CheckMerge: A System for Risk Assessment of Code Merges

Jan-Jelle Kester

Master Thesis
Master of Computer Science
Software Technology specialization

University of Twente
Faculty of Electrical Engineering, Mathematics and Computer Science
Formal Methods and Tools research group

April 26, 2018

Supervisors
prof.dr. M. Huisman
dr. A. Fehnker
ir. R. van Paasen (ALTEN Netherlands)

Abstract

When working on large software projects using version control systems, merges are not always trivial to execute. Even when a merge can be resolved without manual intervention, the resulting program is not necessarily correct. In this study a number of categories of changes that may cause issues during merges are identified. This report introduces two new language-independent algorithms that detect changes from three of these categories. These algorithms work based on the abstract syntax trees (ASTs) of compared program versions and require the differences between these versions to be calculated beforehand. A prototype system has been designed and implemented for the C programming language. The newly developed algorithms perform well in detecting the problematic changes, in the case of one algorithm at the cost of false positives. The prototype system shows the feasibility of such a system, but is not yet suitable for production use. All in all the analysis of source code merges is a promising area of research and with some effort a tool for practical code merge analysis could be produced, helping developers be more productive when carrying out merges with less errors.

Acknowledgements

I would like to thank ALTEN Netherlands, and especially Rob Essink, for providing this project. I have found it an interesting challenge to work on. As a developer I have faced merge problems myself many times, albeit on a relatively small scale. It is satisfying to work on a solution for these problems.

Furthermore I would like to thank my supervisors for the excellent feedback and, at times, tough questions. On many occasions this has forced me to think a bit more about certain problems and their possible solutions, resulting in either a better choice or a better understanding of the reason for a certain choice.

I also would like to thank Robin Hoogervorst for letting me use code we developed together for an assignment of the Software Security course. This code was turned into a test case for the system.

Finally I would like to thank everyone else who has listened to any problems I have had and everyone who has helped to steer me in some direction, even though this sometimes led me on interesting detours. Either way, your comments have helped progress a lot. This includes friends, colleagues at ALTEN Netherlands and FMT staff.

Table of contents

Chapter 1

Introduction

For many software developers merging changes in a version control system is a common task. However, this task is error-prone due to the fact that the merging algorithms commonly used by version control systems do not take the semantics and structure of a programming language into account [19]. Merging is especially risky when versions have diverged significantly, either over time or by very involved changes like refactorings. Some combination of changes in two versions may cause the result of a merge to be different from the expected or wanted result, either because of incorrect computations or syntactical or structural errors [2].

To aid developers with the task of merging software versions ALTEN Netherlands (in this report also referred to as 'the client'), a technical consulting firm, has proposed to develop a software tool for analyzing the risk of code merges by identifying changes that can lead to unwanted results after merging.

1.1 Motivation

Code merges are a common task in large software projects with many contributors. Some of these merges are considered trivial and do not require review. When changes are more involved there is a chance that the code resulting from the merge will not work or will not behave as expected. Merges can accidentally undo earlier fixes or improvements and in some cases introduce new bugs which were not present in any of the versions of the code which were merged together. Manually reviewing code merges takes a lot of time, while possible errors might still not be identified by the reviewer(s).

To overcome this, a software tool is proposed which will analyze code merges and present a risk assessment to the user. This tool should be able to express the risk involved with a particular merge and be able to identify specific parts of the code which are likely to fail after the merge.

```
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    goto fail;
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;
```

Listing 1.1: Code fragment illustrating the 'goto fail' vulnerability

The problems with code merges as described above were noticed in some C/C++ projects at a large technology firm where software engineering consultants of ALTEN Netherlands work on embedded software. These problems especially arise when two versions of a program have been developed in parallel for a while. When these versions are merged many duplicate or conflicting improvements may exist. When the implementation of duplicate improvements is not exactly identical, or if improvements in different versions are not compatible with each other, merging them can be troublesome. Industry standard merge tooling, like those found in common version control systems, does not detect these problems making manual inspection of changes necessary.

A well-known example of a (possible) merge error is the 'goto fail' vulnerability in the Apple TLS implementation [33]. In the function containing the bug, error codes were checked with bracketless if statements, so only the line immediately following the if statement is skipped if the condition does not hold. In this example the conditions check for error codes and if one was present the statement `goto fail;` was executed. Because at some point in the code this statement appeared twice, the second occurrence was always executed, resulting in the function returning no error code while not all checks were performed. It is likely that this bug was introduced by a merge gone wrong, although Apple has not released these specifics. Nevertheless it is a real-world security issue which, if caused by a merge error, might have been prevented by a merge analysis tool, as noted by Wheeler [33]. The offending code is shown in listing 1.1.

1.2 Goals

The focus of this project is to find out whether it is feasible to implement a system to check for code merge problems as illustrated in the previous section. This leads to the following goal:

> *To design and implement a prototype of a system for assessing the risk of side effects of code merges.*

To reach this goal a number of research questions have been formulated. These questions are addressed in the upcoming chapters and the outcomes were used in the development of the prototype. The research questions are as follows:

Q1 Which kind of changes are likely to cause problems in code merges?

Q2 Which algorithms are best suited to compare code versions to find changes?

Q3 Which techniques exist to detect these merge problems from the changes between versions?

1.3 Approach

In order to gain a better understanding of the problem domain, existing literature has been studied on the subjects of source code merging, source code analysis and tree differencing. Together with the client a number of requirements was agreed upon, which were taken into account during the rest of the project. A number of categories of relevant changes were specified based on interviews with developers, experience of the client and the personal experience of the author. Algorithms for the detection of some of the categories have been developed.

In order to make algorithm development easier, a supporting prototype system was developed. First, different tools for parsing C code were evaluated of which one was chosen (see chapter 4 for evaluation criteria and methods). Subsequently a high-level system architecture was designed and a prototype of this system was implemented. A tree differencing algorithm for comparing abstract syntax trees and finding the changes between them was chosen (see section 6.2 for details). Based on the chosen C parser and tree differencing algorithm a simple internal representation of an AST was created and a transformation from the output of the C parser and the internal representation was developed. This allowed the tree differencing algorithm to be implemented and tested.

With the data input handling and tree differencing in place the first ideas for the algorithms were implemented. The algorithms were partially tested

with unit tests and partially with manually evaluated test cases that use the whole system. The results of these tests were used to improve the algorithms. For the final evaluation the precision and recall of the algorithms was calculated to determine the qualitative performance. A number of performance metrics were collected to determine the runtime performance.

1.4 Structure of the report

This report is structured as follows. Chapter 2 contains background information on the problem domain. In chapter 3 changes that are of interest when detecting merge problems are defined in categories, and newly developed algorithms for detecting some of these problems are presented. Chapter 4 lists a number of C code parsers and analysis tools that can provide the necessary information for the algorithms. A high level architecture of the system is given in chapter 5 and implementation details of the prototype can be found in chapter 6. Chapter 7 shows the evaluation strategy and results for the new algorithms. Other works related to this research are discussed in chapter 8. Finally, the conclusions and recommendations for future work are given in chapter 9.

1.5 Contributions

The contributions presented in this report are threefold.

Definition of problematic changes First of all, this report defines the term *problematic change* in the context of code merges. A number of categories of these problematic changes are listed. This is done with the goal of developing algorithms for identifying these changes and reporting these problems to the user that wants to merge two versions of a program.

Detection algorithms Secondly, two new algorithms for the detection of a subset of these changes have been created. These algorithms are able to detect certain problems in code merges that were previously not detected by existing merge tooling.

Analysis system prototype Finally, a system design and a prototype implementation of this design, incorporating the aforementioned algorithms, has been created for the C programming language. This prototype is used to evaluate the algorithms. These contributions show that risk analysis on code merges is worth looking into further as practical applications are feasible and likely to improve the quality of code merges while reducing the work load of developers carrying out these merges.

The source code of the prototype, including test cases, has been published on GitHub. The code is split up over two repositories, one for the analysis system written in Python and one for the custom LLVM analysis pass to support the C parser. The base analysis system is located at `https://github.com/jjkester/checkmerge`. The LLVM analysis pass repository is located at `https://github.com/jjkester/checkmerge-llvm`.

Chapter 2

Background

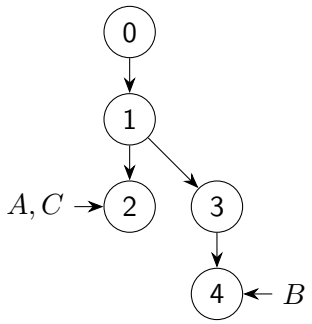
2.1 Version control systems

During the development of software the program evolves rather than being constructed correctly in a single go. Requirements and library dependencies change, new insights come along and different developers work on the program. Version control systems help in the software development process by allowing multiple versions of a program to be worked on at the same time. These systems also keep a history of the evolution of the software so that it is possible to *revert* to an earlier version of a program at any time.

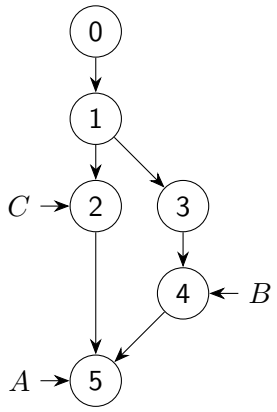
2.1.1 Concepts

The data structure in which the files and their history is stored is called a *repository*. This repository can be stored *centralized* or *decentralized* (also called *distributed*) [24]. When using a centralized version control system developers must check-in individual changes into a central repository on a server. Well-known centralized version control systems are CVS and SVN. With decentralized version control systems each developer has a copy of the repository on their system. He or she synchronizes the repository with copies on different machines. Usually a central repository is chosen to synchronize with, although it is possible to synchronize with each copy individually. Examples of decentralized version control systems are Git and Mercurial.

Changes are saved to the repository as *commits*. A commit describes one or more changes to one or more files in the repository. Many version control systems only save the differences since the previous commit. The commits are linked together to form a consistent history. If a *branch*, which is a separate version, is created a commit might have more than one successor, as shown in figure 2.1a. Branches can be *merged* to bring the changes from one branch to another, as shown in figure 2.1b.



(a) A commit graph showing two branches A and B.



(b) A merge of branch B onto A.

Figure 2.1: Commit graphs showing the merge of diverged branches A and B. A third branch C is not affected.

2.1.2 Merges

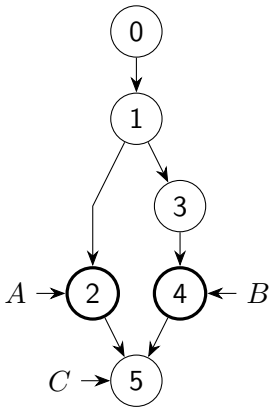
In the context of a merge, a branch can be seen as a collection of changes since t a specific point in time. Given two branches, the changes each branch represents are the changes that were committed since the common ancestor commit of the two branches.

In some cases it is trivial to merge changes to source code. If two changes are merged, and both changes only affect independent parts of a program which are in separate files, the merge can be completed by just combining the changes. If changes affect the same file a new version of the file has to be saved that incorporates both changes, which can be done automatically by many version control systems. However, version control systems and other merge tooling might raise a *merge conflict* if it is not able to compute the result with confidence. An example of this situation is two changes to the same line. The absence of a merge conflict does not imply that the result is as expected, as noted by Aziz, Lee and Prakash in their book about typical problems software engineers run into [2]. This is explored further in chapter 3.

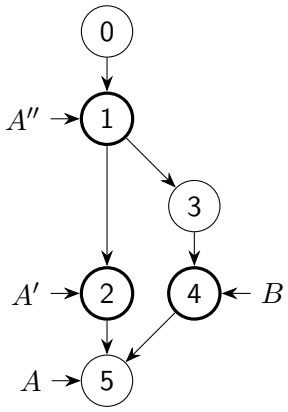
2.1.3 Merge techniques

There are different techniques and algorithms for merging different versions of a program. Mens provides a comprehensive survey of code merging techniques [19]. These techniques can be categorized with respect to a number of properties.

First of all a distinction is made between *two-way merging*, where only two versions of a program are considered, and *three-way merging* where the



(a) A two-way merge of branches A and B into a new branch C.



(b) A three-way merge of branch B onto A, moving the branch pointer to the merge result.

Figure 2.2: A comparison between two-way merges and three-way merges. The inspected versions are denoted by a bold line.

changes in two versions since a common ancestor version are used. With two-way merging all the files in the branches are compared and combined where possible. For large repositories this can become quite resource intensive. Three-way merging helps with merge decisions, especially when some code is removed in one of the versions. It is a performance optimization as well, as only the changes since the common ancestor of the branches are being merged. For example, given the merge in figure 2.1b only the combination of the changes in $\{2, 3, 4\}$ is computed when using three-way merging while for two-way merging also the changes made in $\{0, 1\}$ and all ancestors of 0 that are not shown are taken into account.

Figure 2.2 contains a comparison between two-way merging and three-way merging. Two-way merging takes only the merged versions A and B into account, creating a completely new version C containing the union of the source versions. Three-way merging takes only the changes since a common ancestor version into account, replaying the changes of the merged version B onto the program version A', resulting in A.

Another distinction is made between *textual*, *syntactic*, *semantic* and *structural* merging. Textual merging looks at lines of files and is very common and relatively fast to compute, however, there are many situations in which manual merging is still required as the algorithm can only check for similarities before and after a change. Syntactic, semantic and structural merging take the meaning of the text into account and therefore need to be adapted for specific situations (e.g. specific programming languages) but can yield more precise results.

2.2 The impact of code changes

Changes to source code can introduce bugs immediately or pose problems when merged with other changes. There might be a relation between certain kinds of changes and the risk of introducing problems. In software evolution the impact of changes is analyzed to find relations between kinds of changes and the risk they have to introduce bugs.

The impact of source code changes is a typical software evolution problem. This problem can be approached on a high level where one tries to estimate the amount of time and/or the risk involved with a proposed change, for example a new feature. On a lower level one looks at transformations of source code and the risk of introducing defects. The latter is relevant for this project.

The claim that changes to relatively complex code are more likely to cause issues than changes to relatively simple code seems self-evident. This claim is supported by Munson and Elbaum [20]. In their research they tried to find a metric that could serve as a surrogate for the risk of introducing a fault. They found that a high rate of change in the relative complexity could serve as an indicator for higher risk changes. They do note that there are multiple ways of calculating complexity, which influences the metric. However, the relative complexity of a change is a good indicator of the risk associated to it.

Besides complexity, the size of a change is also a factor. Small code changes are less likely to cause issues than large code changes. Purushotaman and Perry studied small code changes and tried to differentiate between multiple kinds of changes [26]. In their research they categorized changes as *corrective* (repairing faults), *adaptive* (introducing new features) or *perfective* (nonfunctional improvements). They categorized the changes based on keywords in the commit messages. The software of a central (provider level) telephone switch was analyzed and they found that nearly 40% of changes intended to fix a bug introduced a new bug. They also found that less than 4% of one-line changes resulted in a bug, while nearly 50% of changes involving 500 lines of code or more introduced one or more bugs, thereby supporting the claim that small changes are less likely to introduce bugs than large ones.

Bavota et al. looked at refactorings, which typically have a lot of dependent changes. They found that certain refactorings are more likely to cause bugs than others [5]. Refactorings changing the class hierarchy were very likely to cause errors because these changes have impact on many references, and therefore many lines of code. The errors might be due to lack of tool support for this kind of refactorings.

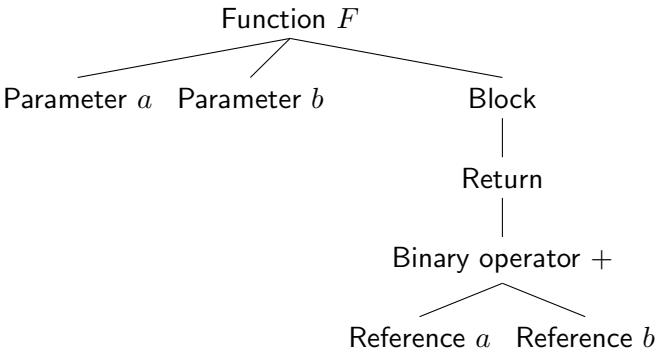


Figure 2.3: Example of an abstract syntax tree of a function performing binary addition.

2.3 Abstract syntax trees and control flow graphs

The source code of computer programs is mostly represented as text. However, there exist other representations for programs, like abstract syntax trees and control flow graphs.

An abstract syntax tree is a tree representation of source code. Each node in the tree represents a construct used in the program. The kinds of nodes that are used depend on the specific programming language, more specifically, it depends on the constructs that a programming language offers. Abstract syntax trees differ from concrete syntax trees in the level of detail they provide. While concrete syntax trees include syntactically relevant characters like parentheses these are omitted in abstract syntax trees. An example of an abstract syntax tree for a very simple program can be found in figure 2.3.

From an abstract syntax tree a control flow graph can be built. A control flow graph shows the possible execution paths in (part of) a program. From this graph information about the relation of parts of the code can be obtained. As a control flow graph is based on statements it can be embedded into an abstract syntax tree. Control flow graphs are typically collapsed by leaving out the nodes that have only one entry and one exit point. When embedded into an abstract syntax tree this is less practical. When evaluating the tree it is more convenient to have control flow information at every node.

2.4 Tree differencing

Tree differencing is the process of finding the differences between two trees. In the context of this project this technique can be used to find subtrees in an abstract syntax tree (AST) that have been changed when compared to

another version. From this information not only regular *merge conflicts* can be found, but these changes can also be further analyzed.

Definition 2.1 (Bille [7])**.** *An* edit script *is a sequence of operations $O_{T \to T'}$ that, when applied, transform a tree T into another tree T'.*

From this definition follows that the difference between two trees can be expressed as an edit script.

Edit scripts can grow large and have no theoretical maximum size, however, algorithms that calculate these edit scripts often try to find the smallest and most logical (from a developer's point of view) number of changes. This is achieved by assigning a cost to every edit operation. The sum of the costs determines the quality of the edit script. The cost of a type of operation is often not fixed and can be used to tweak these algorithms.

Definition 2.2 (Bille [7])**.** *The* tree edit distance *of two trees T and T' is defined as the length of the minimum cost edit script for these trees.*

The tree edit distance is a measure for the similarity of trees. Two trees with a short edit distance have little operations in their edit script and are therefore relatively similar. Trees with a large edit distance have a large edit script and are therefore relatively dissimilar. The cost of each edit operation is often assumed to be 1, however, in some uses cases other constants or cost functions are used. The cost of an edit script is the sum of the cost of the edit operations.

2.5 Source code analysis tools

Besides the specific techniques listed above there are some more generic ways of getting certain information from source code. The code analysis tools listed here are primarily focused on the C programming language (in accordance with the requirements listed in appendix A).

A large number of studies use CIL for C code analysis. CIL a C intermediate language designed for analysis of C programs. It was first described by Necula, McPeak, Rahul and Weimer [22]. CIL is designed to support compiler specific extensions and contains both an abstract syntax tree and a control flow graph. On GitHub[1] the authors describe the tool as follows: "CIL is a front-end for the C programming language that facilitates program analysis and transformation. CIL will parse and typecheck a program, and compile it into a simplified subset of C." CIL is written in OCaml [23] and this is the language that needs to be used to use the tools to analyze the code.

Just like dedicated analysis tools, C compilers have a lot of useful information internally. Clang is a front-end compiler for C and C-like languages

[1]The CIL GitHub repository is located at `https://github.com/cil-project/cil`.

which uses the LLVM back-end [8, 16]. Clang is designed with tool support in mind, meaning that it can provide a lot of information about code to an external program. This makes it suitable for C code analysis. Clang offers a C API that can be used by other programs. It is able to export abstract syntax trees of parsed code which can be used by a number of the described algorithms.

A more generic approach is taken by Rascal. Rascal is a domain specific language for source code analysis and manipulation. It can support many languages, allowing for the reuse of analysis code between different parsers. By default the language supports many programming language concepts, including grammars, data types, parsers and syntax trees. Rascal is being developed at CWI (Centrum Wiskunde & Informatica) [15, 27]. At the moment Rascal only has alpha releases so it might not be stable (enough) for use yet.

Chapter 3

Problematic changes in merges

This chapter describes categories of changes that can cause issues during or after code merges. Following this definition, two new algorithms for detecting some of these categories of changes are developed.

3.1 Problematic changes

A number of kinds of changes can cause issues when combined (for example with a merge) with certain other changes in another version of the software. In the context of this paper these changes will be referred to as *problematic changes*. In order to formally define a problematic change, first a distinction needs to be made between a text-based merge, which is common, and the ideal world situation in which the intentions behind the code are taken into account.

Definition 3.1. *Given the versions of the same (partial) program A and B, the* textual merge *of these programs is the program resulting from combining A and B using a textual merge strategy. A textual merge is represented as* $A \cup^t B$.

Definition 3.2. *Given the versions of the same (partial) program A and B, the (hypothetical)* functional merge *of these programs is the program resulting from combining the functionality encoded in A and B and is represented as* $A \cup^f B$.

As stated before, many merge algorithms in use today perform textual merges. This does not always result in the best possible merge, as a functional merge would. Because functional merging is hard, if not impossible since the intentions of the code need to be known, no algorithms for this have been developed to date. Therefore, the process of merging often includes

manually checking if the result of the automated merge was as intended, and if not, correcting the errors. A *problematic change* is a change that, when merged, does not result in the intended functionality. This can be formalized as follows.

Definition 3.3. *Given a program P and modifications of that program $c_1, \ldots, c_n$, the subsequent version P' resulting from applying these changes is represented as $P' = P \oplus c_1, \ldots, c_n$.*

Definition 3.4. *Given a program P with subsequent versions P_x, C_x is the set of changes such that $P_x \equiv P \oplus C_x$.*

Definition 3.5. *Given a program P with subsequent versions $P_1 = P \oplus C_1$ and $P_2 = P \oplus C_2$, a* problematic change *is a change $c \in C_1$ for which a change $c' \in C_2$ exists such that $P \oplus c \cup^t P \oplus c' \notin P_1 \cup^f P_2$.*

According to this definition a change is only problematic in the context of this report if it is combined with other changes in another version of the program. Whether a change is problematic therefore depends on the context of the merge.

From examples in the literature described in chapter 2 and talks with a number of software developers the following practical problematic changes have been identified:

PC1 Both versions introduce a change at the same point in a program.

PC2 Both versions introduce a change modifying the same value in a scope.

PC3 A version contains a refactoring while the other version added references to the refactored statement(s).

- PC3a An identifier was renamed.
- PC3b An identifier was removed.
- PC3c The type signature of a declared entity changed.
- PC3d A declared entity was split up or merged.

Changes like PC1 are generally not a problem since these are covered by most, if not all, version control systems. Merge algorithms typically are line-based and since these changes occur at the same line (or nearly the same line) a line-based tool will detect a possible problem. A well-known algorithm that does this is the algorithm powering Diff [14]. These detected problems are referred to as *merge conflicts* [2, 19]. Merge conflicts will prevent merging until they are manually resolved.

An example of PC2 is shown in listing 3.1. This example shows two versions of the `calc` function fixing the same bug (the result is 1 too high) that was present in the original version. Listing 3.1d shows the result of merging

```c
int calc(int a, int b) {
    int c = a + b;
    printf("c=%d\n", c);
    return c;
}
```

(a) Original version of the program.

```c
int calc(int a, int b) {
    int c = a + b;
    printf("c=%d\n", c);
    return c - 1;
}
```

(b) Branch A of the program with a bug fixed by changing the return value.

```c
int calc(int a, int b) {
    int c = a + b - 1;
    printf("c=%d\n", c);
    return c;
}
```

(c) Branch B of the program with a bug fixed by changing the intermediate variable *c*.

```c
int calc(int a, int b) {
    int c = a + b - 1;
    printf("c=%d\n", c);
    return c - 1;
}
```

(d) Result of naively merging A and B, which consistently returns a value that is 1 lower than expected.

Listing 3.1: Trivial example of a merge of two correct versions of a program resulting in an incorrect program (PC2).

the two branches with a textual merge algorithm. Only when inspecting the result it becomes clear that it is not as intended as the result is now 1 too low because of the 'double' fix.

The example given in section 1.1, the 'goto fail' vulnerability in the Apple TLS implementation, might be an example of PC2. While with this example it is not publicly known whether this was caused by an incorrect merge, it might be possible. If it were caused by an incorrect merge, an implementation checking for PC2 should be able to detect it.

Another example, shown in listing 3.2, illustrates problem PC3a. In this example two arguments are renamed in one version ($a \to x$ and $b \to y$), while the other version introduces a new occurrence of the variable a. Both versions work as expected (for that version). The result of a textual or line-based merge in listing 3.2d shows the result, which is not a valid program due to the broken reference to variable a.

According to the literature discussed in chapter 2 there is great risk in refactorings (PC3) [5, 31], while small changes are generally less risky [26]. However, many bugfixes are small changes. As shown in the examples in listings 3.1 and 3.2 bugs do not necessarily have only one way to fix them. Therefore it is not feasible to rule out changes based on their (relative) size.

```c
int calc(int a, int b) {
    int c = a + b;
    printf("c=%d\n", c);
    return c;
}
```

(a) Original version of the program.

```c
int calc(int a, int b) {
    int c = a + b;
    printf("c=%d\n", c);
    return c + a;
}
```

(b) Branch A of the program, with an algorithmic change.

```c
int calc(int x, int y) {
    int c = x + y;
    printf("c=%d\n", c);
    return c;
}
```

(c) Branch B of the program, with a refactoring (renamed $a \rightarrow x$ and $b \rightarrow y$).

```c
int calc(int x, int y) {
    int c = x + y;
    printf("c=%d\n", c);
    return c + a;
}
```

(d) Result of merging A and B with a line-based algorithm, which contains a reference to an undefined variable.

Listing 3.2: Trivial example of a merge where a renamed parameter in one version causes a broken reference after merging (PC3a).

3.2 Detection strategies

For each problematic change at least one strategy for detecting the possible problem is discussed below.

3.2.1 Changes at the same point in a program (PC1)

These kinds of changes are already detected by existing merge algorithms as these algorithms are unable to merge these kinds of changes. Therefore a detection algorithm for these kinds of changes is not included in the system. Some of these changes might be picked up by other detection algorithms if the change also satisfies the criteria of another group of changes.

3.2.2 Changes modifying the same value in a scope (PC2)

A change might conflict with another change if both changes affect the same value. Given a single program, *dependence analysis* produces the set of statements that may directly affect the result of a statement. This information can be used to find the dependencies of a changed instruction and see if a change in one version of a program may affect a change in another version of the same program.

Dependence analysis

In dependence analysis a distinction is made between several kinds of dependencies. First of all there are *control dependencies*, which encode that the execution of a specific instruction is conditionally guarded by another instruction. Secondly there are *data dependencies* or *memory dependencies* which encode dependencies between instructions that read or write the same memory.

Definition 3.6 (Banerjee [4])**.** *A statement S_2 has a* memory dependency *on a statement S_1 if a memory location M exists such that:*

1. *Both S_1 and S_2 read or write M;*
2. *S_1 is executed before S_2 in the sequential execution of the program;*
3. *In the sequential execution M is not written between the executions of S_1 and S_2.*

For detecting changes like PC2 memory dependencies are very useful. When a changed statement in one version of the program has a dependency on a statement that is changed in a second version of the same program the merge result might not be as expected. The control dependencies are less useful as changing the condition of a conditionally evaluated block does usually not change the intention of that block.

Given the fact that a statement that accesses a memory location is either a read or a write, four distinct kinds of memory dependence can be distinguished.

Definition 3.7 (Banerjee [4])**.** *Given statements S_1, S_2 where S_2 has a memory dependency on a statement S_1 with memory location M.*

1. *S_2 is* flow dependent *on S_1 if S_1 writes M and S_2 reads M (read after write);*
2. *S_2 is* anti-dependent *on S_1 if S_1 reads M and S_2 writes M (write after read);*
3. *S_2 is* output dependent *on S_1 if S_1 and S_2 both write M (write after write);*
4. *S_2 is* input dependent *on S_1 if S_1 and S_2 both read M (read after read).*

Conflict detection algorithm

A new, naive algorithm has been designed to detect changes based on memory dependencies. This algorithm (algorithm 3.1) takes two abstract syntax trees as input and returns a set containing sets of nodes that form a conflict.

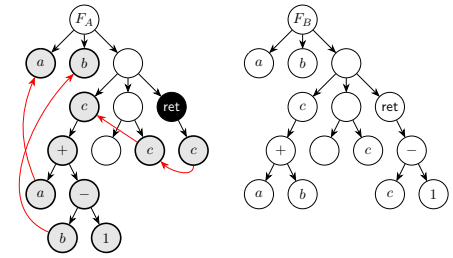
The algorithm uses a number of functions that are defined as follows. The functions *deps*() and *rdeps*() return the recursive memory dependencies and recursive reverse memory dependencies respectively. Descendants

```
function MemDepConflicts(T1, T2)
    R := {∅}
    N := {n | n ∈ T1 ∪ T2 ∧ |deps(n) ∪ rdeps(n)| > 0}
    for n ∈ N do
        M := {mapping(d) | d ∈ deps(n) ∪ rdeps(n)}
        A := {d | d ∈ deps(m) ∪ rdeps(m) ∧ m ∈ M}
        C := {c | (c ∈ M ∪ A ∨ c = n) ∧ changed(c)}
        if (∃c1, c2 ∈ C | c1 ∈ T1 ∧ c2 ∈ T2) then
            R := R ∪ {C}
        end if
    end for
    return OptimizeNodeSets(R)
end function
```

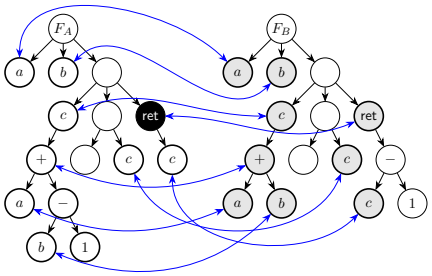
Algorithm 3.1: Memory dependence conflict detection.

```
function OptimizeNodeSets(I)
    R := ∅
    M := Map {n → n | n ∈ I}
    for (S1, S2) ∈ I × I do                          ▷ (S1, S2) ≡ (S2, S1)
        for (n1, n2) ∈ S1 × S2 do
            if n1 ∈ descendants(n2) then
                put(M, n1 → n2)
            else if n2 ∈ descendants(n1) then
                put(M, n2 → n1)
            end if
        end for
    end for
    I := {{get(M, n) | n ∈ S} | S ∈ I}
    for n ∈ I do
        if {i | i ∈ I ∧ n ⊂ i} = ∅ then
            R := R ∪ {n}
        end if
    end for
    return R
end function
```

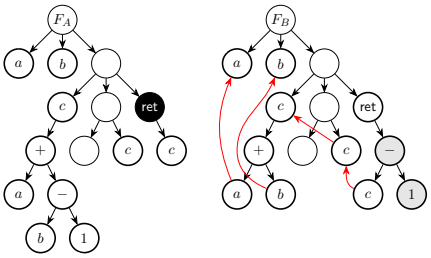
Algorithm 3.2: Merge algorithm for overlapping sets of AST nodes.



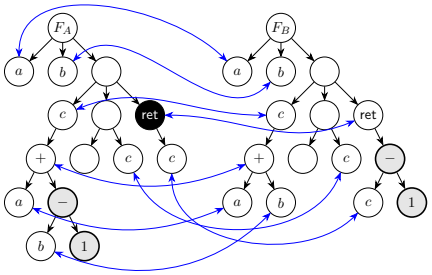
(a) First step of algorithm 3.1. The black node is the inspected node n, the red arrows are dependencies between nodes. Thick-edged nodes are currently inspected. The shaded nodes are in the dependence graph of n.



(b) Second step of algorithm 3.1. The blue arrows are mappings between nodes. The shaded nodes are the mapped counterparts of the previously selected nodes, stored in M.



(c) Third step of algorithm 3.1. The shaded nodes are in the dependence graph of the added nodes of the previous step, stored in A.



(d) Fourth step of algorithm 3.1. All unchanged nodes are removed to reveal a conflict, consisting of the shaded nodes. These nodes are stored in C.

Figure 3.1: Illustrations of the steps taken by algorithm 3.1.

of memory operations (nodes with memory dependencies) are considered as well as it is assumed that these descendants influence their parent. A memory operation with children is typical for value assignments. Therefore the set $deps(n) \cup rdeps(n)$ contains all nodes influencing and influenced by a node n. The function $descendants(n)$ returns the nodes in the tree below the given node. The function $mapping(n)$ returns the counterpart of the given node in the other version, if any exists, and the function $changed(n)$ returns whether the given node is changed.

The algorithm works by iterating over all nodes with memory dependencies either to or from it, which are stored in N. This is to ensure that all possible memory dependence paths are inspected. It then finds all nodes in the dependence graph of the inspected node n, shown in figure 3.1a. For these nodes, if a counterpart exists in the other version, these counterparts

are selected and stored in M (figure 3.1b). For each of the selected counterparts the dependence graph is built again and the resulting nodes are stored in A (figure 3.1c). The nodes in M and A are the nodes that are possibly affected by a change of the inspected node n. The conflicting nodes C are the changed nodes from the set of affected nodes $M \cup A$ and the inspected node n. This result is shown in figure 3.1d. If the set of changed nodes contains at least one element in each tree, the set is added to the result set.

The results are compressed by merging sets of nodes that overlap together. For this purpose algorithm 3.2 has been developed. This algorithm first replaces nodes that are a descendant of another node in the set with the ancestor. This can be done since the ancestor node, given it is a memory operation, covers its descendants. Secondly any set of nodes that is a subset of a larger set is removed to avoid duplicates.

3.2.3 Refactorings (PC3)

Different kinds of refactorings exist. Earlier in this chapter a distinction was made between renamed identifiers, deleted identifiers, changed types of declared identifiers, and split or merged entities (like functions or classes in an object-oriented language). Below these kinds are discussed in more detail, and a new algorithm for detecting renamed and deleted identifiers is given.

Renamed and deleted identifiers (PC3a, PC3b)

Renamed and deleted identifiers can be detected with the same strategy. A new, naive algorithm for this detection is shown in algorithm 3.3.

First, the algorithm iterates over the declarations in the common ancestor D. For each declaration d_0, the mapped nodes d_1 and d_2 in both other versions T_1 and T_2 are looked up. The next part of the algorithm is executed for each version separately.

If a declaration in a version of a program is changed from its counterpart in the common ancestor, there may exist some conflict. The nodes that cause the conflict are the newly added uses of the refactored declaration in the other version. These are determined by looking at the nodes referencing the refactored node in the version without the refactored declaration and discarding the nodes with a mapping.

This process is visualized in figure 3.2.

Changed types of identifiers (PC3c)

Type information is at the moment not present in the intermediate representation, besides from the labels of declarations. Also, given the way C works, it is hard to correctly determine type (in)compatibility in static analysis for some statements. An example of such a statement is accessing

function ReferenceConflicts(T_0, T_1, T_2)
  $R := \varnothing$
  $D := \{d \mid d \in subtree(T_0) \wedge is\_declaration(d)\}$
  for $d_0 \in D$ **do**
    $d_1 := mapping(d_0, T_1)$
    $d_2 := mapping(d_0, T_2)$
    $U_0 := \{u \mid u \in subtree(T_0) \wedge reference(u) = d_0\}$
    if $changed(d_1) \wedge d_2 \neq \varnothing$ **then**
      $U_2 := \{u \mid u \in subtree(T_2) \wedge reference(u) = d_2\}$
      $R := R \cup \{u \mid u \in U_2 \wedge \neg\exists mapping(u, T_0)\}$
    end if
    if $changed(d_2) \wedge d_1 \neq \varnothing$ **then**
      $U_1 := \{u \mid u \in subtree(T_1) \wedge reference(u) = d_1\}$
      $R := R \cup \{u \mid u \in U_1 \wedge \neg\exists mapping(u, T_0)\}$
    end if
  end for
  return R
end function

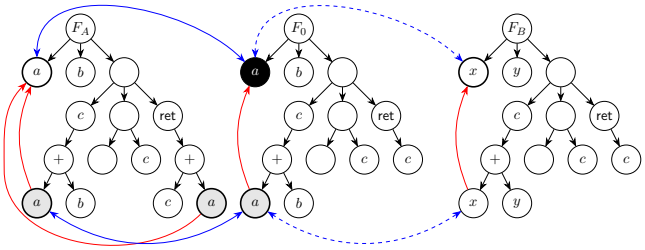
Algorithm 3.3: Detection algorithm for broken identifiers after merging a refactoring.

data in memory through a pointer. Due to the complexity this problem is not addressed in this research.

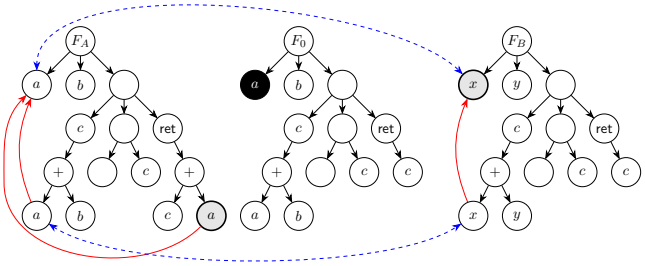
Split or merged entities (PC3d)

Declared entities containing program logic (like classes or functions) can be split up into two or more different entities, requiring two calls instead of one, and two functions can be merged into one. This means that the statements referencing these entities need to be changed as well. Godfrey and Zou describe a method for identifying the splitting and merging of entities [12]. Their method requires user interaction as fully automatic detection is not precise enough. The tool they developed shows the user a list of possibilities, of which the user can choose one. The fact that this technique requires user input makes this method not suitable for this tool.

Some of the problems concerning split and merged entities might be detected by the algorithm for renamed and deleted identifiers. As the original function will no longer exist, the change is picked up as a removed function. Any existing references should be changed to the new function(s) as part of the merge, any new references to the original function will be marked as a conflict. This does not hold when the original function that is split or merged still exists in the code base. While this might not be problematic right away if the implementation of the new function(s) is equal to the orignal one(s),



(a) Process of algorithm 3.3. The black node is the inspected node d_0. The red arrows are declaration references, the blue arrows mappings between nodes from the common ancestor (middle) to the compared versions. The dashed blue arrows are rename mappings. The shaded nodes are the uses that are inspected due to the changed mapping. These are (from left to right) stored in U_1 and U_0 respectively.



(b) Result of algorithm 3.3. The shaded nodes form a detected conflict consisting of a changed declaration in one version and new uses in the other version.

Figure 3.2: Illustrations of the steps taken by algorithm 3.3.

future changes to the implementation might cause strange behavior if part of the code is still using the old functions as the new functionality will not be used in all places it is expected to.

Chapter 4

Code analysis tools

In order to find problematic changes the code that will be merged needs to be analyzed. For every category of changes described in chapter 3, except for item PC1, not only the textual representation of the code, but also its meaning needs to be looked at. A good representation of source code that is often used for code analysis is an abstract syntax tree (AST). Additionally, many algorithms discussed in chapter 2 depend on the AST. The prototype will support the C programming language as this is a specific requirement as listed in appendix A.

In this chapter a number tools are be discussed. The tools under consideration are all able to parse C code and produce an AST. Some of the tools have additional functionality, which is discussed in the next section. The tools have been evaluated on certain criteria to assess their usefulness in the context of this project. One tool was chosen that will be used as a basis for the rest of the project.

4.1 Tools under consideration

For the system under development three tools were considered:

- C Intermediate Language (CIL) [22]
- Clang [8] + LLVM [29]
- Rascal [27]

CIL is used in a number of studies discussed in chapter 2. It is able to compile C programs into an intermediate language which is close to plain C, while keeping references to the original code. This allows for easy analysis since some features of the C language and especially GCC extensions do not have to be taken into account when analyzing the code. CIL can optionally add control flow information to the nodes in the AST. CIL is written in

OCaml, and provides libraries for that language. It also comes with a script that allows it to function as a drop-in replacement for GCC that applies transformations before passing the code on to GCC.

Clang is a front end compiler for the LLVM compiler framework. Besides C it supports a number of C-like languages, including C++ and Objective C. Like CIL, it supports GCC extensions and should therefore be able to compile most C programs. Clang and LLVM were developed with tooling in mind and therefore a number of different APIs for tooling are available. Clang and LLVM libraries are only available for C, however, these ship with Python bindings. Additionally some support for dynamically extending the compiler is available.

Rascal is a metaprogramming environment designed for analyzing, transforming and generating source code. It is designed to support many languages. Rascal comes with its own DSL focused on code analysis and transformation. Rascal has extensions for C code analysis, however, these are still in development. At the time of writing only the support for Java is mature.

4.2 Evaluation steps and criteria

The goal of the tool evaluation process is to make an informed decision on the tool to further use in the project. Each tool is evaluated according to the evaluation process which is defined below. The tools are scored on a number of aspects related to the process steps.

The evaluation process is as follows:

1. Install the tool.
2. Using the tool, build an AST of a minimal example and a larger example.
3. Analyze the AST produced by the tool for completeness and detail.
4. Interface with the tool programmatically to build and output and AST.

There are a number of factors on which the tools are scored. For each score an explanation will be given.

- Installation and configuration: are there any installation or configuration issues?
- Performance: how much time is needed to parse the examples?
- Data quality: how useful and precise is the AST?
- API quality: is it easy to interface with the tool?

– Language support: which variants of C are supported, and are other languages supported as well?

The relative performance of the tools under evaluation is measured by compiling a small benchmark program consisting of a main file, a library and a header file for the library. The benchmark program is a command line program for interacting with a doubly linked list that was used for a Software Security course at the University of Twente. This small benchmark should give an approximation of the performance since building an AST is an integral part of compiling a program. Since not all three tools support dumping an AST from the command line benchmarking the whole compilation process gives the most comparable results.

The performance benchmark is executed with a Python script running on Python 3.5. For each of the tools under evaluation, the script compiled the benchmark program exactly 10 times, measuring the total execution time. The `subprocess` module was used to call the executables of the tools. The system used to run the benchmark on is a relatively modern dual-core laptop computer running Linux.

The program used for the performance benchmark is also used for evaluating the data and API quality, together with a trivial program consisting of only a main function and a return statement.

4.3 Results

The characteristics of each individual tool with respect to the evaluation criteria are discussed below.

4.3.1 C Intermediate Language

For CIL there are clear instructions for installation. First of all OCaml and OPAM, the OCaml package manager, need to be installed. This is fairly straightforward as these are available as packages on the test system. The installation of these dependencies is painless and no configuration is necessary. An OPAL package for CIL exists to make installing it a matter of running a couple of commands.

The script provided by CIL to function as a drop-in replacement for GCC first calls GCC to precompile the code. After CIL processed it, it is then again compiled by GCC. Therefore it is reasonable to assume that this approach will be slower than directly using GCC. In the performance test CIL consistently took just over 1.7 seconds. The test case was also compiled with just GCC, which took around 0.9 seconds. This makes CIL almost a factor two slower than regular GCC.

The data structures provided by CIL are detailed. It provides a number of extensions to a plain AST, including data structures for control flow and data flow analysis. The data structures are documented relatively well.

CIL libraries are available for the OCaml functional language. The API has been documented and some additional instructions are available. Sadly, all I got from CIL was a syntax error on the input file, even when trying to parse the trivial program. This is unexpected given that other researches have had success with CIL and that the command line compiler using CIL is able to produce a working program.

CIL only supports the C programming language. For this language it supports most GCC extensions, according to the authors. The authors claim to have compiled the Linux kernel with CIL, which is usually a good indicator that the compiler can handle most other projects as well.

4.3.2 Clang

Clang was very easy to install. Version 4.0 was present in the package repositories of the test system, so a single command installed Clang successfully. Due to issues with the Python bindings as described below under 'API quality' I opted for Clang 6.0 which is available from the official LLVM package repository, which I added to the software sources of the test system. This was therefore very easy as well.

Clang is a compiler front end and uses the AST for compiling the code to the intermediate representation of the LLVM compiler. Therefore it can be expected that the AST building is optimized for performance. For these small programs it seemed that writing the output to the terminal took more time than actually building the AST. Clang took just under 1.5 seconds in the performance test, which makes it quicker than CIL, but considerably slower than GCC. The relatively large difference is worth mentioning since the speed of a compiler is of importance for larger software projects.

The AST provided by Clang is very extensive and low-level. Data quality is therefore very good. However, some flattening of the tree might be needed in order to more easily compare nodes with each other. A downside of this level of detail is that knowledge of C specifics is required to properly process them.

Clang has an extensive API for which C libraries are provided. For the default `libclang` library there are also Python bindings available. The Python bindings were used since 'playing' with the data structure in a Python shell is much easier than writing C code for the same purpose. It turns out that Clang and the Python bindings don't play well together if they are not the exact same version. Also, the Python 3 compatible bindings are only available with later versions of Clang. The Python bindings in the package repository are for Python 2 only, therefore the Python source code from

the GitHub repository[1] was used. This worked well together with Clang 6.0. It was very easy to access the AST as documented in the C library documentation.

Clang has support for a number of languages in the extended C family, including C++, Objective C and OpenCL C. There is, as expected, no support for other languages. The C language support is good and many GCC extensions are supported by Clang.

LLVM

The LLVM intermediate representation can contain metadata about the original program. This intermediate representation (IR) is much simpler than C code. It is therefore easier to perform analysis on this code after the complexity has been taken care of by the front end compiler, which is Clang in for the C language. The LLVM optimizer already contains a number of analysis algorithms, including control flow graphs and memory dependence analysis. Many parts of the IR can be traced back to a specific location in the original code, making the analysis useful for our purpose. By default LLVM does not output sufficient data for the purposes of this project, but the optimizer can be extended with additional passes to get the data out of the system.

4.3.3 Rascal

Rascal can either be used as a standalone JAR or with an Eclipse plugin. It is noteworthy that Rascal requires a JDK to run, only a Java runtime environment is not sufficient. A separate plugin provides C analysis capabilities. Eclipse update repositories are available for both plugins, making installation very easy. There is no manual configuration required.

I was not able to accurately measure the time taken by Rascal to parse the example code into an AST. This is due to the fact that this can only (easily) be done from their own console, there is no single command line program that can be run and timed. The command to parse the code in the Rascal console returns reasonably quick, however, it is impossible to rank this in comparison to CIL and Clang.

Rascal's tree representation is equally detailed as the representation of the other two tools. It does not add control flow information to the tree by default. Rascal provides a domain-specific language (DSL) to work with the AST. Because of its limited purpose this language is very suited to the task of analyzing source code. The data structures Rascal provides for ASTs are very generic and are designed to be extended by specific language implementations. The C implementation seems to provide the necessary data structures for the supported C90 grammar. However, because of a lack of

[1] The Clang GitHub repository is located at `https://github.com/llvm-mirror/clang`.

	CIL	Clang	Rascal
Installation and configuration	−	+	+
Performance	=	=	
Data quality	+	=	−
API quality	=	+	−
C language support	=	+	−
Other language support	−	=	+
Overall	−	+	−

Table 4.1: Scoring table of the tools under evaluation. A score is either positive (+), neutral (=) or negative (−).

documentation and the fact that both Rascal and the extension for C support are still in development, I was not able to get any practical use out of this tool. Data and API quality are therefore considered to be poor, with the side node that this might improve over time as the code repository seems to be active.

I have not found any claims regarding the level of support for the C programming language. Rascal includes by default a C90 parser which would not be sufficient for modern software. Rascal is extensible to support multiple languages, however, currently only Java is relatively well supported.

4.4 Conclusions

The relative score resulting from the evaluations as described above is shown in table 4.1. Each tool has been scored on each factor. No score is given if it was not possible to evaluate that part of the tool.

At first the 'overall' score in the table was indented to represent a score for the tool considering the evaluated factors. Because only one evaluated tool actually works as expected it is unfair to give a positive rating to an tool that has not been observed in a usable state.

Besides the fact that Clang turned out to be the only properly working tool, it has scored well on the criteria that were defined beforehand. This might be due to the relatively large user base of the tool. Of the evaluated tools Clang seems to be the only one being used (at least with C code) outside of a research context.

Given that Clang scored well on the evaluation criteria and works as expected it will be the tool of choice for this project. Of course the algorithms do not depend on any specific tool.

Chapter 5

System architecture

5.1 Requirements and considerations

For this project a number of requirements have been given by the client, ALTEN Netherlands. These requirements are listed in appendix A. Besides these requirements certain aspects of the chosen code analysis tool (see chapter 4) and algorithms (see section 3.2) need to be taken into account.

The abstract syntax tree (AST) produced by Clang is very detailed and, as one would expect, tied to C structure and context. Some data that is required for the analysis algorithms, which are discussed in section 3.2, is not encoded in the AST, so additional analysis has to be done beforehand. Due to the complexity of some parts of the C language this is relatively hard. The Clang compiler compiles C code to the LLVM intermediate representation (IR) which is easier to analyze. It also allows tracing back instructions from the IR to the original C source code. Therefore it makes sense to do parts of the analysis with LLVM and annotate the Clang AST with information obtained from LLVM.

Because the C parsing and preprocessing will be performed by different programs, it makes sense to have an analysis program that can receive data from any source. This combined with having a programming language independent representation of the program as input format results in the possibility of supporting multiple programming languages.

The algorithms for extracting the changes from the source code all work in a similar way, taking two labeled trees as input and producing an edit script for these trees. This edit script can be used to tag modified nodes to limit the analysis to. The analysis algorithms require both annotated abstract syntax trees and the edit script as input.

5.2 Architecture decomposition

5.2.1 High level architecture

On the highest level, three distinct components can be identified from the requirements. The first component, Parse, is responsible for parsing the source code into an internal representation of the AST. This component is also responsible for adding some analysis data specific to the programming language. Because two versions of a program need to be compared there are two independent Parse components. Secondly, the differences between the two versions need to be calculated. The Diff component serves this purpose and takes the ASTs produced by the Parse components as input. Given the ASTs and the calculated changes, the actual analysis can be performed by the Analyze component. Finally the results need to be reported to the user, for which a Report component exists. The composition of these components is shown in figure 5.1.

Of course a system for configuring and controlling these components needs to be in place, however, these have been omitted from the architecture as these are not considered a core part of the system.

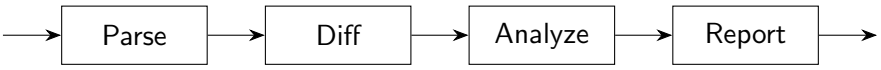


Figure 5.1: Composition of the high-level subsystems.

5.2.2 System components

Parse

The Parse component consists of a number of parsers, each with a different implementation. A parser is implemented for a specific programming language or a specific toolset for parsing code. The parser parses two programs for two-way analysis and three programs for three-way analysis. Both versions that will be merged need to be parsed, and in the case of three-way analysis their common ancestor needs to be parsed as well.

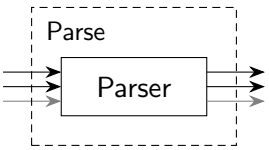


Figure 5.2: Contents of the Parse component for a single run.

The Parser component accepts a program as input and is responsible for generating and parsing an intermediate analysis representation of the program. It also adds additional analysis information to the intermediate

representation. While the exact implementation of this component might differ greatly for different programming languages some detail is given about the design of our C parser.

The C parser requires the path to the file containing the root of the program as input, just as a C compiler would. It then uses the Clang Python bindings to compile the source code into an AST. It also requires analysis performed by LLVM. This analysis is exported to a file using an extension module for the LLVM compiler. This file is then parsed by the parser after which the data is combined with the Clang AST and put into an internal data structure.

The architecture of the C/Clang version of the Parse component, denoted by Parser:Clang, is shown in figure 5.2. In this diagram the Compile AST component represents libclang, the interface with the Clang compiler. The Compile LLVM component represents the regular Clang compiler which compiles the C program into LLVM byte code. The LLVM analysis component represents the LLVM optimizer with the analysis extension library that brings some static analysis results from the LLVM compiler to the Parser component.

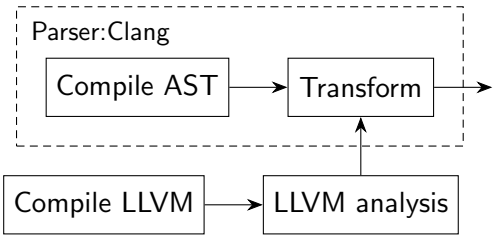


Figure 5.3: Composition of the Clang specific Parser component and supporting programs.

Diff

The Diff component calculates the difference between the parsed abstract syntax trees. The diff compontent takes the ASTs as input and returns the mapping between corresponding nodes in the trees and the changes that can be derived from this mapping. The algorithm that is used in the implementation is described in section 6.2. Pseudocode of the algorithm implementation can be found in appendix B.

Analyze

The Analyze component contains algorithms for identifying changes and comparing the ASTs in order to check the input for possible problems after merging. The analysis component consists of a number of distinct analysis algorithms, which are described in section 3.2. These take the ASTs and the

results from the Diff component as input and produce detected conflicts. Each detected conflict consists of a number of conflicting changes and information on the details of the conflict. Each conflict is assigned a score based on the severity of the conflict. The composition is shown in figure 5.4.

The Analysis components, containing the analysis algorithms, are designed to work independent of each other. Therefore the execution order is not important. Since the algorithms do not modify existing data these could also be executed in parallel.

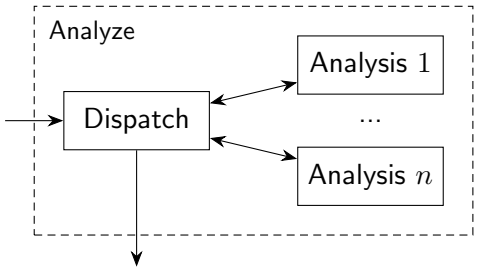


Figure 5.4: Composition of the Analyze component.

Report

The Report component takes the output of the analysis and aggregates scores. It also reports the specific detected problems to the user. The detected problems have scores and the problems are sorted based on this score. A total and average score of the analysis is given to provide insight into the 'danger' of performing the merge.

The developed proof of concept does not support configuring the severity of each kind of conflict to suit specific project needs. Also, no dynamic severities are implemented. Instead, each type of conflict is assigned a constant score to make a distinction between the types of conflicts. An example of a dynamic severity of a conflict would be to have the severity depend on the number of changes in the conflict.

Chapter 6

Implementation

A prototype of the system has been implemented in the Python programming language. The system architecture as described in the previous chapter is used as structure for the subsystems and interfaces. The system uses a plugin system allowing for easy integration of extensions to the functionality.

6.1 Framework

A small framework has been created from the architecture description. The framework additionally includes an intermediate data representation and a plugin system for extensibility.

6.1.1 Internal data representation

The abstract syntax trees are internally represented by a more generic tree structure. The internal data representation has been designed to have a small memory footprint. It therefore does not hold all data that a proper abstract syntax tree would. Instead, the tree nodes contain only the essential information that is required by the algorithms. For extensibility each node can hold additional, unspecified metadata. At this moment the data in the intermediate representation has been selected based on the requirements of the analysis, and the parser is responsible for providing that data. When the system is developed further it might be a good idea to spend time investigating a different data representation that provides more flexibility. One possible solution is discussed in chapter 8.

Dependencies and references

Both dependencies and references are a unidirectional relation between two nodes. For the purpose of the internal representation both are seen as a dependency on another node. A dependency has a type, which is either a control dependency, a kind of memory dependency or a kind of reference.

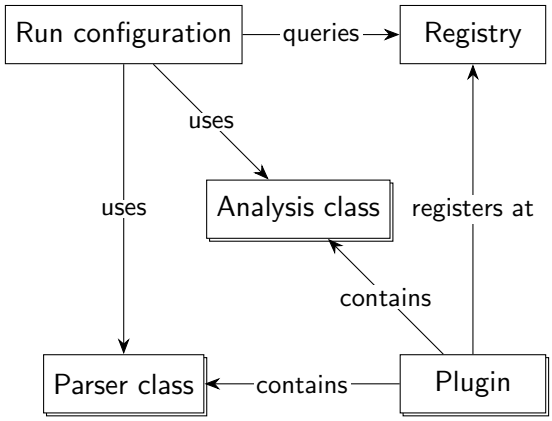


Figure 6.1: Abstract diagram of the plugin system.

Changes

A change is represented as a combination of two nodes, each from a different AST, and a change operation. At this moment three change operations are supported: an insert, a delete, and a rename. Changes are modeled as a 3-tuple containing the node in the first version, the node in the second version, and the change operation. Either one of the nodes might be absent in case of an insert or delete operation. A different diff algorithm might support more change operations in the future, for example, moved subtrees.

6.1.2 Plugin system

For extensibility purposes the system has been designed to be plugin-based. There is a very small base system that consists of code for running the analysis, the intermediate representation and abstract classes that can be implemented by plugins. The default plugin adds implementations for a diff algorithm and some analysis algorithms. The C parser has been implemented in a separate plugin.

Plugins are loaded dynamically by looking for certain files on the Python path. Each plugin can define different implementations that it provides. These implementations are registered in a central registry and can be used when running the system. Figure 6.1 shows an abstract view of the components and their interactions.

6.2 Tree differencing

Tree differencing is used to compute the changes between the abstract syntax trees of two versions of a program. Typical algorithms take two labeled trees

as input, and yield a one-to-one mapping of the nodes in the trees that are considered equal. Non-mapped nodes are either inserted or deleted, mapped nodes with different labels are renamed.

In this section a number of algorithms are presented and the chosen algorithm and its implementation are discussed in more detail.

6.2.1 Considered diff algorithms

The ChangeDistiller algorithm combines comparing the structure of a subtree and the string labels of nodes for mapping nodes of two trees [11]. Leaf nodes in the tree are mapped if their labels (AST node types) match and the string similarity of the values (concrete syntax) is above a certain threshold. For subtrees the same comparisons are used, with the addition that the number of matched leafs compared to the maximum number of leafs in one of the subtrees must exceed a certain threshold.

By default, ChangeDistiller works with coarse-grained AST nodes, resulting in small edit scripts but relatively large changes. This is not ideal for our purpose as detailed changes will result in detailed and specific problematic changes that can be detected. If the extracted changes are relatively large the risk exists that a large number of these changes will be marked as problematic, while making it unclear in which part of the change the problem exists.

The RTED algorithm (Robust Tree Edit Distance) is presented as an efficient way of calculating the tree edit distance independent of the shape of the trees [25]. The authors claim that their algorithm computes at most the same number of subproblems that the competitors need to compute, while being more efficient when the competing algorithms run into the worst case scenarios. RTED computes the optimal edit script by performing an exhaustive search on all possible strategies for calculating the tree edit distance and only doing the actual calculation on the optimal strategy.

RTED does not consider move operations, only insert, delete and rename. This limitation allows the algorithm to be a lot faster compared to algorithms that do consider move operations. Also, computing moves cannot always be done properly as some parts of code tend to repeat many times, for example a variable. However, this approach may lead to a large edit script when existing code is wrapped in another statement, for example when code is wrapped in a conditional statement.

The GumTree toolkit has been developed by researchers at the universities of Bordeaux and Lille and is designed to find edit scripts of abstract syntax trees that are short and close to the developers intent [10]. Gumtree works in three phases. First the tree is iterated top down mapping nodes with the same hash. The hash that is used for this comparison contains a nodes label and value and those of the descendants. Therefore this first phase will only match exactly matching subtrees. Secondly the tree is it-

erated bottom up to map nodes with a lot of matching children, signifying renames. Finally an edit script is computed for the remaining nodes, for which the RTED algorithm is used.

Dotzler and Philippsen proposed a number of optimizations for shortening the computed edit scripts to the aforementioned algorithms [9]. They describe five different optimizations that are applicable to all considered algorithms. The optimizations show improvements on the quality of the mappings as in all cases the optimizations resulted in less than 1.2% of the tests in worse results. The run time of the algorithms does seem to suffer in the case of RTED and GumTree, while the optimizations shorten the average run time for ChangeDistiller.

In conclusion, the more recent algorithm, GumTree, promises the best performance, both in edit script size and time. ChangeDistiller performs not as good and is less suitable for the level of detail that we like to have. Since RTED is used on the subtrees that were not matched by GumTree in the first phase, GumTree could be seen as an optimization for RTED.

6.2.2 Tree differencing implementation

The tree differencing algorithm that is used is an implementation of the GumTree algorithm as described in [10]. Several tweaks to the pseudocode have been made in order to make it more suitable for implementation in Python.

A relatively simple implementation of the *opt*() function in the algorithm has been used. This part of the algorithm matches subtrees if their similarity is above a certain threshold. The similarity is computed based on the shape of the tree and the similarity of the node types and labels. The specific algorithm that is chosen for computing the similarity is the Zhang-Sasha algorithm [34]. This algorithm is chosen in favor of the RTED algorithm that is described above as a working library for the Zhang-Sasha algorithm was available [13], while a working version of RTED was not. The similarity of the nodes based on their type and labels is determined by the Levenshtein distance, which denotes the similarity of two strings. The specific algorithm that is used is the Wagner-Fischer algorithm [30].

The optimizations described by Dotzler and Philippsen [9] were not implemented due to time constraints.

The implementation specifics of the tree differencing algorithm can be found in appendix B.

6.3 Static analysis

The changes that are found in two versions need to be analyzed to find problematic changes as discussed in chapter 3. For this purpose the detection

```python
from checkmerge import CheckMerge, RunConfig
from checkmerge.analysis.dependence import DependenceAnalysis
from checkmerge.analysis.reference import ReferenceAnalysis
from checkmerge.diff import GumTreeDiff
from checkmerge_clang.parse import ClangParser

# Make sure CheckMerge is setup properly
if not CheckMerge.ready():
    CheckMerge.setup()

# Create a new run configuration
run = RunConfig(parse_cls=ClangParser, diff_cls=GumTreeDiff)

# Create a run with a tree-way diff and two analysis algorithms
analysis_run = run.parse('a/program.c', 'b/program.c', '0/program.c'
    ).diff().analyze(DependenceAnalysis).analyze(ReferenceAnalysis)

# Get the analysis results
conflicts = analysis_run.analysis()
```

Listing 6.1: Usage example of the declarative API. This example is without using the plugin system.

algorithms listed there have been implemented. As the pseudocode in chapter 3 has been written based on the prototype implementation in Python there are no significant differences between the pseudocode and actual implementation.

6.4 Interfacing with the system

The system provides two ways to interface with it. There is a command line interface for direct use, and a declarative API for integrating the system into another program or to turn it into a web service.

6.4.1 Declarative API

A single class, `RunConfig`, contains all logic to set up an analysis run. When it is initialized with a valid parser and diff algorithm, methods can be chained to form an analysis chain. Afterwards, or mid-way, result data can be extracted. An example is shown in listing 6.1.

For each chained method a copy of the run configuration object is made, which is returned. Intermediate results are cached. This allows for easy construction of analysis pipelines.

```
python -m checkmerge.cli analyze -p clang -a dependence -a reference
    a/program.c b/program.c 0/program.c
```

Listing 6.2: Usage example of the command line interface.

6.4.2 Command line interface

The command line interface can be accessed by running the `checkmerge.cli` Python module. Via the command line it is possible to diff and to analyze programs, as well as to view the installed plugins and the parsers and analysis these provide. When diffing or analyzing two programs one of the installed parsers must be specified. When analyzing two programs, one or more analysis algorithms must be specified as well. Instructions for using the command line interface can be obtained by running it with the `--help` flag. An example of a command line invocation is shown in listing 6.2, performing the same task as the AST example above.

6.5 C support with Clang and LLVM

The C support plugin uses the Clang Python bindings to derive the AST from a program. A custom LLVM analysis pass is provided to use the LLVM optimizer to get dependency data to enrich the AST with.

6.5.1 Custom LLVM pass

LLVM, being a compiler back-end, can perform a lot of different analysis on transformed C code already. However, this data is mostly kept internal and is not exposed via a command line interface or library. Some analysis functions in LLVM do have output, but this output is mostly aggregates and no raw data.

With an analysis pass custom code can be dynamically loaded into LLVM and executed on parsed code. The custom pass declares its dependence on the memory dependence pass, making sure it is executed. Then the memory dependence information is queried and formatted into a YAML file, which is written to the file system to be read later on by the analysis system.

Because the analysis of LLVM works with the LLVM IR instead of the C code directly, the IR code is compiled with debug information, allowing instructions to be traced back to the original C code. The memory dependence information that is saved is combined with file names and line and column numbers for that purpose. The matching of the memory dependence information with the C code is done in the parser of the analysis system.

6.5.2 Clang parser

The C code is parsed using the Clang compiler front-end. The AST is extracted from the compiler using the public C API, libclang, for which Python bindings are available. The parser traverses the AST, creating the nodes of the internal representation of the tool. Pointers to other nodes in the AST, for example to nodes defining a used function, are kept in dictionaries until the whole AST has been built in the internal representation. This way every pointer can also be made in the internal representation.

Most AST nodes can be parsed in a generic way without considering the specific kind of node. However, due to limitations of libclang, some types of nodes need a special treatment. An API is provided to easily add more special cases in the future when needed.

Chapter 7

Results

7.1 Evaluation

To make sure the prototype behaves as expected, several unit tests are added to the source code. Also some small code examples were created to do preliminary testing of the algorithms. The prototype is also tested using a larger test suite consisting of example programs.

7.1.1 Test plan

The prototype is tested and evaluated using a custom test suite containing programs that were manually created or altered for the purpose of the tests. Each test in the test suite consists of:

- An 'ancestor' version of a program as a single file of C code.
- Two distinct versions of the same program, each as a single file of C code.
- A summary of the changes in both versions, and a list of the expected problems when merging the two versions.

A test is executed by getting a three-way diff of the programs and running both analysis algorithms. The results from the algorithms are compared to the list of expected problems. Problems that are detected but not on the list are further inspected to check if they are a false positive or an actual problem that was not caught by manually inspecting the code. If the detected problem is in fact a valid problem, this change is added to the list for that test case. From these results the following metrics are obtained:

- Precision: fraction of correctly identified problems (true positives) in relation to all detected problems (true positives and false positives).

$$\frac{|\{\text{true positives}\}|}{|\{\text{true positives}\} \cup \{\text{false positives}\}|}$$

– Recall: fraction of correctly identified problems in relation to the expected problems.

$$\frac{|\{\text{true positives}\}|}{|\{\text{true positives}\} \cup \{\text{false negatives}\}|}$$

Additionally, the following metrics are recorded:

– Total lines of code (SLOC) in the three versions.

– Run time for calculating the differences.

– Run time for each analysis algorithm.

These metrics should give an insight into the quality of the results as well as the time performance of the algorithms.

7.1.2 Test cases

Ideally the evaluation should be done on real-world projects, either by examining existing merges (where not a lot of problems are expected) or by creating new merges of versions that were not intended to be merged (where more problems are expected). This would require manually examining the merge to determine any false positives or false negatives, so a lot of automation cannot be achieved. Also, the performance is not yet good enough to use large examples. This makes using real-world tests not feasible.

A number of test cases have been prepared, and each is described below. The test cases are either created specifically for this purpose, or adapted from real-world code.

`calc` function test cases

A number of small and simple test cases based on the previously used `calc` function have been created, each representing a category of changes as listed in chapter 3.

The `calc_1.c` test case has a change in the same position in each version. Since these kinds of problems are not explicitly detected by the tool, no conflicts are expected.

The `calc_2.c` test case is illustrating a double bug fix. This test case is the example from listing 3.1. A single conflict highlighting the two changes is expected.

The `calc_3a.c` test case is illustrating an incomplete refactoring. This test case is the example from listing 3.2. A single conflict highlighting the changed definitions and added variable reference is expected.

The `calc_3b.c` test case has one parameter removed and replaced with the remaining parameter in the first version, and a reference added to the removed parameter in the second version. A reference conflict with the removed parameter and added parameter reference is expected.

`math.c` test case

A file with some math-related functions was written to test a number of change patterns. Most of the changes are not problematic, but some problematic refactorings are present. There is one group of changes that alters the behavior of a for-loop. While this is not explicitly tested, the problem occurs in the assignment of the variables in the conditions of two nested for-loops. A single merge conflict is present in the file.

`dll` test cases

The `dll` program source code is taken from a course on software security. The base version is the code as provided for the students to work on, one of the versions is the code as submitted for an assignment, the other version is the same code with some alterations. The differences of the two versions in comparison with the ancestor are large, the differences between the two versions are not.

The test cases contain a number of changes, most of which are problematic. Most changes fall solely in category PC1. The `dll.c` file has 6 merge conflicts based on the Unix diff program, but no conflicts for the new algorithms are expected. The `main.c` file has 5 merge conflicts based on the Unix diff program, and has a reference conflict as well, which should be detected by the reference analysis algorithm.

`sds` test case

The `sds` (Simple Dynamic Strings) library is a C library that implements dynamic strings. The test case consists of three versions of the code taken from the GitHub repository[1]. The base version is a relatively old version of the code, the two compared versions are a bit newer. The newest version is a continuation on the older version, so there is technically no merge, but there are enough changes to make the results interesting. Some merge conflicts (category PC1) exist. There are a lot of type refactorings, for which no algorithm is available. One detected memory dependence conflict is expected.

7.1.3 Results

The test cases have been evaluated on a modern laptop computer (Intel Core i5-7300U, 16GB RAM) running Linux. The total lines of code is the sum of the lines of code for each of the three source code files in the test case. The run times have been measured by storing time stamps when specific parts of the code are reached. The measurements have been taken multiple times, the best performing result has been taken to account for background

[1]The sds GitHub repository is located at `https://github.com/antirez/sds`.

processes and other external influences. The command line interface was used for running the tests. The lines of code (SLOC) was measured using `sloccount` [32], the elapsed time was measured by comparing timestamps taken by the system while executing the analysis.

Table 7.1 contains the characteristics of each test case, including the lines of code and run times of the test case. There is no direct relation between the number of AST nodes and the lines of code as the nodes from included files are also parsed. This explains the small test cases having a large number of AST nodes. Also, some of these included nodes are not matched by the algorithm. This is due to the height and location of the subtrees, if a subtree is too small it is not matched by the algorithm. Examples of these nodes are constants.

It is notable that the diff step takes the most time and scales exponentially with the size of the program, which is expected based on the characteristics of the algorithm. To match the most similar subtrees iterations over the cartesian product of subtrees are necessary (see appendix B). While the `dll/main.c` test case is, in total, smaller than the `dll/dll.c` test case it takes more time to calculate the differences. This can be explained by the fact that the amount of code in the three versions of the `dll/main.c` test case is roughly equal, while the base version of `dll/dll.c` contains only function definitions without any implementation. This greatly speeds up two of the three diffs that are calculated.

The results for each test case are shown in table 7.2. The first four tests with a single problematic change give a positive result as the problematic changes are identified. In case of the refactorings also a memory dependence conflict was found, which are false positives. For the precision two numbers are given. The precision 'with PC1' assumes that detected problematic changes are true positives if the detected changes include a merge conflict. The precision 'without PC1' is the precision without this assumption.

It is notable that in all test cases except one the recall was 1.00, meaning that the algorithms were able to identify the problems in almost all of the test cases. One problem of category PC2 was not detected. This was pinpointed to the fact that the conflicting changes are located in different basic blocks. The current implementation of the LLVM bindings limit memory dependencies to within a basic block. Therefore the node relations necessary to detect this change are simply not present in the data.

All identified false positives were of category PC2. It is clear that algorithm 3.1 yields a relatively large number of false positives. The algorithm was designed to sacrifice precision for recall, and is therefore expected to yield a relatively large number of false positives. The algorithm performs best when there are not many changes in a piece of code, for example in the `calc_2.c` test case. When a lot of changes are made, the algorithm starts to detect a large number of possible conflicts, most of which are a false positive.

	Number of SLOC[a]	Number of AST nodes[b]	Number of AST changes	Parse time (s)	Diff time (s)
`calc_1.c`	18	2 008	137	0.5	4.5
`calc_2.c`	18	2 008	138	0.5	4.4
`calc_3a.c`	18	2 007	148	0.5	3.9
`calc_3b.c`	18	2 006	143	0.5	3.8
`math.c`	181	2 811	311	1.0	207.5
`dll/main.c`	267	6 783	207	1.9	341.3
`dll/dll.c`	372	8 062	224	2.4	50.8
`dll/dll.h`	72	222	9	0.7	6.4
`sds/sds.c`	1 681	18 093	544	13.2	6 762.7

Table 7.1: Test case characteristics.

[a] All parsed code of all versions, not including dependencies.
[b] All parsed nodes of all versions, including dependencies.

		Precision			Analysis time (s)	
	Number of detected conflicts	With PC1	Without PC1	Recall	Dependence	Reference
`calc_1.c`	0	–	–	–	0.01	0.01
`calc_2.c`	1	1.00	1.00	1.00	0.01	0.01
`calc_3a.c`	2	0.50	0.50	1.00	0.01	0.01
`calc_3b.c`	2	0.50	0.50	1.00	0.01	0.01
`math.c`	10	0.20	0.20	0.67	1.97	
`dll/main.c`	12	0.75	0.08	1.00	2.19	0.03
`dll/dll.c`	7	0.00	0.00	–	1.95	0.04
`dll/dll.h`	0	–	–	–	0.00	0.00
`sds/sds.c`	10	0.50	0.10	1.00	473.7	

Table 7.2: Qualitative and quantitative performance of the algorithms in the prototype. Missing values could not be calculated due to a division by zero.

This also impacts the run time of the algorithm negatively as can be seen from the `sds/sds.c` test case.

There is a lot of similarity between distinct problems identified by algorithm 3.1. This suggests that the 'largest superset' approach taken by algorithm 3.2 might not be sufficiently optimizing the results. If many changes exist within a basic block many combinations of these changes are identified as distinct conflicts.

Merge conflicts tend to result in one or more conflicts detected by algorithm 3.1. Due to the greediness of this algorithm a large part of a basic block is covered from one node. This results in the detection of almost any change within a single basic block. Combined with the duplicates in the results, due to different start nodes, this results in many false positives.

7.2 Known limitations

The prototype currently has a number of limitations. For a number of limitations possible solutions are proposed. Some of these limitations are due to issues that are out of the scope of the project. Others could not be implemented due to time constraints. A list of the limitations is given below.

- Single files are evaluated, changes in dependent implementation files are not taken into account. Changes in included header files are taken into account.
- For many AST node types custom parse rules are needed to properly parse them due to limitations of libclang.
- The current internal representation is somewhat limited, it might require changes when new algorithms are added.
- Not all identified problematic changes are currently detected.
- The programs run quite slow, the algorithms or algorithm implementations need performance optimizations.

Chapter 8

Related work

Little research has been done on static analysis of code merges specifically. The subject of software evolution is closely related though, as this also involves changes between two versions of a program. Therefore the research discussed here is not directly comparable to this study. A number of relevant papers has already been discussed in chapter 2.

8.1 Generic abstract syntax trees

The internal representation used by the prototype does not support all features that nodes in an AST might have. To be fully extensible, and to be able to support more different languages, this representation has to be extended. Meanwhile, other researchers are working on a more generic AST that is designed to support many programming languages without information loss. Babelfish (bblfsh) is a relatively new framework for code parsing, described by the authors as a "universal code parser" [3]. The purpose of the project is to provide language independent parsing for any programming language. The AST representation used by Babelfish holds all features that a language-specific AST would, making code analysis of different languages easier.

While the UAST (universal abstract syntax tree) format that Babelfish uses should encode a full AST, any additional information provided by the native parser (e.g. Clang for C or JavaParser for Java) is most likely to be lost. An example of this is direct pointers from the usage of a named entity to its declaration. This analysis should be done (again) with the UAST as input, trading flexibility and a simpler native parser for the work of re-implementing certain analysis steps.

When starting this Babelfish was in early development and therefore not considered. Since then a lot of progress has been made, making it seem like a more viable choice to use for a project like this one. The C language is at the time of writing not (yet) supported.

8.2 Source code differencing

Considerable effort has been put in source code differencing as textual source code differencing has been known to perform worse than methods taking the syntax of the code into account. Fluri, Würsch, Pinzger and Gall describe an abstract syntax tree based source code diff algorithm [11]. This algorithm finds the differences between two versions of a code base on a syntactic level. The research is based on an earlier tree differencing algorithm which was intended for LaTeX documents. This algorithm was not directly applicable to source code since in most (if not all) common programming languages non-leaf nodes also include relevant information, resulting in suboptimally large change sets. An example is the condition of a loop. Fluri et al. proposed and benchmarked adaptations to make the algorithm viable to use for source code. The benchmark results suggest that the algorithm is promising, however, a comparison with another similar algorithm is missing, therefore the only conclusion is that the changes to the earlier algorithm helped in correctly identifying changes in source code.

Another abstract syntax tree approach is described by Neamtiu, Foster and Hicks [21], which predates the research of Fluri et al. This approach assumes that function names remain constant. This is necessary to be able to base the detection of changes within functions on these function names. The matching of two ASTs is performed with common graph techniques like finding bijections mapping parts of the code in one version to the other version. This approach, including the described algorithm, is relatively simple but may not work in more challenging situations. The algorithm is clearly geared towards smaller changes within single functions but is not useful at all when looking at (larger) refactorings.

The GumTree algorithm by Falleri et al. is more recent than the methods described above. This algorithm has been compared with other, at that moment state-of-art, algorithms, showing a performance increase both in quality of the result and in average run times [10]. This algorithm has been used in this project, and is described in greater detail in section 6.2. At the time of writing there is no publication describing a new, better algorithm, although some optimizations exist [9].

Maletic and Collard propose an XML-based differencing approach of source code with respect to its syntax [18]. This approach requires that versions of a program are converted to the XML representation srcML. Next, the differences can be calculated which are, together with both versions, stored in another XML format srcDiff. This XML can then be queried using XQuery for further analysis. This approach combines an intermediate representation, which still contains programming language specifics, and a differencing algorithm. Due to the level of detail of the XML reprensentation a translation has to be written from the raw compiler AST to srcML.

Asenov et al. describe a merge algorithm for existing version control systems that takes tree structures, like source code, into account [1]. The output of a textual difference algorithm is used to build a tree representation of only the changes. This tree representation is then formatted in a way that a standard textual difference algorithm can compute the changes more precisely than without this additional step. The algorithm can be modified for specific application domains which makes it very versatile. This research focuses primarily on improving merges and reducing the number of merge conflicts rather than just calculating the difference between programs.

8.3 Static analysis of source code changes

Most research into the analysis of changes in source code seems to be focused on software evolution, comparing a version of a program with an older version. Typical goals are gaining insight into the development of a software program and helping developers write maintainable code.

Baxter et al. use abstract syntax trees to detect clones in source code [6]. The research is focused on the traditional antipattern of copying and pasting larger fragments of code instead of writing a proper function to handle generic cases. For this reason small, trivial duplicates, like simple mathematical computations, are excluded from the results.

Refactorings are generally larger, more involved changes. They are usually changes that require modifications at multiple points in the source code. Also, code is refactored often. This combination presents a large risk of breaking code while refactoring. Weissgerber and Diehl show a method for finding refactorings in source code [31]. The method presented in their paper uses signature-based analysis together with clone detection to identify refactoring candidates. When computed for two subsequent versions a refactoring can be identified and the candidates can be compared to find incomplete refactorings. For the Tomcat project this method is able to identify 77% of the refactorings that were documented. Additionally 73% of the identified refactorings were actual refactorings based on inspection of the source code. These numbers increased to 80% and 92% when only looking at structural refactorings (moving classes, interfaces, fields and methods and renaming classes).

Some refactorings are not identified by Weissgerber and Diehl. Godfrey and Zou describe a method for identifying the splitting and merging of source code entities like classes and functions [12]. These are refactorings that are common but relatively hard to identify. They describe understanding the evolution of a code base as a primary goal. Due to the fact that manual input from the user is required, especially when the analysis should be performed in reasonable time, this method might not be suitable for programmatic use.

Silva and Valente created a program for finding refactorings in Java programs called RefDiff [28]. The authors claim that the program can detect many kinds of refactorings, including the ones covered by the previously discussed approaches. According to the verification carried out by the authors RefDiff performs very well with an accuracy of 100% and a recall of 88%.

Chapter 9

Conclusions and recommendations

9.1 Conclusions

The goal of this research was to design and implement a prototype of a system for assessing the risk of side effects of code merges. To achieve this goal, a number of categories of changes that can cause issues with merges were specified. Algorithms to detect some of these changes were developed and implemented in a prototype system. The changes were extracted from calculating the differences between abstract syntax trees (ASTs). The algorithms were evaluated by analyzing a number of artificially created merges as real-world test data is not readily available.

Different categories of changes were identified (Q1). A method for finding the semantic changes in the code was described and implemented (Q2). Two new algorithms were developed and implemented to detect problematic changes for three of the identified categories (Q3).

The results presented in chapter 7 clearly show that the two new algorithms are able to identify the problematic changes. These also show that the results also include a large number of false positives, which was expected given the nature of algorithm 3.1.

All in all, the primary conclusion is that a risk analysis system for code merges is feasible. The presented algorithms can identify the problems, but lack some precision as a large number of false positives are given. While the prototype produces good results, its time performance is not at a level that makes it practical to use in production. In order to have a production-ready tool some improvements, which are discussed in the next section, need to be made.

The client is satisfied with the results, as these show that it is possible to find problematic changes in merges. The current results of the algorithms, even with the false positives, can help developers with merging code by giv-

ing some pointers to possible problems. The results presented in this report give the client confidence that, with the right investment, a production-ready tool can be produced to help developers perform merges. This tool would be used by developers performing merges, possibly running as part of a continuous integration (CI) pipeline.

9.2 Future work

This section contains a number of recommendations that future work can focus on within the context of source code merge analysis. Three topics are discussed below. The first two topics include research challenges, the last two topics include engineering challenges.

Analysis algorithms First of all, the analysis itself can be improved. The algorithms for detecting problematic changes presented in this study do not cover all identified problems. Future research can focus on detecting more changes or identifying other problematic changes that could be detected. Furthermore, improvements could be made to the algorithms introduced in this report.

For example, algorithm 3.1 currently produces a lot of false positives. This algorithm currently takes too many nodes into account for determining whether a conflict exists. Finding filters or other patterns in the tree to look at might reduce the number of false positives greatly.

Risk assessment Secondly, at this point in time the risk assessment is purely based on the presence of certain problematic changes. Future research could investigate the risk involved with certain types of conflicts by looking at the probability of a conflict resulting in a defect, and the severity of this defect.

Additionally it can be interesting to look at detected problematic changes in their context. A risk assessment could be based on the probability of the change breaking the merge or the impact of the remaining defect after merging. It is easy to imagine that different programming languages require different scoring mechanisms as well. For example, in C one could have created a memory leak with a double `malloc()`, which is not (easily) possible in Java. A survey could reveal common defects and detection for these defects could be included.

Practical applications Finally, the practical usability of the tool can be improved, for example by supporting more programming languages and improving the C parser. These improvements primarily require an engineering effort. A framework like Babelfish [3] can be used for this by using it as internal representation of an AST. This would mean that some data, like

memory dependence, would need to be gathered based on this representation. With Babelfish in place it would be relatively easy to support a large number of programming languages.

For practical use the runtime of the system needs to be improved as well. This can be achieved by improving the current implementation or leveraging parallelism where appropriate, for example with tree differencing or running the analysis algorithms.

References

[1] Dimitar Asenov, Balz Guenat, Peter Müller, and Martin Otth. "Precise Version Control of Trees with Line-Based Version Control Systems". In: *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2017, pp. 152–169.

[2] Adnan Aziz, Tsung-Hsien Lee, and Amit Prakash. *Elements of Programming Interviews in Java: The Insiders' Guide*. CreateSpace Independent Publishing Platform, 2015. ISBN: 9781517435806.

[3] *Babelfish - universal code parser*. source{d}. URL: `https://bblf.sh` (visited on Jan. 19, 2018).

[4] Utpal Banerjee. *Dependence analysis*. Vol. 3. Loop Transformation for Restructuring Compilers. Springer, 1997. ISBN: 9780792398097.

[5] Gabriele Bavota, Bernardino De Carluccio, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Orazio Strollo. "When Does a Refactoring Induce Bugs? An Empirical Study". In: *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*. IEEE, Sept. 2012, pp. 104–113.

[6] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. "Clone detection using abstract syntax trees". In: *Proceedings. International Conference on Software Maintenance, 1998.* IEEE, 1998, pp. 368–377.

[7] Philip Bille. "A survey on tree edit distance and related problems". In: *Theoretical Computer Science* 337.1-3 (June 2005), pp. 217–239.

[8] *clang: a C language family frontend for LLVM*. LLVM. URL: `https://clang.llvm.org/` (visited on Sept. 14, 2017).

[9] Georg Dotzler and Michael Philippsen. "Move-optimized source code tree differencing". In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*. ACM, 2016, pp. 660–671.

[10] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Montperrus. “Fine-grained and accurate source code differencing”. In: *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering - ASE '14*. ACM, 2014, pp. 313–324.

[11] Beat Fluri, Michael Wuersch, Martin Pinzger, and Harald Gall. “Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction”. In: *IEEE Transactions on Software Engineering* 33.11 (Nov. 2007).

[12] Michael W. Godfrey and Lijie Zou. “Using origin analysis to detect merging and splitting of source code entities”. In: *IEEE Transactions on Software Engineering* 31.2 (Feb. 2005), pp. 166–181.

[13] Tim Henderson and Steve Johnson. *Zhang-Sasha: Tree edit distance in Python*. URL: `https://pythonhosted.org/zss/` (visited on Jan. 10, 2018).

[14] James Wayne Hunt and Malcolm Douglas MacIlroy. “An Algorithm for Differential File Comparison”. In: *Computing Science Technical Report No. 41*. Bell Laboratories, June 1976.

[15] Paul Klint, Tijs van der Storm, and Jurgen Vinju. “RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation”. In: *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2009, pp. 168–177.

[16] Chris Lattner. “LLVM and Clang: Next generation compiler technology”. In: *BSDCan - The BSD Conference*. 2008.

[17] Vladimir I. Levenshtein. “Binary Codes Capable of Correcting Deletions, Insertions and Reversals”. In: *Soviet Physics Doklady* 10.8 (Feb. 1966), pp. 707–710.

[18] Jonathan I. Maletic and Michael L. Collard. “Supporting source code difference analysis”. In: *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* IEEE, 2004, pp. 210–219.

[19] Tom Mens. “A state-of-the-art survey on software merging”. In: *IEEE Transactions on Software Engineering* 28.5 (May 2002), pp. 449–462.

[20] John C. Munson and Sebastian G. Elbaum. “Code churn: A measure for estimating the impact of code change”. In: *Proceedings. International Conference on Software Maintenance, 1998.* IEEE, 1998, pp. 24–31.

[21] Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. “Understanding source code evolution using abstract syntax tree matching”. In: *ACM SIGSOFT Software Engineering Notes* 30.4 (July 2005), pp. 1–5.

[22] George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs". In: *Lecture Notes in Computer Science*. Springer. 2002, pp. 213–228.

[23] *OCaml*. OCaml. URL: `https://ocaml.org` (visited on Oct. 3, 2017).

[24] Stefan Otte. "Version control systems". In: (2009).

[25] Mateusz Pawlik and Nikolaus Augsten. "RTED: a robust algorithm for the tree edit distance". In: *Proceedings of the VLDB Endowment* 5.4 (Dec. 2011), pp. 334–345.

[26] Ranjith Purushothaman and Dewayne E. Perry. "Toward understanding the rhetoric of small source code changes". In: *IEEE Transactions on Software Engineering* 31.6 (June 2005), pp. 511–526.

[27] *Rascal Metaprogramming Language*. Centrum Wiskunde & Informatica (CWI). URL: `http://www.rascal-mpl.org/` (visited on Sept. 13, 2017).

[28] Danilo Silva and Marco Tulio Valente. "RefDiff: Detecting Refactorings in Version Histories". In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, May 2017, pp. 269–279.

[29] *The LLVM Compiler Infrastructure*. LLVM. URL: `https://llvm.org/` (visited on Sept. 14, 2017).

[30] Robert A. Wagner and Michael J. Fischer. "The String-to-String Correction Problem". In: *Journal of the ACM (JACM)* 21.1 (Jan. 1974), pp. 168–173.

[31] Peter Weissgerber and Stephan Diehl. "Identifying Refactorings from Source-Code Changes". In: *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. IEEE, 2006, pp. 231–240.

[32] David A. Wheeler. *SLOCCount*. URL: `https://www.dwheeler.com/sloccount/` (visited on Mar. 5, 2018).

[33] David A. Wheeler. *The Apple goto fail vulnerability: lessons learned*. Nov. 23, 2014. URL: `https://www.dwheeler.com/essays/apple-goto-fail.html`.

[34] Kaizhong Zhang and Dennis Shasha. "Simple Fast Algorithms for the Editing Distance between Trees and Related Problems". In: *SIAM Journal on Computing* 18.6 (Dec. 1989), pp. 1245–1262.

Appendix A

Requirements for the proposed tool

The requirements for the proposed software tool are outlined here. These requirements have been partially imposed by ALTEN Netherlands and partially derived from the literature discussed in chapter 2.

The key words *must*, *should* and *may* in the list below are to be interpreted as defined in RFC 2119.

R1 The program must identify problematic changes as described in chapter 3.

R2 The program should assign a score to changes based on the type of problem that might arise.

R3 The program should calculate a total score for a merge based on the changes in it.

R4 The program must present the problematic changes and should present the scores to the user.

R5 The program must support at least the C programming language.

R6 The program may be extensible to use with some other programming languages.

Appendix B

GumTree algorithm

Listed below is the GumTree algorithm [10] in pseudocode for reference. The algorithm listed here is derived from the new Python implementation of the algorithm as used in the prototype.

The algorithm consists of two phases, a top-down phase (algorithm B.1) and a bottom-up phase (algorithm B.2). These phases reference other algorithms that can have different implementations. The algorithm used for calculating the edit distance between two subtrees (i.e. the similarity of two subtrees) is the dice algorithm (algorithm B.3). To find the string similarity of a label, the Zhang-Shasha algorithm [34] is used (algorithm B.4).

```
function TopDown(T1, T2, minHeight)
    L1 := PriorityList(height)[T1]
    L2 := PriorityList(height)[T2]
    A := []
    M := {}
    while min(max(L1), max(L2)) ≥ minHeight do
        if max(L1) > max(L2) then
            for all n ∈ L1 | height(n) = max(L1) do
                L1 := L1 ∪ children(n)
            end for
        else if max(L1) < max(L2) then
            for all n ∈ L2 | height(n) = max(L2) do
                L2 := L2 ∪ children(n)
            end for
        else
            H1 := {n ∈ L1 | height(n) = max(L1)}
            H2 := {n ∈ L1 | height(n) = max(L1)}
            for all (t1, t2) ∈ H1 × H2 | t1 ≅ t2 do
                if (∃tx ∈ T2 | t1 ≅ tx ∧ tx ≠ t2) ∨ (∃tx ∈ T1 | tx ≅ t2 ∧ tx ≠ t1) then
                    A := A + (t1, t2)
                else
                    for all n1, n2 ∈ subtree(t1) × subtree(t2)|n1 ≅ n2 do
                        M := M + (n1, n2)
                    end for
                end if
            end for
            for all t1 ∈ H1 | (t1, tx) ∉ A ∪ M do
                L1 := L1 ∪ children(t1)
            end for
            for all t2 ∈ H2 | (tx, t2) ∉ A ∪ M do
                L2 := L2 ∪ children(t2)
            end for
        end if
    end while
    A := sort(A, (t1, t2) → Dice(parent(t1), parent(t2), M))
    for all (t1, t2) ∈ A do
        if (t1, tx) ∉ M ∧ (tx, t2) ∉ M then
            for all (n1, n2) ∈ subtree(t1) × subtree(t2) do
                M := M + (n1, n2)
            end for
        end if
    end for
    return M
end function
```

Algorithm B.1: Top-down phase of the GumTree algorithm. The ≅ symbol denotes isomorphism.

```
function BottomUp(T1, T2, M, minDice, maxSize)
    for all t1 ∈ T1 | (∀tx ∈ T2 | (t1, tx) ∉ M) do                ▷ In post-order
        if ∃tx ∈ children(t1) ∃ty ∈ T2 | (tx, ty) ∈ M then
            Ct2 := tx ∈ T2 | type(t1) = type(tx) ∧ (∀ty ∈ T1 | (ty, tx) ∉ M)
            if |Ct2| > 0 ∧ max( Dice(t1, tx, M) | tx ∈ Ct2) > minDice then
                t2 := tx ∈ Ct2 | Dice(t1, tx, M) = max( Dice(t1, ty, M) | ty ∈ Ct2)
                M := M + (t1, t2)
                if max(|subtree(t1)|, |subtree(t2)|) < maxSize then
                    R := Opt(t1, t2)
                    for all (ta, tb) ∈ R do
                        if
(∀tx ∈ T1 | (tx, tb) ∉ M) ∧ (∀ty ∈ T2 | (ta, ty) ∉ M) ∧ type(ta) = type(tb) then
                            M := M + (ta, tb)
                        end if
                    end for
                end if
            end if
        end if
    end for
    return M
end function
```

Algorithm B.2: Bottom-up phase of the GumTree algorithm.

```
function Dice(T1, T2, M)
    D1 := subtree(T1) − T1
    D2 := subtree(T2) − T2
    S := {d1 ∈ D1 | (∃d2 ∈ D2 | (d1, d2) ∈ M)}
    return 2 |S| / (|D1| + |D2|)
end function
```

Algorithm B.3: Dice algorithm which calculates the ratio of common descendants of nodes.

```
function Opt(T1, T2)
    C := {}
    for all (t1, t2) ∈ T1 × T2 do
        if ¬(∃tx ∈ T2 | (t1, tx) ∈ C ∧ zss(t1, tx) > zss(t1, t2)) then
            C := C + (t1, t2)
        end if
    end for
    return C
end function
```

Algorithm B.4: Algorithm for finding additional mappings. *zss* is the Zhang-Shasha algorithm that calculates the similarity of subtrees, using the Wagner-Fischer algorithm to compute the Levenshtein distance for determining the similarity between node labels [17, 30, 34].