SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Automatic Generation of Test Cases for Incremental Static Analysis

Jonas August

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Automatic Generation of Test Cases for Incremental Static Analysis

Automatische Generierung von Tests für inkrementelle statische Analyse

Author:	Jonas August
Supervisor:	Prof. Dr. Helmut Seidl
Advisor:	Sarah Tilscher
Submission Date:	08/16/2023

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 08/14/2023 Jonas August

Abstract

Static analysis tools have become important in software development. However, their use can be time-consuming, and therefore the incremental static analysis was proposed, which re-analyzes only the changes between two program versions. The challenge in testing these incremental analysis tools is the test case creation with the generation of code changes in large numbers. Currently, these changes are mainly created manually, resulting in a small number of test cases. This thesis presents a concept and its implementation of how the test case creation process can be automated and streamlined.

The implementation showcases the automated test case creation process by using code mutations and the static analysis tool GOBLINT. The toolchain, named "Test Automation for Incremental Analysis" (TAIA), creates and executes multiple test cases derived from given input programs. The different versions are created using mutators based on the abstract syntax tree and artificial intelligence. We used our implementation TAIA to generate and execute 10,710 test cases derived from 1,219 input programs.

Contents

1. Introduction

1.1. Motivation

Static analysis tools have become important in software development. Such tools enable the developer to verify the source code's safety properties and to identify potential bugs. The analysis should be time-efficient for effective integration into the software development process. One concept for performance improvement is the use of an incremental static analysis. This incremental analysis reuses results from a previous analysis and reanalyzes only changed parts and parts dependent on the changes. This saves the developer time, especially when he makes minor changes to the source code and wants to reanalyze the program quickly. [1]

The static analysis tool Goblint, developed by the Technical University of Munich and the University of Tartu, showcases how such an incremental static analysis can be implemented [1]. To ensure the quality of the incremental analysis of Goblint, it should be well-tested with a large base of test cases. However, up to now, many manual steps are needed to create the test cases. Therefore, only a few test cases are available (around 70 incremental test cases compared to around 1,500 non-incremental test cases).

This thesis showcases how this testing process can be automated and streamlined, from generating test cases with mutations to testing itself. This is done by implementing a toolchain for testing the incremental analysis of Goblint.

1.2. Outline

For readers with no experience with the incremental analysis of Goblint, Chapter 2 gives some background information on how the incremental analysis of Goblint works and how it is currently tested. It leads to a discussion of the current manual workflow in Chapter 3 and explores how this process can be automated using code mutations. For creating these code mutations, it introduces a mutator based on syntactical representation and a mutator based on artificial intelligence.

In Chapter 4, the thesis showcases the implementation of the automated test case creation process named "Test Automation for Incremental Analysis" (TAIA). This toolchain creates and executes test cases based on an input program. The test cases are created using the Clang-Tidy Mutation Generator and the AI Mutation Generator. The chapter studies the design decisions, architecture, detailed implementation, and the encountered issues.

In Chapter 5, an evaluation follows, using the non-incremental regression test of Goblint as input programs. The test results of the generated test cases are examined. Furthermore, we evaluate and compare the mutation generators. Finally, we evaluate the execution time.

Chapter 6 examines related work. This includes showing how mutation testing inspired the mutator based on syntactical representation, aligning used mutation types with real-world bug fixes, and exploring a different approach to generating mutations.

Finally, in Chapter 7, the conclusion will offer an overview of the entire thesis and look ahead to future improvements and potential applications of TAIA.

2. Background

This chapter gives some background on how the incremental analysis of Goblint works and how the incremental analysis of Goblint is currently tested.

2.1. Incremental Analysis of Goblint

The incremental analysis in Goblint consists of a preparation and the incremental analysis itself (Figure 2.1). The starting point for the preparation is a previous input program provided by the user. This program is entirely analyzed by the from-scratch analysis and results in the output of the analysis result and the analysis data, which contains the data needed for an incremental analysis. Then the user adds some changes to the previous input program, which results in the changed input program. Now the incremental analysis can be used. The incremental analysis takes as input the analysis data from the previous from-scratch analysis and the changed input program. The results of the incremental analysis may be less precise compared to a full from-scratch analysis.

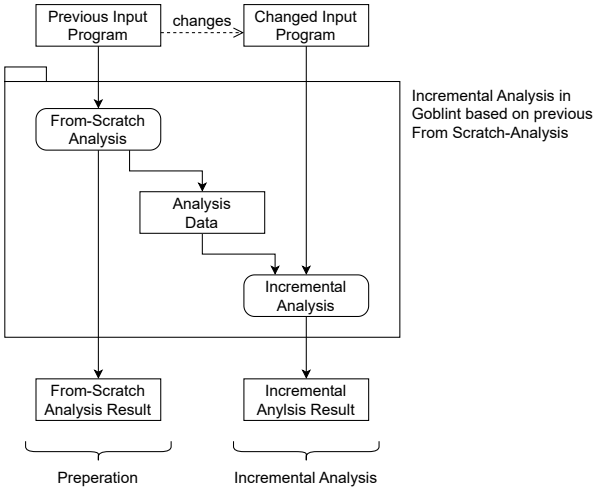

Figure 2.1.: Incremental Analysis in Goblint based on previous From-Scratch Analysis

2.1.1. From-Scratch Analysis

The from-scratch analysis takes an input program that should be analyzed. Goblint uses a top-down solver, which starts at the topmost program point (e.g. the main function). It then

uses a demand-driven exploration to find possible values of the run-time variables at each program point. An option can be activated to save the analysis data. [1]

2.1.2. Incremental Analysis

The incremental analysis is activated by an option, instructing Goblint to load the previously created analysis data. The analysis data contains, among other things, information about the previous analysis results and an intermediate representation of the previous program. In the first step, the changes between the previous program and the input program are detected. The default change detection is based on the abstract syntax tree representation and identifies changes on function level. A change detection based on the control flow graph, which is more fine-grained, can be activated as option. Considering to the detected changes, the already stabilized unknowns are recursively destabilized. Subsequently, these unknowns are resolved again using the demand-driven exploration described in the from-scratch analysis. [1] [2]

2.2. Testing the Incremental Analysis in Goblint

For testing the incremental analysis, the developer currently provides test cases, which can be executed by the Goblint Test Script (Figure 2.2). A test case is stored in a predefined structure to be recognized by the testing script. Each test case consists of three files: an input program with Goblint Checks, a patch file, and a config file. The Goblint Checks are used to encode the expected analysis result. The Test Script starts with prepossessing the test cases and applying the patch file to generate the input programs for the incremental analysis described in Section 2.1. The Test Script runs the incremental analysis and compares the generated warnings for Goblint Checks with the expected result encoded in the check. When one or more of these comparisons are contradicting, the test fails. The test result lists all failed comparisons. When the test result is empty, the test succeeded.

2.2.1. Goblint Checks

For testing the Incremental Analysis, Goblint Checks are used. These checks tell the Test Script what the analysis is supposed to find out. Checks are functions which can be inserted into an input program. The signature of the function is shown in Figure 2.3. The analysis tries to evaluate the expression using its knowledge of the run-time variables. If the expression is true, the check passes. If the expression could not be evaluated because the analysis did not contain enough information for evaluating the expression, the user is warned that the check is unknown. When the expression could be evaluated and is false, the user is warned that the check failed. The results of the checks are printed to the terminal. The example in Figure 2.4 shows how Goblint Checks can be used (terminal output marked with "`TERMINAL`").

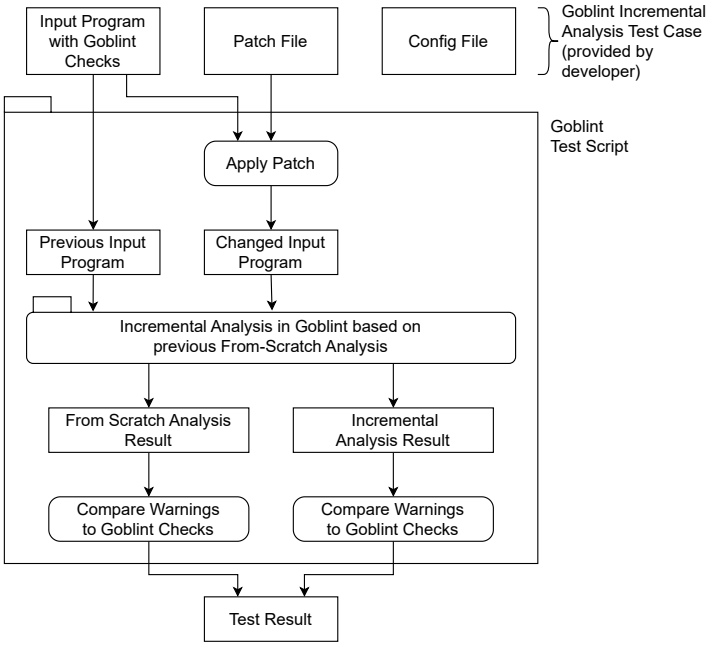

Figure 2.2.: Testing the Incremental Analysis in Goblint

```
void __goblint_check(int expression);
```

Figure 2.3.: Signature of the Goblint Check Function

```
1  #include <goblint.h>
2
3  int main() {
4      int x = 42;
5      int y = 0;
6      int z;
7      /* Current knowledge of the analysis: x=42, y=0, z=? */
8      __goblint_check(x == 42); /* TERMINAL: Check at line 8 succeeded */
9      __goblint_check(y == 42); /* TERMINAL: Check at line 9 failed */
10     __goblint_check(z == 42); /* TERMINAL: Check at line 10 unknown */
11 }
```

Figure 2.4.: Example of Goblint Checks

Goblint Check Annotations

Without an annotation, the Test Script expects the check to succeed. However, one might also want to write tests to verify that the incremental analysis evaluates a check to failing or unknown. Therefore Goblint provides annotations for checks in the form of comments. With these comments, we can annotate for the Test Script which analysis result we expect from a check. Only when the analysis result contradicts the annotated result the test fails. The example in Figure 2.5 shows a test that fails because the analysis result for the last Goblint Check did not match the annotation (test result marked with "`TEST RESULT`").

```
1  #include <goblint.h>
2
3  int main() {
4      int x = 42;
5      int y = 0;
6      int z;
7      __goblint_check(x == 42); // SUCCESS
8      __goblint_check(y == 42); // FAIL
9      __goblint_check(z == 42); // UNKNOWN
10     __goblint_check(x == 42); // FAIL
11     /* TEST RESULT: Test failed at line 10: Expected FAIL, but found SUCCESS */
12 }
```

Figure 2.5.: Example of Goblint Check Annotations

Other annotations

There exist other annotations that encode expected analysis results for data races and deadlocks. However, in this thesis, we want to concentrate on the analysis results encoded with Goblint Checks (success, unknown, or fail). Therefore we will not consider data race and deadlock annotations.

2.2.2. Test Case Structure in Goblint

The Goblint Test Script for the incremental analysis expects the tests to be located in a specified directory in the repository. The tests are organized in a two-level hierarchy. There are groups, which are represented as directories. Each group can contain multiple test cases. A test case consists of three different files. The groups and test cases must follow a naming convention to be correctly recognized by the Goblint Test Script.

Files in a test case

A test case for the incremental analysis consists of three files: An input program (`.c`), a patch file (`.patch`), and a config file (`.json`). From these files, the Test Script can create the input

files for the incremental analysis. The patch files contain all the differences between the previous input program and the changed input program. The patch file can additionally contain changes to the configuration. The config file contains the configuration passed to GOBLINT.

Naming convention

Groups and test cases must follow a naming convention to create a structure that the Goblint Test Script can interpret. The group names and test case names must start with a two-digit numerical id (e.g., 00, 05, 42, 99). This limits the test framework to a maximum of 100 groups with up to 100 test cases each. An exemplary directory structure for a group with two test cases is shown in figure 2.6 [3].

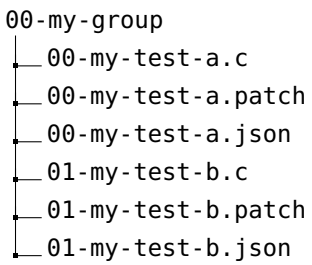

Figure 2.6.: Exemplary Directory Structure of a Test Case Group in Goblint

3. Test Case Creation for the Incremental Analysis of Goblint

Right now, the test case creation for the incremental analysis of Goblint requires many manual steps. This time-consuming manual process is the reason why currently only a few test cases exist for testing the incremental analysis of Goblint. To ensure the quality of the incremental analysis, the number of test cases should be increased. The automation of the test case creation process can create a large number of test cases with little effort for the developer.

3.1. Manual Test Case Creation Process

The manual test case creation process relies on the version control system Git to track the changes for creating the patch file. Figure 3.1 describes the manual test case creation process (creation of the config file is not shown).

- A user program is created as the basis for the test case creation.
- Goblint Checks are added to the user program that are used by the Test Script to verify the results.
- A config file is added, which contains the configuration for Goblint that the Test Script should use.
- The commands "`git add`" and "`git commit`" store the user program and config file in the version control system. The stored user program later becomes the input program.
- Changes are made to the user program, which shell be analyzed with the incremental analysis during the test.
- The Goblint Checks are adopted to the changes.
- The command "`git diff`" creates the patch file between with changes between the previous program and the changed user program. When the config file was changed it is included in the patch.
- The command "`git checkout`" reverts all changes of the user program and config file to the last commit.

An example of creating a simple test case on the terminal can be found in Figure 3.2. For simplicity, the programs are edited with the "`echo`" command and no Goblint Checks are created. [2]

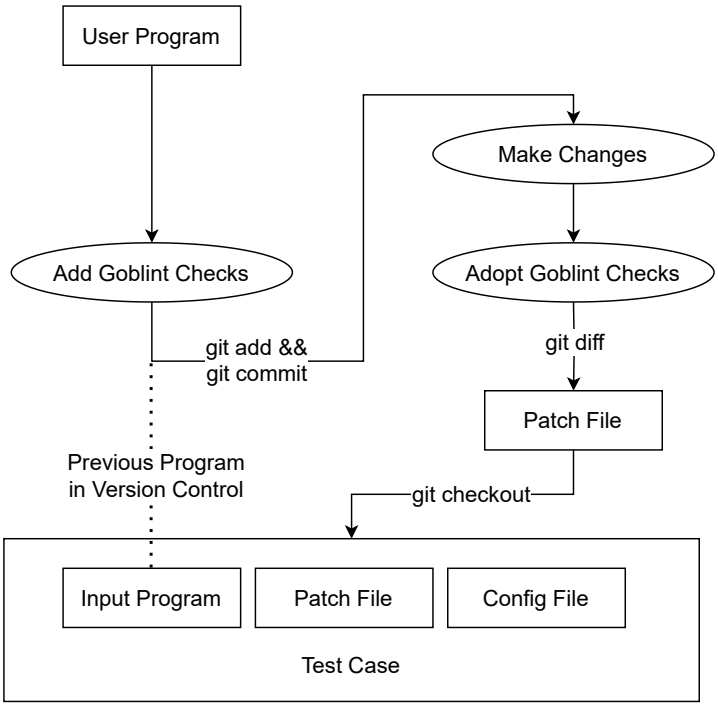

Figure 3.1.: Manual Test Case Creation Process

```
$ echo "int main(){}" > 00-my-name.c
$ echo "{}" > 00-my-name.json
$ git add .
$ git commit -m "Previous program"
$ echo "int main(){int x = 42;}" > 00-my-name.c
$ git diff --no-prefix 00-my-name.c 00-my-name.json > 00-my-name.patch
$ git checkout .
```

Figure 3.2.: Example for the Manual Test Case Creation Process

The manual test case creation process comes with two main challenges for the developer. The one challenge is that working with Git to track the changes requires the developer to execute the Git commands while writing the test case. This interrupts the workflow of the developer. Furthermore, it makes it challenging to work on the previous and changed program simultaneously, as the previous program only exists in the version control system after the commit. The other challenge is that the test case creation process can create only one single test case. For most user programs, there are multiple possibilities to make changes; however, in order to create multiple test cases, the complete test case creation process has to be started from the beginning for each possible change.

3.2. Automated Test Case Creation Process

The automated test case creation process can create multiple test cases based on one single user program. The process is described in Figure 3.3 (creation of the config file was simplified). The first task is to create changes on the provided user program automatically. This is done by the Mutator. The Mutator takes one user program and creates multiple mutated programs. The second task is to analyze the run-time variables of the mutated program to generate the Goblint Checks automatically. The from-scratch analysis of Goblint combined with a transformation is used for this task. The third task is to automatically create test cases from the previous programs and the changed input program by generating the patch file and a structure that the existing Goblint Test Script can execute.

3.2.1. Generating Mutated Programs

The task of the Mutator is to take one user program and to create multiple mutated programs. We assume that the user program is a semantically meaningful program. The mutated programs are created by changing the syntax of the user program. When the syntax changes, this likely results in a semantically less meaningful program. This is why the mutated program should be used as the previous program. We can then interpret the user program as an evolution of the mutated program. The evolution should represent real-world changes.

In this thesis, we explore two types of mutators. One mutator is based on the syntactical representation of the program, and one mutator is based on artificial intelligence.

Mutator based on Syntactical Representation

The Mutator based on the syntactical representation of a program, uses rules to match and transform patterns in the syntax. The most common representation of the syntax of a program is an abstract syntax tree. The abstract syntax tree transforms the program into nodes representing grammatical constructs of the programming language. The edges represent the relationships between the nodes. The root represents the entire program.

With this representation of the abstract syntax tree, mutations can be generated. First, interesting syntax patterns are matched. For each matched pattern, a mutated program is

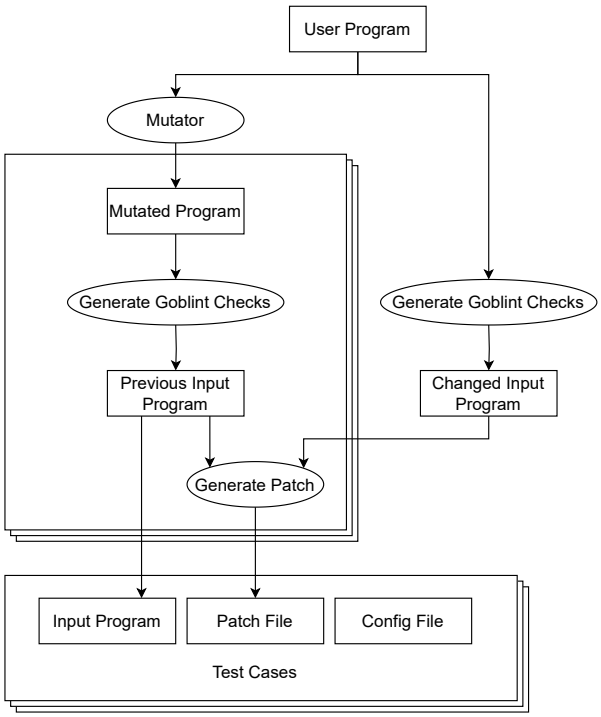

Figure 3.3.: Automated Test Case Creation Process

generated. The mutation is generated by replacing the source code belonging to the nodes of the matched pattern.

The following example shows how a mutation based on the abstract syntax tree works (for simplicity, we only consider void functions and int variables). Figure 3.4 shows the exemplary user program we will use. The abstract syntax tree representation of the user program is visualized in Figure 3.5. The exemplary Mutator matches function declarations. In the example, two function declaration nodes were matched. For each matched node, one mutated program is created by replacing the source code belonging to the body with an empty string. The interpretation of this mutation would be that a function stub was not implemented in the mutated program. Then the function was implemented in the user program.

Mutator based on Artificial Intelligence

Another approach to generate mutations that might have occurred during the development process and lead to the user program is to use artificial intelligence (AI). When we pass the user program to the AI, it can use the program as context for generating the mutations tailored precisely to the user program. We can query the AI multiple times, if we want to create multiple mutations. The challenge in using an AI is writing a good prompt describing the task.

```
1  int x = 0;
2  int y = 0;
3
4  void setX(int arg) {       // Match for mutation 1
5      x = arg;               // Body removed in mutation 1
6  }
7
8  void setY(int arg) {       // Match for mutation 2
9      y = arg;               // Body removed in Mutation 2
10 }
```

Figure 3.4.: Exemplary User Program used for the Example of a Mutator based on the Abstract Syntax Tree

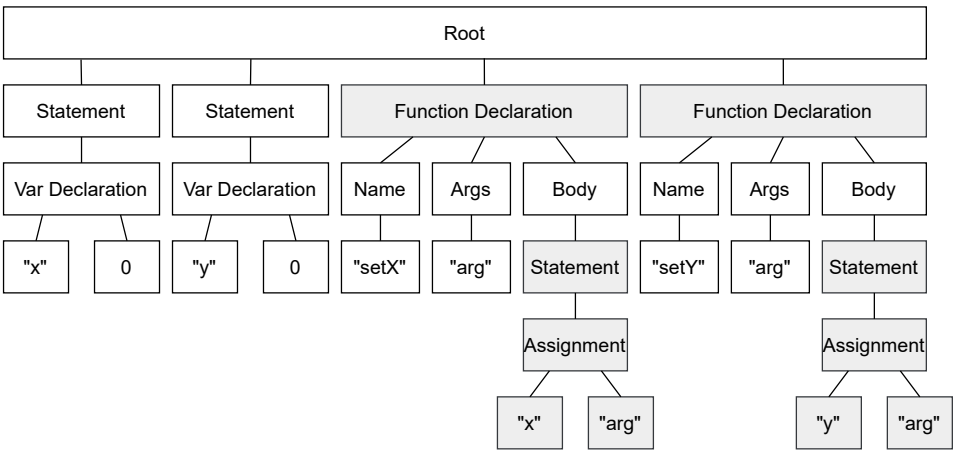

Figure 3.5.: Example of a Mutator based on the Abstract Syntax Tree

3.2.2. Generating Goblint Checks

For creating Goblint Checks, we first need information about the program. This requires a complete analysis of the states of the run-time variables. Based on this analysis Goblint Checks can be inserted.

Goblint provides such a complete analysis and a transformation that can be easily adapted to generate Goblint Checks. This approach requires the assumption that the from-scratch analysis of Goblint is already well-tested and correct.

The from-scratch analysis identifies the states of the run-time variables at each program point. The transformation then writes Goblint Checks into the program after each assignment and each joint point. Joint points are points in the control flow where two paths are joined (e.g., after an if-else statement). The example in Figure 3.6 visualizes the control flow of a simple program. The knowledge of the analysis is noted at each transition. For each value that is not unknown, a Goblint Check would be added to the program.

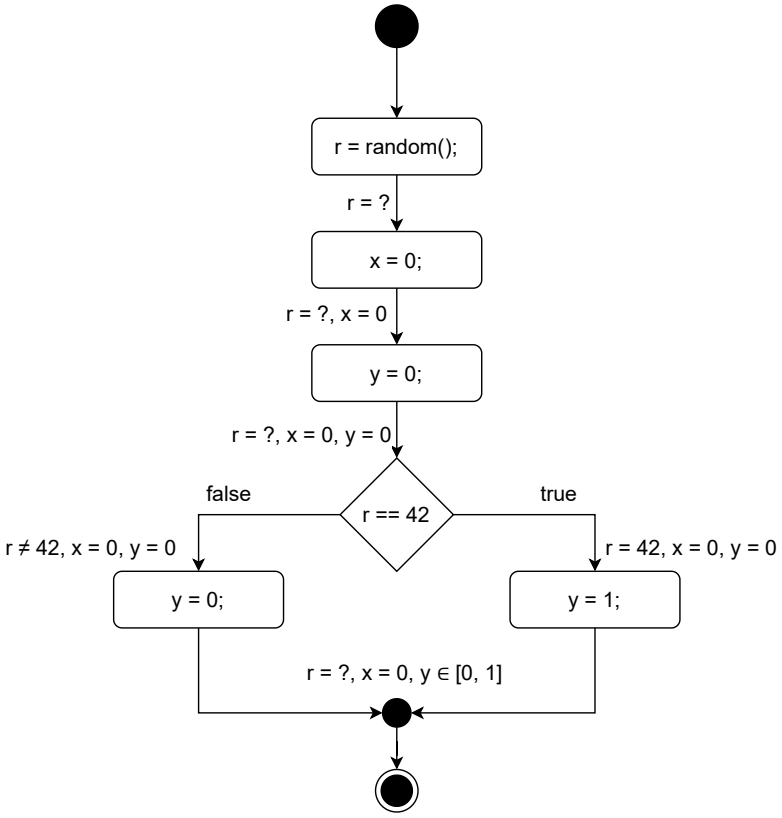

Figure 3.6.: Example of Generation of Goblint Checks

3.2.3. Creating multiple Test Cases

In the manual test case creation process, the version control system Git tracks the changes. In the automated test case creation process, we keep track of the changes by storing the different

mutated programs in separate files. After the generation of the Goblint Checks, there are multiple previous input programs, and one changed input program. For each previous input program, one test case is created. The input program is directly used as the input program of the test case. The patch file is created by comparing the previous input program and the changed input program.

Each test case also includes a config file, which is used to pass the Goblint configuration to the tester automatically. For simplicity we assume in the automated test case creation process that the configuration does not change. The user can provide a config file together with the user program, which is automatically copied to all test cases. When no config file is provided, an empty config file is created.

4. Implementation of the Test Automation for Incremental Analysis (TAIA)

The implementation showcases the automated test case creation for the incremental analysis of Goblint. It is named "Test Automation for Incremental Analysis" (TAIA). The implementation generates test cases using mutations from a given C program file. The mutator based on syntactical representation utilizes custom Clang-Tidy Checks. The Mutator based on artificial intelligence uses the GPT API from OpenAI. The test cases can be either exported or directly run by the Goblint Test Script.

4.1. Design Decisions

4.1.1. Python as preferred Language

We use Python as our preferred language. The advantages are that it is easy to understand and thus easy to maintain. This is important as we want to make extending TAIA with new mutators easy for other developers. As entry points to TAIA, we provide shell scripts for easy execution on the terminal.

4.1.2. Mutator based on Syntactical Representation

When searching for tools that modify the source code based on the abstract syntax tree, linters are a good choice. One of the popular open-source linters is Clang-Tidy. It's purpose is to fix typical programming errors or bugs deduced via static analysis. Clang-Tidy provides an extensible framework with a convenient interface for writing new checks. [4] This is very appealing as the tool can match nodes of the abstract syntax tree and can create modifications of the source code. Furthermore, it promotes easy extensibility. Therefore we use Clang-Tidy to generate our mutations based on the syntactical representation of the programs.

4.1.3. Mutator based on Artificial Intelligence

Generative AI can understand and modify code and therefore can be used as Mutator. We selected for our implementation the GPT API from OpenAI, which is the most popular company providing generative AI. It provides the GPT-3.5 and GTP-4 models, which are capable of understanding and generating code. The company provides an API, including a Python Module. [5]

The models are limited by the number of maximum requests per minute (RPM) and the maximum tokens per minute (TPM). The models differ in the maximum context size and

the input and output pricing. We use the cheapest model GPT-3.5 with a maximum context of 4,000 tokens to minimize cost. The RPM limit should not be relevant for our use case, as we can make around 58 requests per second. The different models are compared in Table 4.1. [5][6]

Table 4.1.: Comparison of the OpenAI GPT models

Model	Max. Context in Tokens	Requests per min.	Tokens per min.	Pricing Input per 1K Tokens	Pricing Output per 1K Tokens
GPT-4	32 000	3 500	90 000	0.06$	0.12$
GPT-4	8 000	3 500	90 000	0.03$	0.06$
GPT-3.5	16 000	3 500	90 000	0.003$	0.004$
GPT-3.5	4 000	3 500	90 000	0.0015$	0.002$

4.1.4. Passing of Goblint Parameters

The incremental test cases support passing Goblint Parameters with the config file. The non-incremental regression test takes a different approach. The test files can contain a single line comment marked with "PARAM" in the first line. The parameters contained in the comment are appended to the calls of Goblint during testing. As we would like to use the non-incremental regression tests as input for TAIA without losing the information about the parameters, we adopted the tester to use for the incremental tests the config file, as well as the first line parameter comment of the input files.

4.2. Architecture

The architecture of TAIA is based on the layered architecture pattern (Figure 4.1). The top layers provide two starting points for execution. The first starting point is the Executor that takes a single C file as input, checks for dependencies, and starts the Main Script. The second starting point is the Batch Processor, which takes a directory as input and uses all C files in the directory as input files and executes them as a batch. The Main Script is responsible for providing the command line interface and starting the Program Generator, the Test Case Generator, and the Test Case Executor.

The temporary directory is used for the communication between the Program Generator, the Test Case Generator, and the Test Case Executor. It contains also the generated programs and test cases.

The Program Generator pre-processes the input file, starts the two mutators, generates the Goblint Checks, and adds the annotations to the Goblint Checks. All programs are checked for correct compilation with gcc. Before the script terminates, the collected metadata is exported into the Meta Data File.

The Clang-Tidy Mutation Generator calls the Clang-Tidy Executable to generate mutations of the input program. The Clang-Tidy Checks specify the desired mutations.

The AI Mutation Generator uses the Python Module `openai` provided by OpenAI to use the GPT API. The Mutator prompts the AI to generate a mutation of the input program.

When all mutations are generated, the Goblint Check Generator calls the Goblint executable to generate the Goblint Checks using the transformation described in section 3.2.1. The Goblint Check Annotator adds the corresponding annotations to the generated Goblint checks.

With all programs generated, the Test Case Generator generates the files required for the test cases. For each mutated program, one test case is created. The Test Case Executor copies the generated test cases to the incremental tests directory used by the Goblint Test Script. Then the Goblint Test Script is executed.

There are three support scripts: The Meta Data Manager, the Statistics Generator, and the Utility Script. The Meta Data Manager stores information about the mutations, exceptions, test results, and performance. The Statistics Generator merges meta data files generated during the batch processing and calculates statistics. The Utility Script contains functions and constants used by multiple scripts.

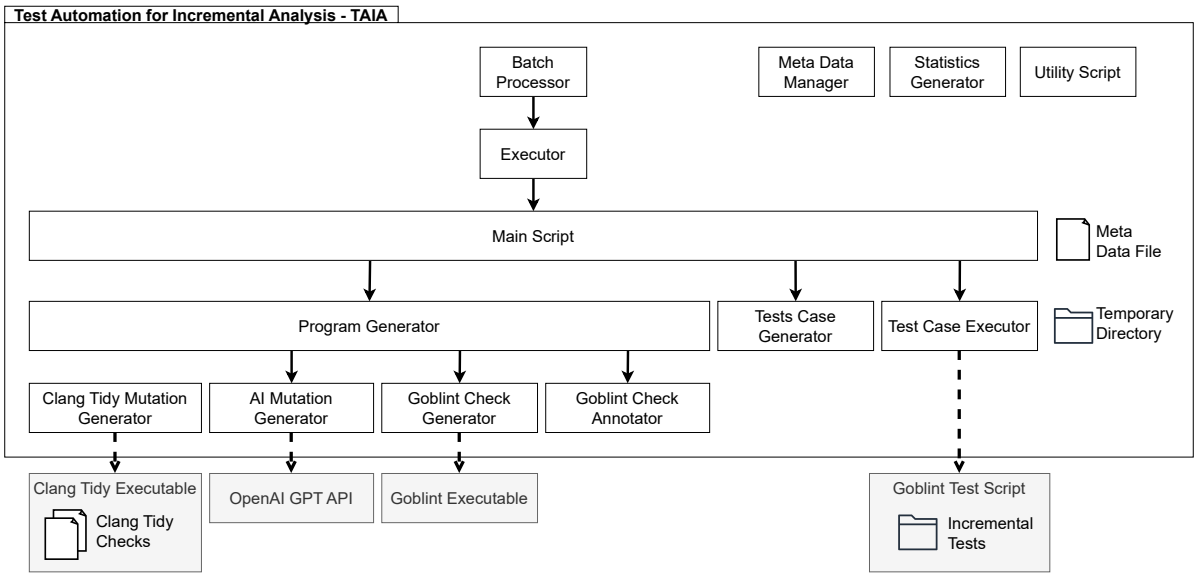

Figure 4.1.: Architecture of TAIA

4.3. Detailed Implementation of TAIA

In this section, we detail the parts of the architecture involved in generating the test cases. The other parts of the architecture are described shortly for completeness.

4.3.1. Batch Processor

The Batch Processor takes all C files inside a user-defined directory as input files. For each input file, the Executor is called. To prevent excessive execution time, each run has a timeout of ten minutes.

Input files in the given directory can be skipped by defining an ignore file. Ignore files contain paths that should be ignored. The ignore file shown in Figure 4.2 would ignore all files, with an absolute path containing "`analyzer/tests/regression/00-sanity/`" or an absolute path ending with "`analyzer/tests/regression/01-cpa/34-def-exc.c`".

After all files in the batch are executed, a summary is printed, categorizing the files: successful tests, failed tests, not compiling input files, not compiling input files after generation of Goblint Checks, ignored files, timed out files, and files with exceptions. Additional statistics can be collected and printed.

```
1 analyzer/tests/regression/00-sanity/*
2 analyzer/tests/regression/01-cpa/34-def-exc.c
```

Figure 4.2.: Exemplary Ignore File

4.3.2. Executor

The Executor provides the starting point of executing TAIA on one single C input file. The Executor checks that all required Python Modules, RUBY, GCC, and CURL are installed. Then the Main Script is called.

4.3.3. Main Script

The main script first parses the command line arguments for TAIA provided by the user. All configurations that were not specified by the command line arguments are interactively asked from the user by showing questions and checkboxes on the terminal. The most relevant configurations are: should the Clang Tidy Mutation Generator be used, should the AI Mutation Generator be used, the selection of mutation types for the Clang Tidy Mutation Generator, the parameters for the AI Mutation Generator, and whether the test cases should test for correctness or precision of the incremental analysis.

When the configuration of TAIA is specified, the Main Script starts the Program Generator, the Test Case Generator, and the Test Case Executor.

4.3.4. Program Generator

The Program Generator pre-processes the input file, calls the Mutators, calls the Goblint Check Generator, calls the Goblint Check Annotator, and checks if the generated files compile with GCC. The pseudo-code shown in Figure 4.3 helps to understand the Program Generator. The called scripts communicate via files inside the temporary directory. The Program Generator

creates a new temporary directory and copies the input file into the temporary directory. The input program receives the index 0. The next step is pre-processing the input program, removing existing Goblint Checks, adding needed includes, and sanitizing the parameter string. The mutators are called to generate mutations of the input program that are stored in the temporary directory. Each mutator returns the new maximum index. After generating the mutations, the Program Generator checks for each program if it compiles with GCC. Then the Goblint Check generator is called. Then again, the program is checked for correct compilation with GCC. Finally, the Goblint Check Annotator is called.

```
1  void generatePrograms(Configuration config) {
2      Path tmpDir = createTempDirectory();
3      Path inputProgram = getProgramPath(tmpDir, 0);
4      copy(config.inputFile, inputProgram);
5
6      preProcessInputProgram(inputProgram);
7
8      int maxIndex = 0;
9      if (config.enableClang)
10         maxIndex = clangTidyMutationGenerator(config, tmpDir, maxIndex);
11     if (config.enableAi)
12         maxIndex = aiMuationGenerator(config, tmpDir, maxIndex);
13
14     for(int i = 0; i <= maxIndex; i++) {
15         Path programPath = getProgramPath(tmpDir, i);
16         checkWithGcc(programPath);
17         goblintCheckGenerator(programPath);
18         checkWithGcc(programPath);
19         goblintCheckAnnotator(config, isUserProgram, programPath);
20     }
21 }
```

Figure 4.3.: Pseudo Code of the Program Generator

Input Program Pre-Processing

The pre-processing of the input program removes the existing Goblint Checks. This is required due to the mutations of the program that change the source code and, therefore, could result in failing Goblint Checks. We automatically generate new Goblint Checks after the mutation of the program.

As preparation for the Goblint Check generation, we want to add, when not existent, the include of `goblint.h`, which is the header file for the Goblint Checks. This ensures that the linking after generating the Goblint checks works correctly.

Lastly, we sanitize the first line parameter comment of the input file because some parameters are incompatible with the incremental analysis. The parameters that are changed during

the sanitizing are listed with a short explanation in the Appendix A.1.

Checking for Compilation

All processed programs are checked for correct compilation, as only correct programs should be considered. The compilation is checked for the input program (index 0) and the mutated programs. The Goblint Check Annotator might create a program that does not compile. Therefore the compilation is also checked after the Goblint Check Generator call.

If the input program does not compile, no test cases can be generated at all, and the program execution is stopped. If a mutated program does not compile, no test case can be generated. In this case, the program is skipped, and the Program Generator continues with the next mutated program.

All default header files included by the Goblint Test Script are included when calling GCC. Furthermore, the temporary directory is included, where the user can place header files.

4.3.5. Clang-Tidy Mutation Generator

The Clang-Tidy Mutation Generator uses CLANG-TIDY to apply mutations to the input program. Clang-Tidy Checks define transformations that can be applied when running CLANG-TIDY. There are two modes in which CLANG-TIDY can be run. The default mode prints on the terminal all transformations that could be performed by the activated Clang-Tidy Check. The fix-mode additionally applies the possible transformations. A line filter can be specified to select which possible transformations should be applied.

The example in Figure 4.4 shows the terminal output of running CLANG-TIDY in the default mode with a Clang-Tidy Check "`unary-operator-inversion`" that transforms if statements by inverting them (the example is simplified). The tilde characters indicate the range of the source code that would be transformed. Underneath the tilde characters, the replacement is shown. Here the expression of the if statement is packed into brackets, and the unary operator for inverting the expression is pre-pended. When applying the transformation in the fix-mode, we would add the option "`fix`". To apply only the second transformation, a line filter "`line-filter=[11]`" could be specified.

```
$ clang−tidy −check=unary−operator−inversion input.c
input.c:7:9: check: unary−operator−inversion can be applied
    7 |     if (a < b) {
      |         ^~~~~
      |         !(a < b)
input.c:11:9: check: unary−operator−inversion can be applied
   11 |     if (c <= d) {
      |         ^~~~~~
      |         !(c <= d)
```

Figure 4.4.: Exemplary Clang-Tidy Terminal Output

The Seven implemented Checks

We created seven different Clang-Tidy Checks that resemble real-world program changes. The checks try to resemble either an evolution of the program or the fixing of a bug. The Table 4.2 lists the checks and gives a short explanation of what they try to resemble. The transformed program is used during the incremental analysis as the previous input program, and the user program as the changed input program.

Table 4.2.: The Seven Clang-Tidy Checks (Mutation Types)

Abbreviation	Name	Resembles
RFB	Remove Function Body	Implement a function
RT	Remove Thread	Implementation of parallelism
CR	Constant Replacement	Increase number of threads
RIS	Remove If Statement	Adding missing error handling
UOI	Unary Operator Inversion	Fixing boolean logic bug
LCR	Logical Connector Replacement	Fixing boolean logic bug
ROR	Relational Operator Replacement	Fixing off-by-one error

The following checks have been added to Clang-Tidy. For each check, we provide an example, which shows the input program and the transformed program (marked with the `TRANSFORMED` comment).

- **Remove Function Body (RFB)**
  This mutation type resembles the implementation of a function stub. The check removes the body of a function. The check must consider the function's return type to create a generic return statement. We use five different generic returns for the different types: "`return`" for void; "`return 0`" for pointer, null-pointer, integral, and char; "`return 0.0`" for float; "`return (struct name){}`" for struct; and "`return (union name){}`" for union. An example can be found in Figure 4.5.

- **Remove Thread (RT)**
  This mutation type resembles the implementation of a parallel program that was previously sequential. The check replaces POSIX thread creation calls with a direct call of the thread function. The difficulty with this check is that the "`pthread_create()`" call returns an integer value. This value is used to check whether the thread was created successfully. However, when we replace the "`pthread_create`()" call directly with the thread function, the return types might not match. Therefore we must split the removal of threads into a two-step process. The first step is to call the Clang-Tidy Check, Function Wrapper (FW), which wraps the thread function "`thread_function()`" into a wrapping function "`thread_function_wrapper()`", which returns zero. For this purpose a newly introduced Clang-Tidy Check Function Wrapper (FW) has been added to Clang-Tidy. The second step is to call the Clang-Tidy Check Remove Thread (RT),

which replaces the POSIX call with the wrapped function name. An example can be found in Figure 4.6.

- **Constant Replacement (CR)**
  This mutation type resembles the increase of the number of threads. The idea is that constants define how many threads are created inside a loop. The transformation would reduce the number of threads to one. The check sets all constants unequal zero to one. When creating the check, we must consider that macros are parsed to multiple constants. An example can be found in Figure 4.7.

- **Remove If Statement (RIS)**
  This mutation type resembles the adding of error handling. The idea is that if statements without an else branch are often used for error handling. For example "`if (exception) exit(-1);`". The check removes complete if statements that do not have an else branch and replaces them with an empty statement. An example can be found in Figure 4.8.

- **Unary Operator Inversion (UOI)**
  This mutation type resembles the fixing of a boolean logic bug. The idea is that an inverted if statement resembles a bug. The check inverts if statements by using the unary operator. An example can be found in Figure 4.9.

- **Locgical Connector Replacement (LCR)**
  This mutation type resembles the fixing of a boolean logic bug. The check switches the boolean operators || and &&. An example can be found in Figure 4.10.

- **Relational Operator Replacement (ROR)**
  This mutation type resembles the fixing of an off-by-one error. The check switches the relational operators <=, < and >=, >. An example can be found in Figure 4.11.

```
int sum(int a, int b) {
    return a + b;
}

// TRANSFORMED:
int sum(int a, int b) { return 0;}
```

Figure 4.5.: Example for the Remove Function Body (RFB) Clang-Tidy Check

```
void *threadFunction(void *arg) {
    printf("Hello␣World\n");
}

int main() {
    result = pthread_create(&thread, &attr, threadFunction, NULL);
}

// TRANSFORMED:
void *threadFunction(void *arg) {
    printf("Hello␣World\n");
}
int threadFunction_wrap(void *arg) {        // 1) FUNCTION WRAPPER – FW
    threadFunction(arg);                    // 1) FUNCTION WRAPPER – FW
    return 0;                               // 1) FUNCTION WRAPPER – FW
}                                           // 1) FUNCTION WRAPPER – FW

int main() {
    result = threadFunction_wrap(NULL);   // 2) REMOVE THREAD – RT
}
```

Figure 4.6.: Example for the Remove Thread (RT) Clang-Tidy Check

```
#define MY_MACRO_5 5
#define MY_MACRO_0 0
int a = 42;
int b = 0;
int d = MY_MACRO_5;
int e = MY_MACRO_0;

// TRANSFORMED:
#define MY_MACRO_5 5
#define MY_MACRO_0 0
int a = 1; /* Replaced Constant 42 */
int b = 0;
int d = 1; /* Replaced Constant 5 */
int e = MY_MACRO_0;
```

Figure 4.7.: Example for the Constant Replacement (CR) Clang-Tidy Check

```
if (error) {
    exit(−1);
}

// TRANSFORMED:
;
```

Figure 4.8.: Example for the Remove If Statement (RIS) Clang-Tidy Check

```
if (a < b) {
    printf("a␣is␣less␣than␣b\n");
}

// TRANSFORMED:
if (!(a < b)) {
    printf("a␣is␣less␣than␣b\n");
}
```

Figure 4.9.: Example for the Unary Operator Inversion (UOI) Clang-Tidy Check

```
if (a && b) {
    printf("Both␣a␣and␣b␣are␣non−zero\n");
}

// TRANSFORMED:
if (a || b) {
    printf("Both␣a␣and␣b␣are␣non−zero\n");
}
```

Figure 4.10.: Example for the Locgical Connector Replacement (LCR) Clang-Tidy Check

```
if (a <= b) {
    printf("a␣is␣less␣than␣or␣equal␣to␣b\n");
}

// TRANSFORMED:
if (a < b) {
    printf("a␣is␣less␣than␣or␣equal␣to␣b\n");
}
```

Figure 4.11.: Example for the Relational Operator Replacement (ROR) Clang-Tidy Check

Automated Clang-Tidy Check Application

The task of the Clang-Tidy Mutation Generator is to automatically create mutated programs of the input program by applying the seven checks we created. The pseudo-code in Figure 4.12 helps to understand how the Clang-Tidy Mutation Generator automates the mutation generation. We create all possible mutations for all seven checks.

The first step is to call the Clang-Tidy Check in the default mode. The Clang-Tidy Mutation Generator analyzes the terminal output of the Clang-Tidy Check call. With the terminal output, we determine the transformations that could be applied. Transformations can be separated into transformation groups, which contain the lines of the transformations that could be applied. Typically a transformation group consists of only one single transformation. The only exemption is the Constant Replacement Check, which groups all usages of a macro into one transformation group. To identify the transformation group for macros we print together with the Constant Replacement Check the macro names to the terminal to identify the lines for the different transformation groups.

Then we iterate over each transformation group. We create a new copy of the input program with the index i. Then we call the check with the fix-mode and the line group as a line filter to apply the selected transformation group to the copied program. The only special case is when we process the Remove Thread check. Here we first need to retrieve the thread function name used for the tread creation. Therefore we print together with the Remove Thread check the function name to the terminal to identify the function name corresponding to the transformation group. The next step is to call the Function Wrapper check to create the wrapped thread function. The wrapping of the thread function adds four additional lines to the source code. We compensate for that by increasing the line numbers of the transformation group by four.

Now we created all possible transformations of our Clang-Tidy Checks and can return the new maximum index to the Program Generator.

Changes to Clang-Tidy

Apart from the Clang-Tidy Checks that we have added into Clang-Tidy, we needed to adopt some options. The reason is the way the current options handle Clang Compiler Errors. Clang-Tidy differentiates between warnings and Clang Compiler Errors. The results of the checks are warnings, and compiler errors are errors that would appear using the Clang Compiler. However, the Clang Compiler Errors mostly do not affect the ability of Clang-Tidy to apply the transformations. As we already check for correct compilation with gcc, we ignore the Clang Compiler Errors when possible. However, to ignore these errors, we need to implement two additional options in Clang-Tidy.

The first problem is when we run Clang-Tidy in default mode, we can not differentiate between Clang compiler errors and other exceptions. This is because in both cases the return code is identical and unequal zero. Therefore we introduced the option "`quiet-return`", which returns zero when compiler errors are found. It also implicitly activates the option "`quiet`", suppressing some printing we do not require.

```
1  int clangTidyMutationGenerator (Configuration config, Path tmpDir, int maxIndex) {
2      Path userProgram = getProgramPath(tmpDir, 0);
3      for (check : ["RFB", "RIS", "RT", "CR", "UOI", "LCR", "ROR"]) {
4          List<List<Integer>> transformations = callCheckDefaultMode(check, userProgram);
5          for (transformationGroup : transformations) {
6              int i = ++maxIndex;
7              Path newProgram = getProgramPath(tmpDir, i);
8              copy(userProgram, newProgram);
9              if (check.equals("RT")) {       // SPECIAL CASE: Remove Thread − RT
10                 String functionName = getThreadFunctionName(transformationGroup, newProgram);
11                 wrapThreadFunction(functionName, newProgram);
12                 transformationGroup.apply(line −> line + 4);
13             }
14             callCheckFixMode(check, transformationGroup, newProgram);
15         }
16     }
17
18     return maxIndex;
19 }
```

Figure 4.12.: Pseudo Code of the Clang-Tidy Mutation Generator

The second problem is that when we run Clang-Tidy in fix-mode, transformations are applied to warnings and clang compiler errors. However, we only want to apply the transformations for warnings, not for errors, as we want to generate only changes implemented in the Clang-Tidy Checks. Therefore we implemented the option "`fix-warnings`" that only applies the transformations of warnings and ignores Clang Compiler Errors when possible.

4.3.6. AI Mutation Generator

The AI Mutation Generator uses the GPT API from OpenAI to generate mutations of the input program.

The model GPT-3.5 processes text using tokens. One token represents a sequence of characters. The model uses the statistical relationships between the tokens to understand the text. The response is generated by producing a sequence of tokens. A rule of thumb is that one token generally corresponds to four text characters. [7]

To use the GPT API, the user has to create an account with payment information. The user can generate API keys for the GPT API using this account. The AI Mutation Generator uses the Python Module `openai` provided by OpenaAI. The `openai` module is initialized with the API key and can then be used to make requests to the GPT API.

A request consists of the model and the prompt. The prompt should provide the program and the request to generate a mutation. The maximum context limits the size of the request and the response. This is why we pass code snippets rather than the complete source code. To give the user control of the snippet selection, the user can define interesting lines. When

the user provided no interesting lines, all lines are interesting lines.

To select the snippet, we randomly choose a start line from the interesting lines. Together with the snippet length, we can extract the snippet from the source code. Then the prompt containing the snippet is constructed and passed to the request. The result of the request replaces the snippet in the source code with the mutated snippet. The replaced snippet will be marked in the source code with comments at the snippet's start and end.

The pseudo-code in Figure 4.13 helps to understand how the AI Mutation Generator works.

```
1  int AIMutationGenerator(Configuration config, Path tmpDir, int maxIndex) {
2      openai.authenticate(config.apiKey);
3
4      for (config.AIMuationCount) {
5          int i = ++maxIndex;
6          Path userProgram = getProgramPath(tmpDir, 0);
7          Path newProgram = getProgramPath(tmpDir, i);
8          copy(userProgram, newProgram);
9
10         List<Integer> interestingLines = config.interstingLines;
11         if (interestingLines.empty())
12             interstingLines = getAllLines(newProgram);
13         int snippetStart = selectRandomInt(interstingLines);
14         int snippedEnd = snippetStart + config.snippetLength;
15         String snippet = extractSnippet(snippedStart, snippedEnd, newProgram);
16         String prompt = getPromptInstruction(snippet);
17         String result = openai.request("GPT-3.5", prompt);
18         replaceSnippet(snippedStart, snippedEnd, newProgram, result);
19     }
20
21     return maxIndex;
22 }
```

Figure 4.13.: Pseudo Code of the AI Mutation Generator

The Prompt

The prompt instructs the AI model to create a possible previous version of the provided code snippet. We focus on mutations that the Clang Tidy Mutation Generator can not generate. The response should contain the mutated snippet and a short explanation. The full prompt is given at the end of the following explanation.

The first part of the prompt is used to set the context. Then we explain what mutations we already have implemented and do not want. The challenge here is to make clear that the AI should not orientate itself on these mutations but instead try something different. Thirdly we introduce the code snippet and the concrete task for the AI. Fourth, we need to

instruct the AI to generate compiling code and keep the snippet's structure. Otherwise, the AI often adds includes or function definitions cut off by the snippet. This would result in a not compiling program after inserting the snippet into the complete program. Sixth, we want to structure the answer into an explanation and the returned snippet using HTML-like tags. This is important to extract the explanation and the code automatically. Lastly, we append the code snippet.

Here is the full prompt we use for the request:

You are a developer for C, helping me with my following question. I want to understand the typical process of code evolution by looking at how developers make changes over time for testing an incremental analysis of the static C analyzer Goblint.

The following mutations are already generated by me. So please do not generate programs that can be generated by these mutations: removal of function bodies, inversion of if statements, switching <= with < and >= with >, replacing constants unequal 0 with 1, replace pthread calls with function calls, switching && with ||, removal of if statements with no else part. Please do not consider these mutations as examples of how your code changes should look like. Just try to prevent doing things that could be done with these mutations.

Below is a snippet from a C file that represents a part of the finished program. My question is how a previous version of this code could have looked like before some typical code changes have been done by developers. Please generate me such a previous version!

The code you generate should be a self-contained snippet that could directly replace the provided excerpt in the initial, complete program. It should preserve the overall functionality of the program and must not cause any compilation errors when reintegrated into the larger code base. Please consider the dependencies and interactions with other parts of the program when generating the previous version of the code. Your generated code should be able to interact correctly with the rest of the program, just like the initial excerpt does. You do not have to add import statements or function declarations, or closing brackets when these are cut off in the snippet, but when they are in the snippet, you need to add them to preserve the whole program.

Use these keywords ("<EXPLANATION>", "</EXPLANATION>", "<CODE>", "</CODE>") similar to HTML tags to structure your answer. Your answer should have the following structure for identifying the different parts of the response, as it is interpreted by another program: <EXPLANATION> (Response: Explain what you have changed in one or two sentences) </EXPLANATION> <CODE> (Response: the previous version of the code) </CODE>

Here the code snippet: "'c CODE SNIPPET *"'*

4.3.7. Goblint Check Generator

The Goblint Check Generator generates Goblint Checks using the Goblint Executable. Goblint already provides the Transformation Assert, that inserts Verifier Assert functions after the assignment of a variable and at join points about all local variables. The signature of the Verifier Assert function is show in Figure 4.14. The Verifier Asserts are currently used for the Competition on Software Verification [8]. The principle of how the transformation works is described in Section 3.2.2.

To generate Goblint Checks instead of the Verifier Asserts, we adopted the transformation and added a new option. With the new option activated, Goblint Checks are generated instead of Verifier Asserts, by switching the function name and keeping the expression used as argument (Figure 4.14). The pseudo-code in Figure 4.15 helps to understand how the Program Generator uses the Goblint Check Generator.

```
void __VERIFIER_assert(int expression);
void __goblint_check(int expression);
```

Figure 4.14.: Signature of the Verifier Assert and Goblint Check Function

```
1 void goblintCheckGenerator(Path programPath) {
2       callGoblint("--input=" + programPath,
3                   "--activate-transformation-assert",
4                   "--transformation-mode=Goblint-Checks",
5                   "--transformation-output=" + programPath);
6 }
```

Figure 4.15.: Pseudo Code of the Goblint Check Generator

Use of the C Intermediate Language

Goblint translates the program into the C Intermediate Language to insert Goblint Checks. The C Intermediate Language is used for program analysis and transformation. After parsing and type checking, the program is compiled into a simplified subset of C. [9]

Translation to CIL causes the removal of all comments in the source code. Therefore, the first line parameter comment has to be re-added after the transformation.

Limitations

The limitations are that only top-level variables are currently asserted (e.g. not variables inside an array or struct). Access through pointers is not supported. [3]

4.3.8. Goblint Check Annotator

The Goblint Check Annotator adds the annotations to the Goblint Checks. These annotations are required to tell the tester which results are expected from the Goblint Check. We differentiate between the previous input program executed with the from-scratch analysis during testing and the changed input program executed with the incremental analysis during testing.

The different Goblint Check Annotations

The Goblint Checks analyzed with the from-scratch analysis should hold. Therefore we annotate the checks in the previous input program with `SUCCESS`.

The incremental analysis, however, might be less precise. Therefore, two new annotations have been introduced to test the correctness or precision.

When testing for correctness, we want to accept Goblint Checks that are successful or unknown but not failing. For this expected behavior, we added a new annotation `NOFAIL` to Goblint, as no comparable annotation existed.

When testing for precision, we want to accept Goblint Checks that are successful or failing but not unknown. We explicitly accept failing, as a failing Goblint Check indicates that the implementation of the incremental analysis is incorrect, but not that it got less precise. For these precision tests, we added a new annotation `NOTINPRECISE` to Goblint, as no comparable annotation existed.

The Table 4.3 gives an overview of the different annotations and where they are used.

Table 4.3.: Goblint Check Annotations

	Test for Correctness	Test for Precision
Previous Input Program (From-Scratch Ana.)	SUCCESS	SUCCESS
Changed Input Program (Incremental Ana.)	NOFAIL	NOTINPRECISE

Automated Goblint Check Annotation

To automate the Goblint Check annotation, we have to make some case distinctions. The program index differentiates the previous input program and the changed input program. The decision if the correctness annotations or the precision annotations should be generated, is determined by the configuration of TAIA. The pseudo-code in Figure 4.16 helps to understand the Goblint Check Annotator.

```
1  void goblintCheckAnnotator(Configuration config, boolean isUserProgram, Path programPath) {
2      if (isUserProgram) {        // Changed Input Program provided by user
3          if (config.test_for == "CORRECTNESS")
4              annotateGoblintChecks(programPath, "NOFAIL");
5          else
6              annotateGoblintChecks(programPath, "NOTINPRECISE");
7      } else {                    // Previous Input Program generated with mutation
8          annotateGoblintChecks(programPath, "SUCCESS");
9      }
10 }
```

Figure 4.16.: Pseudo Code of the Goblint Check Annotator

4.3.9. Test Case Generator

The task of the Test Case Generator is to bring the generated programs into the test case structure used by the Goblint Test Script. This includes the generation of the files for each test case and the generation of names that follow the naming convention. The detailed test case structure is described in Section 2.2.2. The file structure we generate is visualized in Figure 4.17. The pseudo-code in Figure 4.18 helps to understand the Test Case Generator.

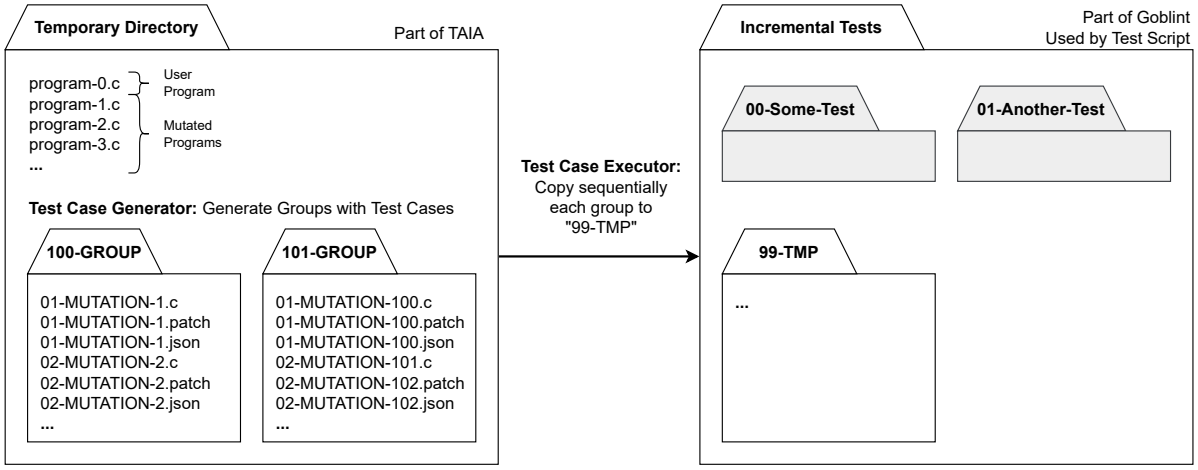

Figure 4.17.: Test Case Generator File Structure

Generating Files in Test Case

The test case consists of the input program, the patch file, and the config file.

The input program is the previous input program generated by the mutators. This previous input program can be copied.

The patch file contains the changes between the input program and the changed input program. We generate the patch file with the command `diff` from the DIFF-UTILS of the Free Software Foundation GNU [10]. The patch contains the path of the files used to create the patch. When we move the test case to the incremental tests directory of the Goblint Test Script, the paths are no longer correct, and the patch can not be applied. Therefore we need to adopt the path as if the test case was generated in the incremental tests directory of the Goblint Test Script.

When the user provides a config file, it is copied it into the test case. When no config file was provided, we create an empty config file.

Naming the files

The name of the test case consists of the test case index and a user-defined string. When generating the test cases, we start with the index 01 to align the mutated program index and the test case index. We increment the test case index for the next test case. To associate

```
List<Path> testCaseGenerator(Configuration config, Path tmpDir, int maxIndex) {
    List<Path> groupDirectories = new List();
    int groupIndex = 100;
    int testCaseIndex = 1;
    for (int i = 1; i <= maxIndex; i++) {
        String groupName = f"{groupIndex}-GROUP}";
        if (directoryNotExists(f"{tmpDir}/{groupName}")) {
            createDirectory(f"{tmpDir}/{groupName}");
            groupDirectories.append(f"{tmpDir}/{groupName}");
        }

        String testCaseName = f"{format(testCaseIndex,'%02d')}-MUTATION-{i}";
        Path testCaseFile = f"{tmpDir}/{groupName}/{testCaseName}";

        Path previousProgramFile = getProgramPath(tmpDir, i);
        Path changedProgramFile = getProgramPath(tmpDir, 0);

        copy(previousProgramFile, testCaseFile + ".c");

        diff(previousProgramFile, changedProgramFile, f"--out={testCaseFile}.patch");
        adoptPathsInsidePatchFile(testCaseFile + ".patch");

        if(config.configPath != null)
            copy(config.configPath, testCaseFile + ".json");
        else
            createEmptyConfig(testCaseFile + ".json");

        testCaseIndex++;
        if(testCaseIndex > 99) {
            testCaseIndex = 0;
            groupIndex++;
        }
    }
    return groupDirectories;
}
```

Figure 4.18.: Pseudo Code of the Test Case Generator

the test case names with the mutated program index *i*, we use the string `MUTATION-i`. The different files of the test case can be identified by the file name endings (`.c`, `.patch`, `.json`).

Naming the groups

The group's name consists of the group index and a user-defined string. With the naming convention, a group can maximally contain one hundred test cases. To circumvent this limitation, we create a new group when no more test cases can be created inside the current group. For the new group, we increment the group index and keep the group's name. The Test Case Executor expects the first group to have the index 100 and the user-defined string `GROUP`. We accept the violation of the naming convention by the index greater 99, as the Test Case Executor renames the group always back to `99-TMP` before test execution. When we adopt the paths in the patch file, we must keep this renaming in mind.

The user can choose to export the test files. Then the user has to provide a group name that follows the naming convention. For exported test files, we increase the group index until it violates the naming convention.

4.3.10. Test Case Executor

The Test Case Executor executes the Goblint Test Script on the generated test case groups. The Test Script expects a group of test cases in the Incremental Tests Directory. The Incremental Tests Directory contains groups of test cases created by different Goblint developers. Therefore we want to reserve only one group for TAIA. We decided to reserve the group `99-TMP`.

The group directories generated by the Test Case Generator are passed as a list. For each generated group, we copy the group's content into the group `99-TMP` inside the Incremental Tests Directory. Then the tester is executed on the group `99-TMP`, and the test return code is stored. When the return code is unequal zero, the test failed. After the test cases in the group are executed, the test case group is removed to make space for the next group. The Test Case Executor returns zero when all tests pass and minus one when a test fails. The pseudo-code in Figure 4.19 helps to understand the Test Case Executor.

4.3.11. Supporting Scripts

Meta Data Manager

The Meta Data Manager is responsible for storing information about the mutations, the exceptions, the test results, and the performance. The information is stored in the YAML format. During the execution of the script, the data is kept in a global variable for fast access. When the execution is stopped, the collected data is stored in the temporary directory as a YAML file.

```
1  int testCaseExecutor(Configuration config, List<Path> groupDirectories) {
2      result = 0;
3      Path incrementalTestsPath = config.GoblintIncrementalTestsPath;
4      for (group : groupDirectories) {
5          copyDirectory(group, f"{incrementalTestsPath}/99-TMP");
6          testReturnCode = startTester("--groupName=99-TMP")
7          if (testReturnCode != 0) result = -1;
8          removeDirectory(f"{incrementalTestsPath}/99-TMP");
9      }
10     return result;
11 }
```

Figure 4.19.: Pseudo Code of the Test Case Executor

Statistics Generator

The Statistics Generator processes the information stored in the metadata files and creates statistics. This is useful for getting detailed insights into the batch processing (e.g. the average performance, the number of mutations, or exception causes). Depending on the batch size, the system generates thousands of metadata files. The Statistics Generator can merge these metadata files into a single statistics file to simplify the user's interaction.

Utility Script

The Utility Script contains functions and constants that are used by multiple scripts. For example, the colors used for printing to the terminal or a function to check the validity of a test group name regarding the naming convention.

4.3.12. Issues

Transformation Assert generates Incorrect Goblint Checks

When generating Goblint Checks with the Transformation Assert provided by Goblint, sometimes incorrect Goblint Checks are created. We created an issue for this on GitHub (link can be found in Appendix A.4.1). It seems to be a bug related to variables renaming during the transformation. When the variables are renamed, the assertions still contain the old variable names. This causes some Goblint Checks to be unknown or fail, while all Goblint Checks generated should be successful. We worked around this bug by analyzing the program after the transformation with Goblint. We use the terminal output to determine the Goblint Checks that are unknown or failing. These Goblint Checks are removed.

Transformation Assert generates Dead Code Goblint Checks

The Transformation Assert provided by Goblint sometimes creates Goblint Checks that are dead code. Dead code is code that can never be executed at run-time. When a Goblint Check

is dead code, the analysis provides no result for the Goblint Check. This causes the tester to fail, as it expected a result for the Goblint Check but received none from the analysis. We worked around this bug by analyzing the program after the transformation with Goblint. We use the terminal output to determine the dead code lines and remove all Goblint Checks that are dead code.

Goblint Test Script uses Imprecise Matching for Annotations and Goblint Checks

A problem with the Goblint Test Script is the way it matches the annotations. The Test Script matches all strings occurrences of the annotations. However, this causes the Goblint Test Script to interpret variables like `myVariableFAIL` as an annotation. Therefore we adopted the Goblint Test Script to only match the annotations after the start of a one-line comment and when they are not part of another word.

Another problem is that the Goblint Test Script matches the `extern` definitions of Goblint Checks as an actual check. However, this is just the definition of the check. Therefore, we adopted the matching in the Goblint Test Script to only match Goblint Checks not preceded by the word `extern`.

5. Evaluation

This chapter evaluates the implementation of the "Test Automation for Incremental Analysis" (TAIA). We look into the usefulness of the regression tests as input files for TAIA, the test results for the correctness and precision test, the generated mutations, and the execution time.

5.1. Setup

The following configuration is used for the evaluation:
CPU: Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz
Size of memory: 15774.5 MiB (total)
OS: Linux 5.19.0-50-generic 64 Bit
Linux-Kernel: Ubuntu 22.04.2 LTS

TAIA was configured to use the Clang-Tidy Mutator, to use the change detection based on the control flow graph, and to create and execute the correctness tests (unless otherwise stated).

5.2. Regression Tests as Input

As input files we used the existing non-incremental regression tests of Goblint that have been created to test the from-scratch analysis of Goblint. These should contain the most relevant cases for testing the from-scratch analysis of Goblint; therefore, we assume that they also cover many of the relevant test cases for the incremental analysis.

Table 5.1 gives an overview of how many files of the regression tests could be used. 1,569 `C` files have been available in the regression test directory (as of 2023-08-03). 350 of these files had to be excluded, as they did not compile or raised exceptions (the detailed breakdown of the excluded files can be found in Appendix A.2). For the following evaluation, we considered only the 1,219 remaining files.

Table 5.1.: Regression Tests as Input

Available C Files	1569	100.00%
Excluded C Files	350	22.31%
Remaining C Files	1219	77.69%

5.3. Test Results for Correctness and Precision

5.3.1. Correctness Test

The execution of the correctness test resulted in 99.98% successful test cases (10,710 total test cases). Two failing test cases were reported as issue on GitHub (link can be found in Appendix A.4.2). Using the change detection based on the abstract syntax tree instead of the change detection based on the control flow graph did not change the test results.

5.3.2. Precision Test

The execution of the precision test resulted in multiple test cases that detected a precision loss. The loss of precision depends on the selected change detection. Table 5.2 shows the number of test cases with precision loss. The share of test cases with precision losses is about two percent.

Table 5.2.: Result of Precision Tests

Change Detection	Test Cases	Precision loss	
Based on Control Flow Graph	10710	234	2.18%
Based on Abstract Syntax Tree	10710	239	2.23%

5.4. Evaluation of the Mutation Generators

5.4.1. Clang-Tidy Mutation Generator

The Clang-Tidy Mutation Generator created 10,980 mutations based on the 1,219 input files. Only 270 mutations resulted in an exception (the breakdown of the exceptions can be found in Appendix A.3). 10,710 test cases have been generated successfully. The Table 5.3 shows the evaluation details by the different mutation types.

29 of the 2741 Remove Function Body mutations created empty patches. These were caused by input files, which already contained a function with an empty body.

The average execution time to generate a mutation was 0.210 seconds. On average 8.79 successful mutations could be generated per input file.

5.4.2. AI Mutation Generator

For evaluating the AI Mutation Generator, we manually analyzed the types of mutations the AI generated. 50 input files were selected randomly to evaluate the AI Mutation Generator. The AI generated one mutation for each input file with a snippet size of 100 lines (11.482 seconds average execution time and 0.08645$ total costs).

Table 5.4 shows the results of the 50 AI mutations. Only 52.00% were executed without an exception (mostly caused by compilation errors). The following listing gives an overview of the mutation schemes that the AI used.

- **Remove existing code**
  remove superfluous code, remove closing of file streams, remove switch statement, remove function call
- **Change existing code**
  change order of mutex locking and unlocking, reverse order of statements, rename variables
- **Add Code**
  adding assertions

The mutations that added assertions are problematic, as some of the created assertions were incorrect and caused the test to fail. Apart from that, most of the other mutations are interesting examples of mutation schemes that could be considered for implementation by the Clang-Tidy Mutation Generator.

Table 5.3.: Evaluation of Clang-Tidy Mutations

Mutation Type	Generated	Exceptions	Success	
Total	10980	270	10710	98.35%
Constant Replacement (CR)	4087	71	4016	98.26%
Remove Function Body (RFB)	2746	5	2741	99.82%
Relational Operator Replacement (ROR)	1278	72	1206	94.37%
Unary Operator Inversion (UOI)	1260	36	1224	97.14%
Remove If Statement (RIS)	798	5	793	99.37%
Remove Thread (RT)	669	79	590	88.19%
Logical Connector Replacement (LCR)	142	2	140	98.59%

Table 5.4.: Evaluation of AI Mutations

Total Requests	50	100.00%
Resulted in Exception	24	48.00%
Remaining	26	52.00%
therein Removed Existing Code	12	24.00%
therein Changed Existing Code	9	18.00%
therein Added New Code	5	10.00%

5.4.3. Comparison of the Mutation Generators

The Clang-Tidy Mutation Generator and the AI Mutation Generator differ clearly in their functionality and their performance:

- **Successful Generated Mutations**
  Nearly all mutations of the Clang-Tidy Mutation Generator can be successfully used by TAIA (98.35%). In contrast, only about half of the mutations of the AI Mutation Generator can be used successfully (52.00%).
- **Execution Time**
  The Clang-Tidy Mutation Generator is much faster (0.210 seconds average execution time) than the AI Mutation Generator (11.482 seconds average execution time).
- **Determinism**
  The mutations generated by the Clang-Tidy Mutation Generator are deterministic and can be easily reproduced. The mutations generated by the AI Mutation Generator are non-deterministic and vary for each request.
- **Costs**
  The Clang-Tidy Mutation Generator is free of charge, while the AI Mutation Generator uses the commercial pay-per-use GPT API. However, the costs for the use of the API are low (on average, 0.001729$ for one request).

Comparing these categories, we conclude that the Clang-Tidy Mutation Generator is best for daily use, as it can quickly generate a large number of mutations for free. The AI Mutation Generator is useful for generating small amounts of mutations that can be used as inspiration for generating new mutation types for the Clang-Tidy Mutation Generator.

5.5. Execution Time

The execution time of the batch of 1,219 input files was 1 hour 51 minutes. The average execution time per input file was 5.5 seconds, and the average execution time per test case was 621.8 milliseconds.

In the following, we examine the correlation between the file size of the input files and the resulting execution time. Figure 5.1 shows the relation between the input file size and the execution time. In general, the execution time increases exponentially with the file size. Regardless of the file size, there are some runaways regarding the execution time in both directions. The figure shows that the regression tests of Goblint contain mainly small files.

Table 5.5 shows the average execution time by sub-functions. In general the execution time increases with the file size. There are some exceptions that can be explained with the runaways. The Test Case Executor and the Goblint Check Generator take up the most time. The execution of GCC and CLANG-TIDY is comparably fast. The Test Case Generator takes only minor time.

In the Goblint Check Generator, we worked around the bugs of the Goblint Transformation Assert by running the analysis a second time (see Section 4.3.12). When the bug is fixed, we could save the second run of GOBLINT, and the execution time of the Goblint Check Generator would be approximately halved.

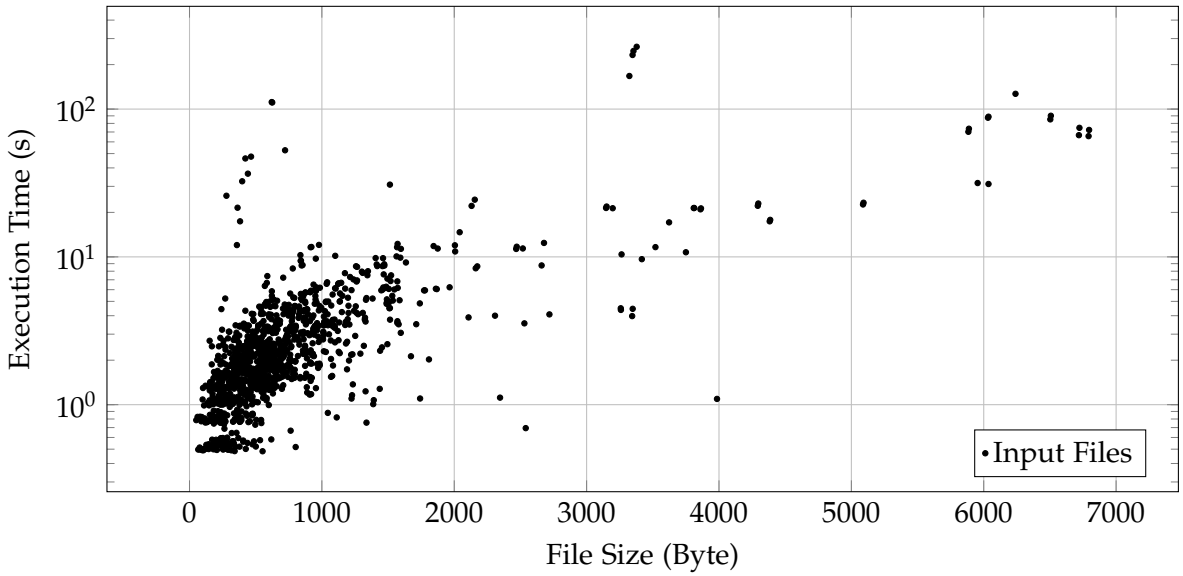

Figure 5.1.: Execution Time based on Input File Size

Table 5.5.: Average Execution Time based on Input File Size

Buckets of File Size (Byte)	≤ 1000	> 1000 ≤ 2000	> 2000 ≤ 3000	> 3000 ≤ 4000	> 4000 ≤ 5000	> 5000 ≤ 6000	> 6000 ≤ 7000
Files in Bucket (Count)	1001	160	18	21	4	5	10
Avg. Exec. Time (ms)	2602	5013	9667	54231	20083	44267	78873
therein Test Execution	1072	2187	4209	30737	8888	19584	34626
therein Check Gener.	1155	2104	4131	20568	8733	19641	35672
therein GCC	192	410	841	2317	1729	3747	6422
therein Clang-Tidy	165	285	4411	525	652	1126	1866
therein Test Gener.	8	18	35	72	72	154	273

6. Related Work

6.1. Mutation Testing as Inspiration

The approach of using mutations based on the syntactical representation was inspired by mutation testing. Mutation testing generates mutations using specified transformations on the abstract syntax tree. The difference between our goal and the goal of mutation testing is that we test for correctness and precision, while mutation testing tests for test coverage. However, the need to create code mutations is the same. [11][12][13][14]

6.2. Comparison with Real-World Bug Fixes

The mutation types that we designed for the Clang-Tidy Mutation Generator focus on bug fixes, as well as on the advancement of the program (e.g. implementation of a function). We used the paper "Mining Frequent Bug-Fix Code Changes" by Osman et al. (2014) to compare our bug fixes with their analysis of real-world bug fixes. According to this paper, the most frequent code changes for bug-fixing can be categorized into missing null checks, missing invocations, wrong naming, and undoing of invocations [15].

We implemented in TAIA one mutation type that resembles the adding of missing error handling, which covers the category "missing null checks" of the paper. Additionally, to the categories described in the paper, we implemented mutation types that resemble the fixing of boolean logic bugs and off-by-one errors. The second category of "missing invocations" could be used as inspiration for a further mutation type that removes function calls.

The other two categories mentioned in the paper were not implemented due to limited relevance. The category of "wrong naming" does not align with our objective since our focus is on mutations changing the semantics, not syntax. The category of "undoing invocations" is limited in its relevance, as it would result in numerous possible mutations, considering that any defined function could be introduced at any point in the program.

6.3. Mutations based on Git Commits

A different approach for generating mutations compared to our implementation would be to use the different commits of a Git repository. Each commit represents one change to the program, and we can use these changes as mutations. This has the advantage that we do not have to generate changes to the program on our own, as we use real-world changes. This approach is used by the static Python type checker Mypy for testing its incremental analysis [16]. In Goblint, a script exists using this approach for bench-marking

the incremental analysis [1]. We did not pursue this approach further, as we wanted to focus on new approaches for automatically generating test cases.

7. Conclusion

This thesis shows that the automation and streamlining of the testing process for incremental analysis is possible. The implementation of the "Test Automation for Incremental Analysis" (TAIA) showcases this successfully for the incremental analysis of Goblint. Compared to the current 72 existing incremental test cases, we could generate additional 10,710 incremental test cases by mutations and thus were able to increase the number of test cases significantly.

Two approaches for mutating programs have been analyzed. The first approach to generate mutations based on the abstract syntax tree representation was showcased in the Clang-Tidy Mutation Generator. This approach was highly suitable as it is quick, deterministic, reliable, and free of charge. The approach can be used to generate large numbers of mutations for the automated test case creation process.

The second approach to generate mutations using artificial intelligence was showcased in the implementation of the AI Mutation Generator. This approach was less suitable as it is slower, non-deterministic, error-prone, and not free of charge. Therefore, the approach can only be used to generate small numbers of mutations as inspiration for generating new mutation types for the Clang-Tidy Mutation Generator.

Due to the modular design of TAIA and Clang-Tidy, further mutation types can be added with minimal effort. This could increase the number of generated test cases further. Inspirations could be found from studies about real-world code changes, as described in Chapter 6.2 and the AI Mutation Generator, as described in Chapter 5.

TAIA is currently focused on the analysis results that can be encoded with Goblint Checks. However, it could also be extended to generate annotations that encode expected data race and deadlock analysis results.

In the future, TAIA could be used to test newly developed features. By running TAIA on the (newly) created non-incremental regression tests, the developer could get first insights into the behavior of the incremental analysis without writing any incremental test cases manually.

A. Appendix

A.1. Sanitizing Goblint Parameters

These parameters are changed during the sanitizing of the first line parameter comment:

- **Witness**
  Witnesses are used to prove witness invariants. There are some options for witnesses that require an input YAML file. However, this witness data can not be used for the incremental analysis. Therefore we will remove the options "`witness.yaml.validate`" and "`witness.yaml.unassume`". The analysis Unassume also depends on these input files. Therefore we remove "`--set ana.activated[+] unassume`".

- **Autotune**
  "Autotune tries to intelligently activate analyses based on the input files" [3]. The problem is that when the activated analysis changes during an incremental run, the incremental analysis can crash. Therefore we remove "`--enable ana.autotune.enabled`".

- **Warnings**
  The Goblint Test Script relies on comparing the warnings printed on the terminal. We need to remove options that disable these warnings. Therefore we remove "`--disable warn.assert`".

- **Transformation Assert**
  The Transformation Assert adds Verifier Asserts. However, these assertions interfere with the Goblint Test Script. Therefore we remove "`--set trans.activated[+] assert`".

- **Analysis with no Support for Marshaling**
  Marshaling is used to deserialize the analysis results to store them for the incremental analysis. Some analyses do not support marshaling and therefore can not be used with the incremental analysis. Therefore we remove "`--set ana.activated[+] apron`" and "`--set ana.activated[+] affeq`".

- **Analysis depending on input files**
  The analysis File analyses correct file handle usage. This analysis can fail if referenced files are not found in the temporary directory. Therefore we remove "`--set ana.activated[+] file`".

- **Analysis Assert**
  We need the Assert analysis to receive the analysis results of the Goblint Checks.

Therefore we activate this analysis at the end of the first line parameter comment in case it was deactivated with a previous parameter: "`--set ana.activated[+] assert`".

- **Solver td3**
  The incremental analysis is only implemented for the solver TD3. Therefore we set the solver at the end of the first line parameter comment to TD3 in case the solver was changed with a previous parameter: "`--set solver td3`".

A.2. Regression Tests as Input (Breakdown of Excluded Files)

The table A.1 shows the detailed breakdown of the input files excluded for the evaluation. Most input files were excluded due to bugs in the Transformation Assert of GOBLINT (link to the GITHUB issues can be found in Appendix A.4.3). The input files with a long jump crash GOBLINT when the auto-tuner activates the long jump analysis during the incremental analysis (link to the GITHUB issue can be found in Appendix A.4.4). Therefore all files testing the long jump analysis are excluded. Input files that did not compile with GCC are excluded. Input files causing a stack overflow in GOBLINT are excluded. Two files intend to crash GOBLINT, which are therefore excluded. One completely empty file is excluded. One input file with invalid C syntax causing a crash is excluded.

Table A.1.: Regression Tests as Input (Breakdown of Excluded Files)

Available C Files	1569	100.00%
Excluded C Files	350	22.31%
therein Transformation Assert Exceptions	252	16.06%
therein Input Files with Longjumps	48	3.06%
therein Input Files with GCC Compilation Errors	44	2.80%
therein Input Files causing a Stack Overflow in Goblint	2	0.13%
therein Input Files intending an Exception in Goblint	2	0.13%
therein Input Files that are empty	1	0.06%
therein Input Files that use invalid C syntax	1	0.06%
Remaining C Files	1219	77.69%

A.3. Evaluation of Clang-Tidy Mutations (Breakdown of Exceptions)

The Table A.2 breaks down the causes for exceptions raised by the mutations. Most exceptions are caused by Clang compiler errors. These prevent, in some cases, the application of fixes. The GCC compilation errors are caused by two mutations. The first is Remove Function Body, which has, in some cases, problems with generating a generic return statement for structs. The second mutation is Constant Replacement, which replaces constants in switch statements. This can result in switch statements with two equivalent cases, which causes a compilation

error. Some mutations trigger the bugs of the Transformation Assert of Goblint (link to the GitHub issues can be found in Appendix A.4.3).

Table A.2.: Evaluation of Clang-Tidy Mutations (Breakdown of Exceptions)

Total Number of Mutations	10980	100.00%
Causing Exception	270	2.46%
therein Clang Compiler Errors preventing Fix	211	1.92%
therein GCC Compiler Errors	40	0.36%
therein Transformation Assert Exceptions	19	0.17%
Remaining Mutations	10710	97.54%

A.4. Links to GitHub Issues

The following subsections contain the links to the GitHub issues referenced in the thesis (as of 2023-08-10).

A.4.1. Transformation Assert generates Incorrect Goblint Checks

The following link leads to the referenced issue:
`https://github.com/goblint/analyzer/issues/1115`

A.4.2. Failing Correctness Tests

The following link leads to the referenced issue:
`https://github.com/goblint/analyzer/issues/1134`

A.4.3. Issues related to Transformation Assert

The following links leads to the referenced issues:
`https://github.com/goblint/analyzer/issues/1132`
`https://github.com/goblint/analyzer/issues/1131`
`https://github.com/goblint/analyzer/issues/1130`
`https://github.com/goblint/analyzer/issues/1129`
`https://github.com/goblint/analyzer/issues/1115`

A.4.4. Auto-Tuner activates Long Jump Analysis during Incremental Analysis

The following link leads to the referenced issue:
`https://github.com/goblint/analyzer/issues/1106`

List of Figures

List of Tables

Bibliography

[1] H. Seidl, J. Erhard, and R. Vogler. "Incremental Abstract Interpretation". In: *From Lambda Calculus to Cybersecurity Through Program Analysis*. Ed. by A. Di Pierro, P. Malacaria, and R. Nagarajan. Lecture Notes in Computer Science. Technische Universität München. Garching, Germany: Springer Cham, 2020, pp. 142–158. DOI: 10.1007/978-3-030-41103-9.

[2] Saan, Simmo and Schwarz, Michael and Erhard, Julian and Tilscher, Sarah and Vogler, Ralf and Apinis, Kalmer and Vojdani, Vesal. *Goblint Documentation*. https://goblint.readthedocs.io/. Accessed: 2023-07-04.

[3] S. Saan, M. Schwarz, J. Erhard, S. Tilscher, R. Vogler, K. Apinis, and V. Vojdani. *Goblint*. Accessed: 2023-07-17. DOI: 10.5281/zenodo.5735006. URL: https://github.com/goblint/analyzer.

[4] The LLVM Project. *Clang-Tidy - A clang-based C++ linter tool*. https://clang.llvm.org/extra/clang-tidy/. Accessed: 2023-07-04.

[5] OpenAI. *OpenAI Platform Documentation*. https://platform.openai.com/docs/introduction. Accessed: 2023-07-13.

[6] OpenAI. *OpenAI Pricing*. https://openai.com/pricing. Accessed: 2023-07-13.

[7] OpenAI. *OpenAI Tokenizer*. https://platform.openai.com/tokenizer. Accessed: 2023-07-13.

[8] Prof. Dr. Dirk Beyer. *Competition on Software Verification*. https://www.sosy-lab.org/. Accessed: 2023-08-10.

[9] Kerneis, Gabriel. *CIL Repository*. https://sourceforge.net/p/cil/code/ci/master/tree/. Accessed: 2023-07-04.

[10] GNU - Free Software Foundation. *GNU Diffutils*. https://www.gnu.org/software/diffutils/manual/diffutils.html. Accessed: 2023-07-25.

[11] T. Fidry. *Mutation Testing: Better Code by Making Bugs*. https://www.youtube.com/watch?v=dlVASJ-MbUE. PHP Developer Days 2018, Lecture. PHP USERGROUP DRESDEN e.V., Apr. 26, 2019.

[12] N. Lohmann. *mutate_cpp Repository*. https://github.com/nlohmann/mutate_cpp. Accessed: 2023-07-04.

[13] T. Fidry. *awesome-mutation-testing Repository*. https://github.com/theofidry/awesome-mutation-testing. Accessed: 2023-07-04.

[14] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman. "Chapter Six - Mutation Testing Advances: An Analysis and Survey". In: *Advances in Computers*. Ed. by A. M. Memon. Vol. 112. Advances in Computers. Elsevier, 2019, pp. 275–378. DOI: 10.1016/bs.adcom.2018.03.015. URL: https://www.sciencedirect.com/science/article/pii/S0065245818300305.

[15] H. Osman, M. Lungu, and O. Nierstrasz. "Mining frequent bug-fix code changes". In: *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. 2014, pp. 343–347. DOI: 10.1109/CSMR-WCRE.2014.6747191.

[16] Python Software Foundation. *MyPy - Incremental Checker Python Script*. https://github.com/python/mypy/blob/5617cdd03d12ff73622c8d4b496979e0377b1675/misc/incremental_checker.py. Accessed: 2023-08-07.