Software-Maintenance: Assignment 1

Univ.-Prof. Dipl.-Ing. Dr.techn. Franz Wotawa, Dipl.-Ing. Oliver A. Tazl,
Alessa Angerschmid, Laura Rössl, Clemens Walluschek
Institute for Software Technology, Graz University of Technology

March 5, 2020

1 Organization

You can achieve a total of 20 points for Assignment 1. The assignment consists of a theoretical (Section 2) and a programming part (Section 3), which count 50% each.

1.1 Group Finding and Registration

For this lecture, you need to build groups of 4 students. If you need members for your group, please use the newsgroup (`tu-graz.lv.soma.gruppensuche`). Register your group using the web interface (you will find a link on the TeachCenter).

1.2 Tutorial

There will be a tutorial for the programming assignment. In this tutorial the framework will be explained. Additionally, we will provide some hints about your programming task (useful data structures, what to test, what to keep in mind etc.). The tutorial takes place on **March 5^{th}, 3:00 pm** at HS i9, Inffeldgasse 13.

1.3 Question Hour

We will also provide a question hour on demand. This means, the question hour will only take place if there are questions in the newsgroup beforehand. We will announce this in the newsgroup, too. If there are enough questions, the question hour takes place on Thursday, **March 19^{th} 2020, 3:00 pm** at HS i9, Inffeldgasse 13, ground floor. The question hour ends when there are no students with questions. Thus, we recommend that you be there at 3:00 pm.

1.4 Framework

You can download a framework from the TeachCenter. It contains a LaTeX template for the theoretical part and a folder with source files for the programming assignment.

1.5 Submission

The submission consists of three parts: GIT repository, issue reports on `GitLab` and a hardcopy. For all three parts the deadline is **Monday, March 23^{rd} 2020, 10 am**. Late submissions will **not** be accepted and unreadable submissions will receive 0 points.

1.5.1 Programming Assignment via GitLab

For the duration of the assignments, you need a `GitLab`[1] account. After the group registration ends a repo will be created for your group.

[1] `https://git-students.ist.tugraz.at/`

During the development of your program you must use `GitLab`'s issue tracking system to report software bugs[2]. Recording issues allows you and your team to maintain a high software quality and capture, report and manage bugs. For our purposes you must define a label named "BUG" (or "bug") and add issue reports whenever you find problems in the current version of your implementation. Adding this label is essential so that we can isolate the bug reports easily. While your group can decide whether they would like to assign bugs, define due dates etc. selecting the correct label is mandatory. Do not forget to close issues, once you have fixed them. You will be asked to show some of the issues you have created at the mandatory interviews. Appendix A contains a basic introduction into bug reporting and how to add them in `GitLab` .

Repository Structure

In order to allow automated testing, the following repository structure is obligatory for the folder `submission1/`:

- `readme.txt` description of your practical assignment
- `Slicer/` folder with the programming assignment
  - `staticslicer_<your group number>.jar`
  - `build.xml` the ant-build file for your project
  - `Java.g4` grammar file which describes the Java language
  - `src/` source code in the provided package-structure
  - `testdata/` data for the test cases
  - `lib/antlr-4.5-complete.jar` ANTLR framework library
  - `doc/` Javadoc files

Do **not** upload any other files or folders to the master branch of our repository!

Testing

Please note that failing the public test cases (public.java, public1.java, public2.java) will lead to zero points for this part of the assignment. Additionally, we use private test cases to measure the performance of your submission. The scoring of the implementation will be made on the basis of the passing of test cases which will cover different parts of the grammar.
For testing your application we use Java 8.
Use the `readme.txt` to describe which features you have implemented as well as missing features.

1.5.2 Hardcopy

Hand in a hardcopy of your solution of the theoretical part (Section 2) in the mail box on the ground floor of Inffeldgasse 16b. For that, we provide the LaTeX-file `assignment1.tex` in the framework which contains a cover sheet that you must use for your submission as well as a template of how we would like to receive your solutions. Hand written documents will be accepted if the hand writing is in a readable form. However, unreadable submissions will receive 0 points! You can either write in German or English.

1.6 Interviews

Interviews will be held on **March** 26^{th} and 27^{th} **2020**. Watch the newsgroup for timeslots. All team members need to be able to explain your theoretical solutions as well as your source code. You should also be able to describe the bugs which you have inserted into the bug database. In addition, you should be prepared to solve small examples by hand.

[2]Of course you may also use the issue tracking system for additional purposes, e.g., backlog. In case you do think about appropriate labels.

1.7 Plagiarism

Discussion of the tasks between the lecture participants is welcome, but plagiarism will lead to a score of 0 points. Do not include any personal information (name, student ID, mail-addresses etc.) in your source code. Your files will be uploaded to an external server in order to cross-check against other submissions.

If you have questions regarding the assignment you should post them to the newsgroup (`tu-graz.lv.soma`).
Good luck!

2 Theoretical Part

2.1 General Notes

- Use directed graphs for CFG, PDG, Dynamic Slicing and Hitting Sets.
- For the dynamic slices you must use the table representation as described in the Lecture Notes.
- You must mark the following dependencies in an easily distinguishable format (different arrow styles and/or different colors), for example:
  - **Control dependency**: ——————>
  - **Data dependency**: - - - - - - ->
  - **Symmetric dependency**: <- . - . - . ->
  - **Potential data dependency**: ========>

  Clearly indicate which format you are using!
- The function calls `read(x)` and `write(x)` mean reading something from the standard input stream and writing something to the standard output stream respectively.
- **Important:** in order to receive full points, please write down all your final results at the end of every task! Examples:
  - `Static Slice for (5, {a}): {1,2,3}`
  - `Minimal Hitting Set: {1,2}, {1,4,5}, {2,6,7}`

2.2 Tasks

1. Static slicing - advanced [*3.5 points*]

   Consider the program below.

   (a) Calculate the static slice for the given slicing criteria using the **table algorithm**. [*2.5 point*]

   (b) Calculate the static slice for the given slicing criteria using the **program dependency graph**. [*1.0 points*]

   i. (20,{a})

   ii. (21,{d})

```
1  begin
2      a = b * 2;
3      c = a + d;
4      while( a < c + 5) do
5          begin
6              a = a + b;
7              d = c - a;
8          end;
9      do;
10     if( a > c ) then
11         begin
12             b = a + 2;
13         end;
14     else
15         begin
16             c = c - 10
17         end;
18     fi;
19     d = a + c
20     write(a);
21     write(d);
22 end;
```

2. Control flow graph [*1.5 points*]

   Draw the **control flow graph** of the following programs:

   (a) Program from task 1 [*0.5 points*]

   (b) Program below [*1.0 points*]

```
1  begin
2      a = m * o;
3      s = m + a;
4      if(o < i) then
5        begin
6            if(x == 0) then
7              begin
8                  x = x + 1;
9                  s = o * 5;
10             end;
11           else
12             begin
13                 x = 10;
14             end;
15           fi;
16       end;
17     else
18       begin
19           o = o + m;
20           while(m <= a) do
21             begin
22                 m = x * s;
23             end;
24           od;
25       end;
26     fi;
27     write(s);
28     write(m);
29 end;
```

   Hint: you will need this in the programming assignment!

3. Minimal hitting sets [*2 points*]

   Create the directed graph and calculate the **minimal hitting sets** for the following conflict sets:

   (a) ({s, o}, {m, a}, {a, s, i, g}, {o, n, e}) [*0.5 points*]

   (b) ({t, u}, {g, r, a, z}, {a, s, u}, {s, g, a}) [*0.75 points*]

   (c) ({c, a}, {b, d, a}, {c, d, e, f}, {c, e, j}) [*0.75 points*]

   You do not have to draw every intermediate step. However, you must hand in the final graph with the minimal hitting sets marked. Furthermore, as described in 2.1 you must write down all final results in order to receive full points!

4. Dynamic slicing [*3.0 points*]

   Calculate the following slices for the program below with the slicing criterion:
   ({y = 2, t = 1}, 29^{20}, {z})

   (a) dynamic slice [*2.0 points*]

   (b) relevant slice [*1.0 points*]

```
 1 begin
 2     z = 20;
 3     x = 5;
 4     read( y );
 5     read( t );
 6     while( y < 5 ) do
 7         begin
 8             y = y * 2;
 9             z = z - (y + x);
10         end;
11     od;
12     if( x > t ) then
13        begin
14           x = t * x;
15           while( z < 0 ) do
16                begin
17                    t = t + 20;
18                    z = t + y;
19                end;
20           od;
21           t = x * 10 + z;
22        end;
23     else
24        begin
25           x = e + z;
26        end;
27     fi;
28     write( x );
29     write( t );
30 end;
```

3 Programming Assignment

3.1 Objectives

The aim of the programming assignment is to take the theories presented in the lecture and produce a practical application. You should implement the acquired knowledge about **static slicing** by implementing a **Java application** that calculates the static slice for a given Java program with a PDG [*10 Points*]. Additionally, you must add at least two issue reports (see Section 1.5.1 and Appendix A) and be able to show them during the interviews. You will not receive points for that, however, we will deduce the points of your programming part by a maximum of 1 point if you do not fulfill this requirement! The programming task counts 50 % of the first assignment.

3.2 General Overview

The framework is able to read a Java program from a file. The filename can be platform independent, absolute as well as relative file paths can be handled. The framework implements the parsing of input files by splitting them into the different statements (have a look at the Java Doc of statement classes for further information). An output class (`DataIO.java`) is implemented to print the lines of the slice in the correct format to the output file (method `printSlice()`).

Here is a list of your tasks and some general hints:

- calculate the CFG of the input program using the list of statements
- find a proper representation for your CFG
- Use the CFG to calculate the PDG
- start to find the program dependencies
- calculate the data dependencies
- calculate the slice using your PDG
- extract the slicing criterion of the command line (args[1])
- write the slice to the standard output using the provided method
- think of an intelligent data structure for a useful internal representation of your graphs

Using the framework

If you want to use **Eclispe** for implementing the programming task take the following steps for creating a new project based on the framework:

- Open the file `.classpath` with any editor, and, according to the `TODOs`, set the two occurances of `path_to_your_project` to the absolute path to your project.
- In Eclipse, choose `File` > `New` > `Java Project from Existing Ant Buildfile`.
- Navigate to the source folder and choose the file `build.xml`.
- Check `Link to buildfile in the file system`.

If you want to use **IntelliJ** take the following steps, respectively:

- On the start page, choose `Import Project` > `Import project from external model`.
- Check `Eclipse`.
- Select the source folder and, on the next page, check the module `slicer` as the Eclipse project to import. The project is now imported.
- In the Project View > right click on the project's name > `Open Module Settings`.
- In the new window, choose `Platform Settings` > `SDKs` > `Classpath`.
- Click `+`, navigate to the source folder and choose `lib/antlr-4.5-complete.jar`.

Compilation, Execution and Testing

Build an executable .jar file `staticslicer_<yourgroupnumber>.jar` from your source code. This can be done using the command `ant jar`. Using IntelliJ, you may also do the following steps to achieve the same:

- Under `File` > `Settings` > `Plugins` check `Ant Support`.
- Under `View` > `Tool Windows` choose `Ant Build`.
- In the new `Ant Build` window, click `+` and add the `build.xml` as a build file.
- Run `jar`.

Your program must be able to be executed from the command line/shell with the command:

```
java -jar staticslicer_<your group number>[.jar] <input-file> <slicing criterion>
```

The slicing criterion is a pair (i, V) where i is a line number in the program and V is the set of variables of interest. The static slice for the slicing criterion should be calculated with the help of the provided statement classes (have a look at the Java Doc of statement classes).
This is how a program call for group no. 1 looks like:
`java -jar staticslicer_1.jar testdata/input/test1.java "(12,{z})"`

You can also directly run the program by editing the `Run configurations` of your project and adding command line arguments (e.g. `testdata/input/test1.java "(12,{z})"`) which may be the fastest way for testing your program.

Input Program Specification

Your slicer should calculate the static slice for Java programs. You can assume that all input programs meet valid Java syntax and that the programs consist of only one file and that they contain a main, but no other methods. This is how an input file can look like:

```
1  class Test1 {
2    public static void main() {
3      int x = 1;
4      int y = 2;
5      int z = 0;
6      if (y > 0){
7        z = 4 + x;
8      }
9      z = z + y;
10   }
11 }
```

Parsing and Slicing

The framework already reads the input file and automatically extracts all information of the program. It then creates a list of different statements including information like referenced and defined variables. This process is done by using the `ANTLR` framework. If you are interested in any details about `ANTLR` please check the `ANTLR` homepage.

The following statements are created:

- variable definitions
- variable declarations
- if-statements
- else-statements
- while-loops

- for loops

Each statement is represented by a corresponding statement class (have a look at the Java Doc of statement classes for detailed information). You do not need to implement object handling, method calls, arrays, lists or other non-primitive datatypes!

Output specification

Please use `DataIO.java` and its method `printSlice()` to print the calculated slice to the console. If you do not use this method or print any additional characters to the console automated testing may not be possible. Please also make sure that the slice is represented as a list of increasing statement numbers. This is how an textual output looks like: `[4, 5, 6, 7, 9, 11]`. Keep in mind that your slicer will be automatically tested. Thus, it is important that your implementation meets the output specifications.

If you need a graphical output of your calculated CFG and PDG (the corresponding nodes of your slice will be highlighted) you can use the methods `createCFGdotFile` and `createPDGdotFile` of the class `Visualizer.java`. However, these methods will only be executed if you pass a third command line argument *debug* to your program
(e.g. `java -jar staticslicer_1.jar testdata/input/test1.java "(12,{z})" debug`).

A Bug Reporting

A.1 General Notes

There are three key points to bear in mind when creating a bug report for a team of developers [1]:

- **Show me (how to show myself)**: How exactly can the bug be reproduced (be as precise as possible)?
- **Tell me exactly what you saw**: What has actually happened?
- **Tell me why you think what you saw is wrong**. What would you expect to happen instead?

Before reporting any issue make sure that the bug is not already known and that you are using the latest version/commit. In case there are multiple problems, report each issue separately. Some general advice to follow when writing your report [1, 2]:

- include a short and concise title
- be specific
- give rather more information than less (however stick to the relevant parts)
- use precise wording
- if suitable provide good attachments (screen captures/log files/etc.)
- reread what you wrote before submitting it
- follow up on your bugs

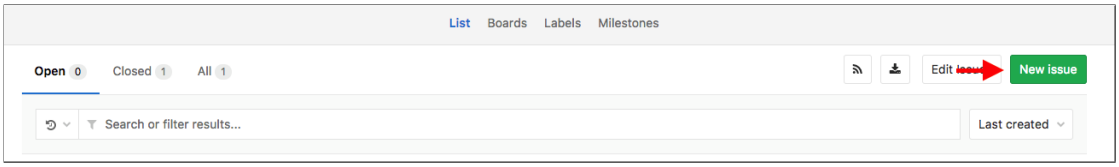
A.2 How to Create a Bug Report in `GitLab`

1. Create a new issue (Figures 1a).
2. Describe the bug (Figures 2a and 2b).
3. Select/Create a label for the bug (Figures 3a and 3b). You may also assign the bug to a developer, add a weight and due date etc.
4. Work on fixing the open issues and close them once resolved (Figures 4a and 4b).

References

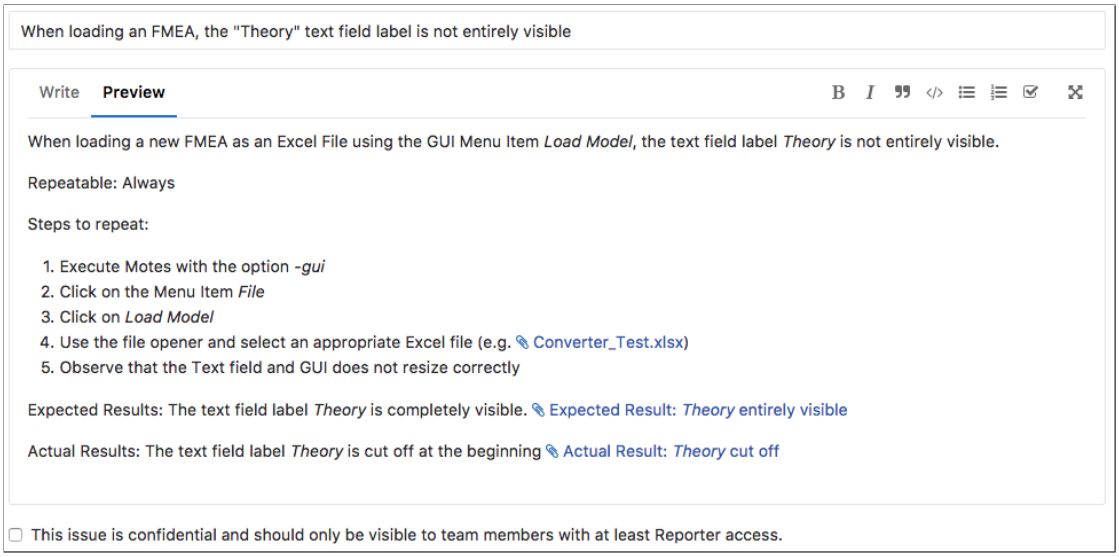
[1] Simon Tatham. *How to Report Bugs Effectively*. 1999.
`https://www.chiark.greenend.org.uk/~sgtatham/bugs.html`
(visited on 24/08/2017)

[2] Joel Montvelisky. *Principles of Good Bug Reporting*. 2008.
`http://qablog.practitest.com/principles-of-good-bug-reporting/`
(visited on 28/08/2017)

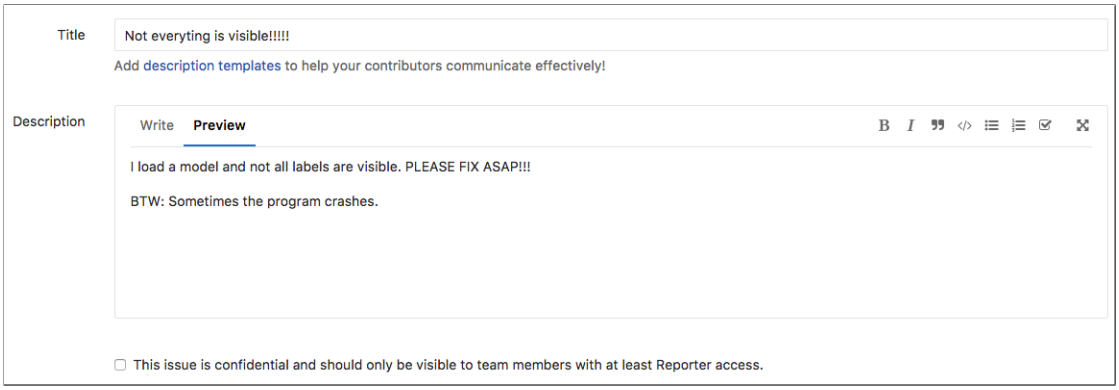


(a) Add a new issue

Figure 1: Creating a new issue in `GitLab`

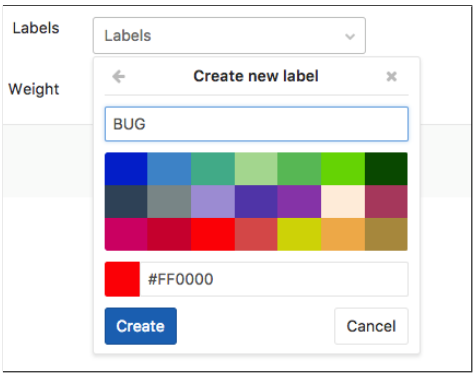


(a) Good example

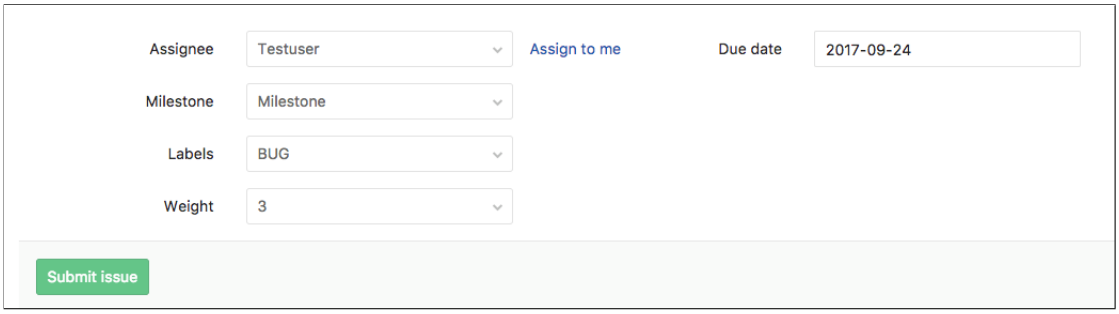


(b) Bad example

Figure 2: Describe the bug

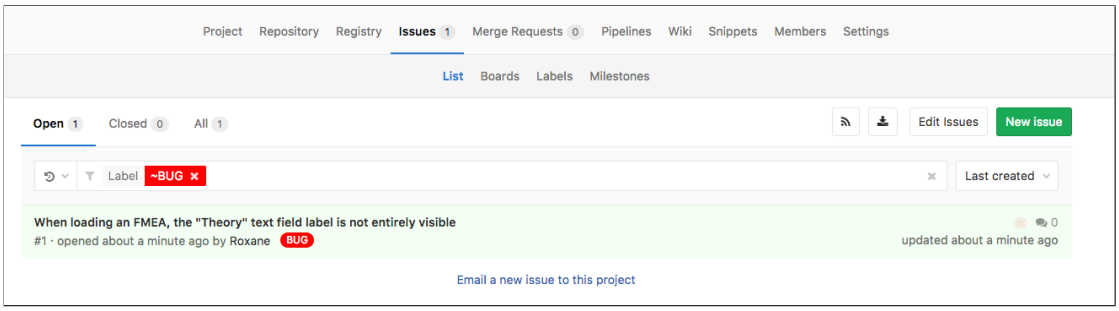


(a) Define a new label name and color code

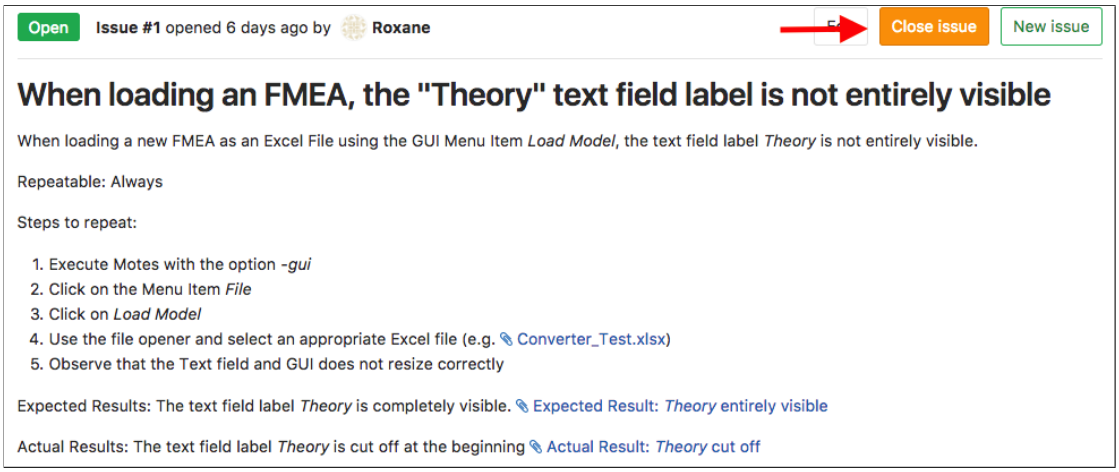


(b) Provide additional specifics, e.g., assign the bug to a developer, define a weight (i.e., impact of the bug), or set a due date

Figure 3: Add specifics to the bug



(a) Open issues



(b) Close fixed issues

Figure 4: Use the issue tracking system