value than FindBugs'. Thus, the more someone values precision over recall, the more interesting FindBugs' null pointer analysis becomes. At $\beta = 0.5$, it is slightly superior to ours.

Next, we looked at the reasons why our tool produced false positives. Tables 6.9 and 6.10 summarise the reasons for the dead store and null pointer analysis respectively. We can see that for the dead store analysis, the most common reasons across all languages are

	FxCop		**Our Tool**		**Both**	
	TP	FP	TP	FP	TP	FP
Dead Stores	18	1	46	4	24	0

Table 6.7: The number of true (TP) and false positives (FP) per analysis for FxCop and our tool on C# systems.

	Precision		Recall		F_1		$F_{0.5}$	
	OT	RT	OT	RT	OT	RT	OT	RT
	Comparison with FindBugs on Java systems							
Dead Stores	96%	100%	100%	12%	0.98	0.22	0.97	0.41
Null Pointer	34%	100%	88%	13%	0.49	0.22	0.39	0.42
	Comparison with FxCop on C# systems							
Dead Stores	95%	98%	52%	20%	0.67	0.34	0.81	0.56

Table 6.8: The precision, recall and F_β values calculated for our tool (OT) and the reference tools (RT) FindBugs and FxCop per analysis.

Language	Reason	Amount
Java	Unrecognised `@SuppressWarnings`	4
	Complex exception handling	2
	Definition of a class inside a method	1
	Misparsed complex statement	1
ABAP	Misparsed `define` statement	1
C#	Misparsed preprocessor directive	4

Table 6.9: The reasons for false positives produced by our dead store analysis.

errors in the interpretation of certain language constructs, e.g. conditions or preprocessor directives. For Java, our simplified CFG for exception handling also caused 2 false positives.

Listing 6.1 shows an example taken from Lucene[6] of the complex interaction of try-catch-finally constructs in Java that are not correctly represented in our CFG. Because both the `try` and `catch` branch use the `finally` clause, our tool cannot tell whether or not the control flow coming into the `finally` clause can reach line 9. It therefore reports a null pointer finding for `buffer` in that line, even though the `throw` statement in line 5 prevents this.

For the null pointer analysis, dependencies between conditions are by far the most common reason for false positives. Our simplified condition parsing and exception handling also negatively affected its precision.

An example for a condition dependency is shown in listing 6.2. This code is taken from SweetHome 3D. Our tool reports a null pointer finding for the variable `name1` in line 6, because it does not recognise the dependency between the `if` and the `else if`, which makes it impossible for `name1` to be `null` at this point. If `name1` were `null` in that line, `name2`

[6]We removed irrelevant lines from the code snippet to make it easier to read

Language	Reason	Amount
Java	Condition dependency	22
	Complex condition	2
	Complex exception handling	1
	Definition of a class inside a method	1
	Unrecognised `assert` statement	1
ABAP	Condition dependency	2
C#	Condition dependency	46
	Complex condition	8
	Complex lambda expression	1
	Unrecognised standard library method	1
	Misparsed preprocessor directive	1

Table 6.10: The reasons for false positives produced by our null pointer analysis.

must be `null` as well or the `if` branch would have been taken. This, however, means that the expression `name1 != name2` is `false` and the second part of the `else if` condition will not be evaluated.

6.6 RQ2b: Execution time compared to other tools

In this research question we compare our approach against FindBugs and FxCop in terms of their execution time.

6.6.1 Study Design

To measure the execution time, we ran our tool and FindBugs three times on all 5 Java study systems mentioned in RQ2a with the analyses listed in table 6.3. Furthermore, we ran our tool and FxCop three times on the 3 C# study systems from RQ2a with only their respective dead store analyses enabled. We timed each execution and aggregated them on their minimum value in order to get measurements with as little background noise as possible (e.g. from system processes that run at the same time). This study was executed on a 64bit notebook with an Intel® Core™ i5-3317U CPU. FxCop was executed under Microsoft Windows 7, the other tools under Ubuntu Linux 12.10.

6.6.2 Results

Figure 6.1 shows the minimum of the running times measured per system for all three tools. Our analysis constantly performs better time-wise than both reference tools. In

Listing 6.1: An example of exception handling that our approach does not represent correctly in the CFG. A false positive null pointer finding is generated for `buffer` in the highlighted line.

```
1 CachingTokenFilter buffer = null;
2 try {
3   buffer = new CachingTokenFilter(source);
4 } catch (IOException e) {
5   throw new RuntimeException("Error analyzing query text", e);
6 } finally {
7   IOUtils.closeWhileHandlingException(source);
8 }
9 buffer.reset();
```

Listing 6.2: An example of a dependency between two conditions that our approach does not recognise. A false positive null pointer finding is generated for `name1` in the highlighted line.

```
1 String name1 = this.appearance.getName();
2 String name2 = appearance2.getName();
3 if ((name1 == null) ^ (name2 == null)) {
4   return false;
5 } else if (name1 != name2
6            && !name1.equals(name2)) {
7   return false;
8 }
```

comparison with FindBugs, it is between 10 and 14 times as fast. Compared to FxCop it is between 2 and 6 times as fast. This confirms our believe that using a shallow parser and heuristics instead of constructing a full AST greatly reduces the analysis time.

6.7 RQ3a: Precision and recall on different languages

In this research question we compare the precision of our approach on Java, ABAP and C# systems.

6.7.1 Study Design

In order to compare the precision of our tool on all three languages, we needed to sample some more findings for our tool on ABAP and C#. For ABAP, all three analyses needed to be sampled and evaluated, for C# and Java, the results from RQ2a could be reused. Thus, for C# only the self assignment and null pointer analysis needed to be evaluated.

For each analysis/language combination where the results from RQ2a could not be reused, we sampled up to 100 findings from all ABAP and C# systems listed in table 6.2. Table 6.11 shows the number of findings sampled across all systems per language and

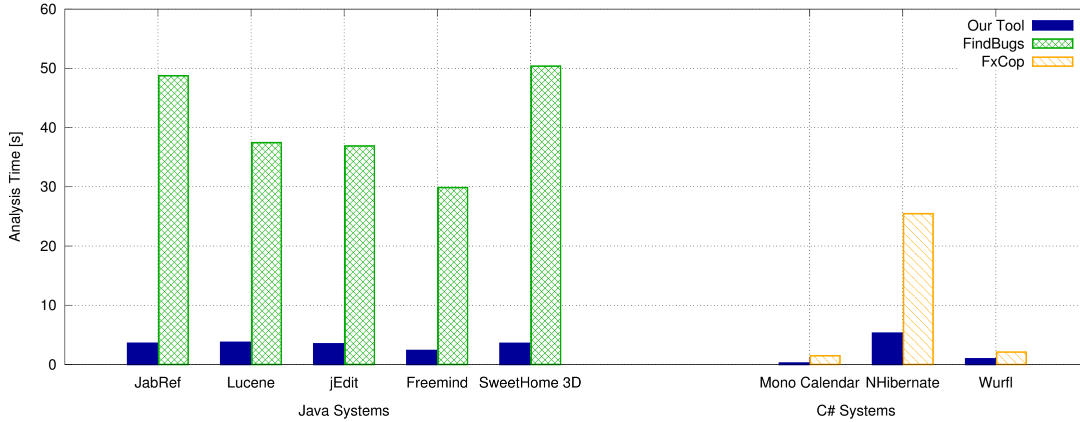


Figure 6.1: Comparison of the time it takes our, FindBugs' and FxCop's data flow analysis to execute on several systems.

	ABAP		C#		Java	
	Total	Sample	Total	Sample	Total	Sample
Dead Store	1058	100	223	71	583	195
Self Assignment	3	3	0	0	0	0
Null Pointer	7	7	210	100	125	41

Table 6.11: The number of findings produced and sampled per language and analysis for our tool. The values for Java and for C#'s dead store analysis were taken from RQ2a.

analysis. Afterwards, we calculated the precision of our analyses based on our manual assessment of these findings.

6.7.2 Results

Using the same method as in RQ2a, we calculated the precision of the ABAP and C# analyses. Table 6.12 shows the resulting numbers together with the results from RQ2a. Once again, the self assignment analysis did not produce enough findings to allow us to calculate a sensible precision. All three findings we sampled for ABAP were true positives.

When we compare these values, we can see that the dead store analysis performs equally well on all languages with a precision value of 95% and higher. The null pointer analysis also performs comparably well on C# and Java with a precision around 30 to 40%. It has a much higher precision on ABAP systems, but due to the low number of findings produced on these systems we do not have a high confidence in this value. Given a larger sample of ABAP code that yields more findings, we might obtain a lower precision. We can thus conclude that our approach produces results of roughly equal precision on

	ABAP			C#			Java		
	TP	FP	Precision	TP	FP	Precision	TP	FP	Precision
Dead Stores	99	1	99%	70	4	95%	187	8	96%
Self Assignment	3	0	–	0	0	–	0	0	–
Null Pointer	5	2	71%	43	57	43%	14	27	34%

Table 6.12: The number of true (TP) and false positives (FP) as well as the precision per analysis and language for our tool. The results for Java and for C#'s dead store analysis were taken from RQ2a.

different languages.

6.8 RQ3b: Execution time on different languages

In this research question we compare the execution time of our approach on Java, ABAP and C# systems.

6.8.1 Study Design

We executed our tool on all study systems three times and timed each execution. The resulting running times were aggregated, again using the minimum function, and normalised on the number of SLOC of the analysed system so we could compare them. We furthermore aggregated all measurements per language to obtain an average running time for each language. These values will give the running time that can be expected when the analyses are run on a system written that language. This study was executed on a 64bit notebook with an Intel® Core^{TM} i5-3317U CPU under Ubuntu Linux 12.10.

6.8.2 Results

Figure 6.2 shows how our tool performed on Java, C# and ABAP systems. The mean values per language are 0.0476 $\frac{s}{1000\ SLOC}$ for Java, 0.0492 $\frac{s}{1000\ SLOC}$ for ABAP and 0.0516 $\frac{s}{1000\ SLOC}$ for C#. This indicates that the language specific parts of our tool have roughly the same time complexity.

6.9 Threats to Validity

There are several threats to the validity of the results of our case study.

6.9.1 Internal

First of all, the manual assessment of all reported findings in the case study was done by a single person: the author of this thesis. If the assessor miscategorises a finding, this

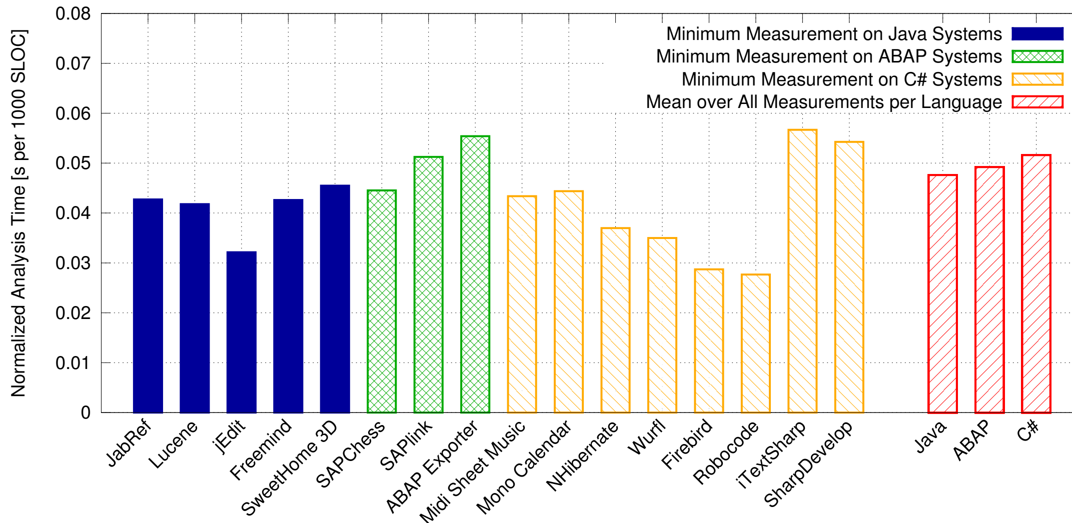


Figure 6.2: The normalised time it takes our analyses to execute on systems in different languages. The graph shows the minimum of 3 measurements per system and the mean value for each language over all measurements.

will change the resulting precision and recall values. To mitigate this threat, each finding was carefully examined. In situations where the assessor was not entirely certain how to categorise a finding, a second, independent opinion was obtained from another person. Both assessors have experience with each of the analysed programming languages.

Secondly, since we cannot know all places in the source code of a program where a data flow problem exists, we cannot give an accurate value for the recall of our analysis. The recall values we report in section 6.5 are therefore only estimates. They are, however, the best estimate we could achieve at the point of writing. At the very least, they can be read in relative terms: our recall is higher than FindBugs' and FxCop's.

Finally, for FindBugs' null pointer analysis on Java systems only a very small number of findings was reported, as it was heavily optimised to reduce false positives. The same happened for our null pointer analysis on ABAP systems, because null pointers are not a commonly used programming idiom in ABAP code. For both, our reported precision values must therefore be taken with care. We do, however, believe that FindBugs' precision value is reliable, as it has been reported as having a high precision in several other publications [12, 13, 20, 24].

6.9.2 External

We took two precautions to maximise the transferability of our results to different systems, languages and domains.

First of all, we chose the three languages for which we implemented a back-end with care: While Java and C# are very similar languages, both syntactically and semantically,

ABAP differs highly from the two due to its extensive use of keywords and integration of domain-specific aspects from the SAP environment. This choice of languages mitigates the threat that the case study results are not transferable to other programming languages.

Furthermore, we also chose the systems on which we performed the analyses in the evaluation with care. We provided a mix of different sizes, ranging from several thousands to several hundreds of thousands of SLOC, and domains, ranging from text editing software to database back-end libraries. This mitigates the threat that our results are not transferable to other systems.

6.10 Discussion

This case study demonstrated several things:

- Our language model can be applied to very different languages (Java and C# vs. ABAP) with comparable results.
- There is a high amount of code reuse between similar languages (Java and C#) as well as vastly different ones (Java/C# and ABAP), which makes it easy to add more languages to the system. With an implementation effort between 400 and 2400 SLOC for a new back-end, integrating a new language should be easy and fast.
- Our shallow parser and heuristic-based approach performs significantly faster than a language specific tool that constructs a full AST to understand the entire source or bytecode.
- Our language model provides enough information to allow simple data flow analyses to operate with very high precision ($> 90\%$) and complicated ones with medium precision ($> 30\%$).
- When compared to a state of the art tool, our approach exhibits a higher recall for all analyses and a higher precision for simple ones. For complicated ones, our approach performs only slightly worse according to the $F_{0.5}$ score.

Due to the fact that both the dead store and the null pointer analysis have a very high recall ($> 80\%$ for Java systems and $> 50\%$ for C# systems), we believe that their precision can still be increased by implementing more false positive filters, thus mitigating the shortcomings of our approach to some extent.

One area where our most complicated analysis – the null pointer analysis – could benefit greatly from such filters is the detection of dependencies between conditions. In the case study, such dependencies were the main cause of false positives across all systems and programming languages. Being able to detect them and filtering the respective findings could potentially increase the precision of the null pointer analysis well beyond 50% at the expense of a smaller recall.

6.10.1 Limitations of the Framework

There are several apparent limitations to our framework that came to light during the evaluation. First of all, exception handling is not accurately represented in the CFG we construct. This lead to several false positives in the case study, as we do not know which statements might throw an exception and where exactly it would be caught. On the other hand, this simplified CFG is easier to construct and thus a new language back-end can be created much faster.

Furthermore, our current condition parsing algorithm is limited in that it only recognises simple conditions. Paired with the fact that many paths through a program are infeasible as several conditions depend on each other, this produces many more false positives in path sensitive analyses.

Finally, we have restricted ourselves to intra-procedural data flow analysis that completely ignores the call sites of the analysed method, any information about the possible return values of invoked methods, and any information from instance and static fields of a class. This also leads to an increase in false positives and probably also in false negatives. On the other hand, this greatly decreases analysis time and complexity and allows all of our analyses to be run incrementally.

7 Conclusion

In this thesis we presented a language model that captures the essential information necessary to perform data flow analyses on the source code of programs. Based on this model we created a framework that can perform a variety of such analyses, independent of the programming language in which the analysed source code is written. We furthermore implemented several different data flow analyses based on our framework. Both the analyses and the framework itself were evaluated in a case study.

The study showed that our approach produces useful results (in terms of analysis precision and recall) within a reasonable amount of time. These results can be replicated across projects written in several different programming languages. Moreover, we demonstrated that our approach can compete with established, language dependant data flow analysis tools, even though it has less accurate information about the source code.

Finally, we were able to show that new programming languages can be integrated into the existing framework with little development effort, ranging between 400 and 2400 SLOC.

7.1 Future Work

There are several interesting research opportunities that arise from this work. First of all, it is possible to add more and different languages to our approach and to evaluate them using another case study. Especially dynamic languages like JavaScript or Ruby, which are semi-functional and do not feature static typing, could be interesting and challenging research subjects.

This leads to another area where our work can be improved upon: Extending our language model to support a wider variety of languages. For example, functional languages are currently not at all supported by our model, since a CFG cannot easily be constructed for them.

Moreover, improving the path-sensitive analyses by finding dependencies between conditions would be helpful. As our case study has shown, this is the most common reason for false positives in these analyses. Such improvements could range from simple false positive filters that can detect syntactically equal conditions or sub-expressions therein to an inference- or boolean-logic-based approach that uses an SMT solver to filter infeasible paths through the program as shown by Junker et al. in [14]. The latter approach would also allow us to greatly simplify the existing null pointer analysis.

It would also be interesting to test the influence of false positive filters on the precision and recall of our analysis. This could help us to better understand the trade-off between the two and allow us to optimise the usefulness of the findings our analysis generates.

Finally, there are several interesting analyses that could be implemented based on our framework. Some of them require an extension of the language model to make additional information about the source code available to the analyses.

Constant propagation Finding variables that always contain a constant value can help identify values that should be extracted to constants for reuse purposes. Furthermore, this information may be used by other analyses to increase their accuracy.

Branch feasibility Detecting infeasible branches in a program has two advantages: First, these are often indicative of bugs or mistakes and second, this information can be used to speed up other analyses. These can simply ignore such branches as they can never be entered at runtime.

Type analysis It would be interesting to use type information to find bugs such as useless type checks or impossible casts.

Inter-procedural analysis Finally, it would be interesting to extend our current analyses across method boundaries. Having information about the return values of function calls and the parameters with which a method may be invoked enables a whole new set of analyses and can greatly improve the accuracy of existing ones.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] F.E. Allen. Control flow analysis. *Proceedings of a symposium on Compiler optimization*, pages 1–19, 1970.

[3] Andrew W. Appel. SSA is Functional Programming. *SIGPLAN Notices*, 33(4):17–20, 1998.

[4] John Aycock and R. Nigel Horspool. Simple Generation of Static Single-Assignment Form. In David A. Watt, editor, *CC*, volume 1781 of *Lecture Notes in Computer Science*, pages 110–124. Springer, 2000.

[5] James R. Cordy, Thomas R. Dean, and Nikita Synytskyy. Practical language-independent detection of near-miss clones. In *Proc. Conf. Centre for Advanced Studies on Collaborative research (CASCON)*, pages 1–12. IBM Press, 2004.

[6] Ron Cytron and Jeanne Ferrante. Efficiently Computing Phi-Nodes On-The-Fly. *ACM Trans. Program. Lang. Syst.*, 17(3):487–506, 1995.

[7] Florian Deissenboeck, Elmar Jürgens, Benjamin Hummel, Stefan Wagner, Benedikt Mas y Parareda, and Markus Pizka. Tool Support for Continuous Quality Control. *IEEE Software*, 25(5):60–67, 2008.

[8] Serge Demeyer, Sander Tichelaar, and Patrick Steyaert. FAMIX 2.0 - The FAMOOS Information Exchange Model. Technical report, University of Berne, 1999.

[9] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A Language Independent Approach for Detecting Duplicated Code. In Hongji Yang and Lee White, editors, *Proceedings of the International Conference on Software Maintenance*, pages 109–118. IEEE Press, September 1999.

[10] Lloyd D. Fosdick and Leon J. Osterweil. Data Flow Analysis in Software Reliability. *ACM Comput. Surv.*, 8(3):305–330, 1976.

[11] Lars Heinemann, Benjamin Hummel, and Daniela Steidl. Teamscale: Software Quality Control in Real-Time. In *Proceedings of the 36th ACM/IEEE International Conference on Software Engineering (ICSE'14)*, 2014. To appear.

[12] David Hovemeyer and Wiliam Pugh. Finding Bugs is Easy. *ACM SIGPLAN Notices*, 39(12):92–106, 2004.

[13] David Hovemeyer and William Pugh. Finding more null pointer bugs, but not too many. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '07, pages 9–14, New York, NY, USA, 2007. ACM.

[14] Maximilian Junker, Ralf Huuck, Ansgar Fehnker, and Alexander Knapp. SMT-Based False Positive Elimination in Static Program Analysis. In Toshiaki Aoki and Kenji Taguchi, editors, *ICFEM*, volume 7635 of *Lecture Notes in Computer Science*, pages 316–331. Springer, 2012.

[15] Thomas Kinnen. Supporting Reuse in Evolving Code Bases using Code Search. Master's thesis, Technische Universität München, Munich, Germany, October 2013.

[16] Michele Lanza, Stéphane Ducasse, et al. Beyond language independent object-oriented metrics: Model independent metrics. In *Proceedings of 6th International Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, 2002.

[17] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 75, Washington, DC, USA, 2004. IEEE Computer Society.

[18] Tom Mens and Michele Lanza. A Graph-Based Metamodel for Object-Oriented Software Metrics. *Electronic Notes in Theoretical Computer Science*, 72(2), 2002.

[19] Jan Midtgaard. Control-flow analysis of functional programs. *ACM Comput. Surv.*, 44(3):10, 2012.

[20] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. A Comparison of Bug Finding Tools for Java. In *Proc. of the Int'l Symp. on Software reliability Engineering*, pages 245–256. IEEE, 2004.

[21] Y. N. Srikant and Priti Shankar, editors. *The Compiler Design Handbook: Optimizations and Machine Code Generation, Second Edition.* CRC Press, 2007.

[22] C. J. van Rijsbergen. *Information Retrieval.* Butterworth-Heinemann, London, 2nd edition, 1979.

[23] Eugen-Nicolae Volanschi. A portable compiler-integrated approach to permanent checking. *Autom. Softw. Eng.*, 15(1):3–33, 2008.

[24] Stefan Wagner, Jan Jürjens, Claudia Koller, and Peter Trischberger. Comparing Bug Finding Tools with Reviews and Tests. In *Proc. of Int' Conf. on Testing of Communications Systems*, pages 40–55, 2005.