university of groningen

faculty of mathematics and natural sciences

Bachelor project

Non Functional Testing In An Automated Assessment System

Expanding Justitia's assessment capabilities

Aloys Akkerman
First supervisor: *prof. dr. G.R. Renardel de Lavalette*
Second supervisor: *dr. A. Meijster*

July 5, 2013

Abstract

Justitia is an automated assessment system which assesses submissions (source code send it in by students) by running the submission with some input and comparing the output to a reference file. However, this type of testing can only test whether the answer is correct and not test whether the used algorithm meets some requirements. We are looking for a judging system that could test the implementation of search and sort algorithms by students.

In this thesis I will look at some automated assessment systems, none of them having the required assessment capabilities we are looking for. Thus, I created a new non functional judge. This judge is capable of assessing submissions to programming exercises that focus on the implementation of algorithms rather than the results.

Contents

Chapter 1

Introduction

At the University of Groningen the bachelor program Computing Science uses an online assessment system called "Justitia" (after Lady Justice). This online assessment system receives submissions from students who are trying to solve some programming exercises. Until now Justitia was only able to compare the output of a submission to the expected output. This kind of testing is called functional testing. Functional testing is a type of black box testing, it does not care about the actual implementation as long as a testcase produces the expected output. However, some exercises are not testable with this system. Some algorithms might have more than one correct answer. For other exercises we might want to check if some algorithm has a good runtime complexity, since less efficient implementations might give the correct answer but are unacceptably slow.

To illustrate this we could think of the "Higher, Lower" game (which is quite similar to binary search). A, the judge, will provide B, the submission, with a range of numbers e.g. 0 to 300. A tells B that he has chosen a number from that range and instructs B to guess which number. The only feedback provided after each guess is whether B should guess a higher or a lower number or that B has guessed it right. The judge could now check whether the submission is making the expected new guess each time. The teacher can decide how strict to judge the submission. He may require a perfect implementation of binary search, or give some upper bound for the amount of guesses (based on the minimum amount of guesses needed).

In this thesis I will describe an automated assessment concept which allows to assess submissions beyond functional testing. It might, for example, be challenging to create an exercise which could return multiple correct answers, but still be testable by comparing the output to a reference output. Or to test whether the student has implemented some algorithm correctly, or if the implementation meets the efficiency requirements. This now becomes possible using the new assessment method I describe in this thesis.

While doing research on this subject I found a good survey of automated assessment tools by Kirsti M Ala-Mutka [1]. This survey gives a proper insight in a couple of these systems. While reading more on the subject I found that there are some great

tools available, but they do not cover our assessment concept. Therefore I created an assessment tool which fits our requirements.

Justitia is already used in at least 3 courses of the Computing Science curriculum to judge submissions. A few other non Computer Science courses use the tool as well. The new functionality can be used in these courses to test the understanding of certain algorithms, like binary search, by the students. Some new exercises can be developed and hopefully with the new assessment capabilities other courses or universities might start using the new Justitia system.

Chapter 2

Requirements

The bachelor project discussed in this thesis was not an available project, so there were no requirements provided beforehand. During the first meeting with my supervisors we discussed what I would be creating, but immediately we discussed the requirements as well. The first requirement is also the most important, and it was directly mentioned by A. Meijster. The last requirements is implied, because my judging system would be used in Justitia, thus it should be integrated.

1. For students who use Justitia, there should be no difference between running their code from a shell command-line and running it by submitting their code in Justitia. Since this tool will be used in the first year of the Computing Science curriculum the students should not have to worry about system calls that occur during judging. Therefore all communication between the assessment tool and the submission must be using `stdin` and `stdout`.

2. Writing the logic for the judge must be easy for a teacher with a fair amount of programming experience. A template will be available, so the teacher should only have to write the judging logic and not the communication with the submission.

3. Everything must be documented, the source code as well as how to use the tool. The following aspects should be treated:

   - How to write assessment logic and build a judge.
   - How to run the tool locally for testing purposes.
   - How to integrate the tool with Justitia.

4. The new judge should be integrated with Justitia.

Chapter 3

About Justitia

The documentation of Justitia describes itself as follows: "Justitia is a program that automatically checks the answers to programming exercises."[6] More generally, we could say that Justitia is an automated assessment tool. Justitia was originally designed by A. Meijster and T. van Laarhoven and implemented by T. van Laarhoven for use in the course "Imperative Programming" at the computer science department of the university of Groningen.

3.1 Workflow

Justitia has a simple workflow for students:

1. The student writes a solution for a programming exercise. The solution is just an ordinary program that receives its standard input from the keyboard while its standard output is the screen.

2. The student submits the solution by sending the source files to Justitia.

3. Justitia compiles the submission (and indicates what went wrong on compilation failure).

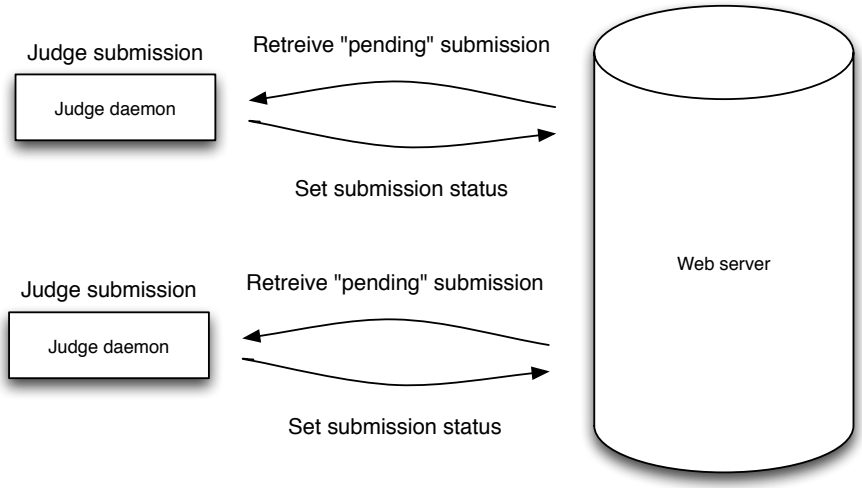
4. Justitia runs the compiled submission with some test input sets and compares the output to a reference output.

5. Justitia gives the user feedback whether the submission is accepted or not (and if not, Justitia may provide additional hints). If the submission exceeds a runtime limit it is stopped and the submission will be marked as not accepted.

3.2 Features

Justitia tests all submissions by inputting test sets and comparing the output with a reference output, this is pure functional testing. Submissions can also be judged without input data, but the output is still compared to a reference output.

One of the great aspects of Justitia is the distribution of its judges. These judges run as daemons on the systems that are available for the practicals. Using this setup assures us that the front-end of Justitia will keep running fast even though many submissions might be made. This is especially useful during practical midterms, because a substantial amount of submissions will be made in a small time frame and we cannot afford any slowdowns.



When a submission is done it is stored in the database, and the status is marked "pending". The front-end will not do anything more. A running daemon judge will connect to the database and check for the oldest submission with the "pending" status. The judge will run the submission (and mark the status "judging"), save the resulting output and compare it to the reference output. If the submission's output is equal to the reference output the judge will mark the status "passed" and otherwise "failed: wrong answer". If the judging fails for another reason, e.g. a compile error or runtime error, the status will be marked accordingly. If the submission did pass but was send in after the deadline it will be marked "missed deadline".

Justitia is by design not limited to a specific programming language (family). It allows you to create compile and run scripts for any language. Currently Justitia has built-in support for C, C++, Java, Python and Matlab (which runs in Octave on Linux).

Justitia supports two storage methods, the file system and the database. You can use both storage methods at the same time, this provides a backup and allows for a

more easy migration to another server. All submissions, both accepted and rejected, are stored by Justitia so that a submission could be reviewed.

3.3 Current usage

Currently Justitia is used primarily for three courses: Imperative Programming, Algorithms & Data Structures in C and Algorithmics. These courses let their students submit solutions to programming problems. A part of the final grade for these courses depend also on the results of the practicals, thus a reliable assessment tool is necessary.

Chapter 4

Related work

A survey of automated assessment approaches [1] gives an insight in a few of these systems and the advantages and disadvantages. Some of these assessment tools, e.g. Try [5], were already implemented more than two decades ago. Others are more recent such as CourseMarker [3], HoGG [4] and Online Judge [2].

4.1 Automated assessment systems

4.1.1 TRY

The TRY system is probably one of the oldest assessment tools. The system was created by Kenneth A. Reek in 1988/1989. This system is not an online assessment tool, since the internet did not yet exist the way we know it today. Instead of submitting written programs online, the student submits the program by running a provided tool called 'try'. Using the Unix set-user-id (`sudo suid`) mechanism, the TRY program can access the teacher's files even though the student does not have the required permissions.

The TRY system will run the submitted program by reading the input files (which are not accessible for the students) and use them as input for the program. The resulting output file is compared with the teacher's reference output file. The student will receive feedback whether the submission was accepted or if it failed some test case. The system also keeps a record of submissions in a log file.

Even though this system is not an online system, the assessment method - comparing output with reference output - in Justitia is the same. TRY is a very basic system and lacks many features other systems nowadays might have, but it is surely a system many other systems are based upon.

4.1.2 Online Judge

Online Judge is a basic online automated assessment system, which has been used in various courses at the National University of Singapore. The main reason to create this system was the amount of time it takes to check all submission for a programming exercise by hand. With this online solution the teacher would only have to check the maintainability of the code and not the correctness or efficiency, while the students receive instantaneous feedback on the correctness and efficiency of their submission.

Online Judge runs a student uploaded solution using some hidden input files. The output is compared with the expect output and the student receives the feedback "accepted", "wrong answer" or "syntax error". Syntax error means that the output is correct but differs in spacing compared to the reference output. All submission are subjected to a plagiarism check to discourage plagiarism. While checking the submitted solution the system also monitors the runtime and memory usage. If either one of them exceeds a given maximum for an exercise the submission will be rejected as well.

The core of this system is quite similar to Justitia. Both systems compare the output of the submission to a reference output and set some limits on the memory and time usage. The main difference with this system is that Justitia is lacking an automated plagiarism checker. A weak point in this system is that students could simply print the input to `stdout`. Now the students could write a program that only has a few print statements containing the expected output.

4.1.3 HoGG

HoGG (Homework Generation and Grading) is an automated assessment tool that is developed by and used at Rutgers University. The HoGG system is a bit different from the other described systems. It is only suitable for the Java programming language and uses Java's Reflection to find required methods and test them.

The students using HoGG must submit their solutions at different stages of completion. This makes it possible to give students a grade based on the partial completion of the exercise. This allows students to still receive reasonable grades if their solution produces correct results for most testcases, but not all.

The HoGG system is really different from the other discussed automated assessment systems in the way submissions are judged. Justitia does not provide this manner of judging, using Java Reflection, to judge submissions. However, the partial completion of an exercise is also possible in Justitia using multiple test cases. The teacher could design the test cases, in Justitia, so that each test case will need additional implementation. This allows students to test their new functionality incrementally.

4.1.4 CourseMarker

CourseMarker is the replacement of Ceilidh, an automated assessment system developed in the mid-80's. Ceilidh was used at the university of Nottingham for over 13 years. Ceilidh did have its limitations, and to overcome some of them CourseMarker was built. CourseMarker was implemented in 1998 and was already in use in that same year. In 2000 CourseMarker was made available to other universities, resulting in more than 20 universities using CourseMarker.

CourseMarker is a very broad system which can do more than only assessing programming exercises. It provides also options to have multiple choice questionnaires and submit essays. These different capabilities are provided by a broad range of tools that have been developed for Ceilidh (of which most are ported to CourseMarker). Even though CourseMarker may be very extendable, the correctness of a program is checked in the same manner as TRY, Online Judge and Justitia do. However, CourseMarker could also check the syntax of the submission, do a complexity analysis, analyse the program structure and check for specified program features.

Justitia and CourseMarker might not look similar, because CourseMarker is a very broad system. However, in the aspect of checking the correctness of a program both CourseMarker and Justitia run the submission with a predefined data-set and validate. CourseMarker provides however much more analytical options. CourseMarker is written in Java, therefore the system can not be altered while running. Justitia may be altered on the fly, which improves the flexibility for the maintainers.

4.2 Non functional testing

As seen in TRY, Online Judge and CourseMarker, non functional testing in automated assessment is not yet mainstream. Even though CourseMarker will dynamically analyse the submission, the acceptance of the submission is still primarily based on its correctness.

Automated assessment approaches for program efficiency are often based on the running time of the program, this is also available in Online Judge. Even though the CPU time is used, this is still not a good method. Consider a program containing an algorithm that should meet certain efficiency requirements. Although the algorithm might meet the efficiency requirements, the used CPU time could still be too high due to inefficient other parts of the program. However, those parts do not need to comply with the efficiency requirement of the algorithm. Besides, it is difficult to measure performance due to caching effects, current system load, network and I/O-bottleneck and other limiting factors.

To be able to test, for example, the efficiency of a sorting algorithm we need a solution which will monitor the number of steps that are taken rather than the execution time. In such a solution we could check if an algorithm has the correct runtime complexity and predict the scalability (by calculating the order of growth). This

will allow us to check if students did understand the working and implementation of certain algorithms.

In this thesis I develop a system which allows the assignment judge to communicate with the submission, allowing for a very broad range of possible use cases. For example students might implement a simple game like tic-tac-toe. These submissions could play against their judge and will only be accepted if they beat the judge, requiring a certain degree of intelligence in the submissions.

Chapter 5

Design

5.1 Student workflow

The workflow for the student is identical to the existing workflow for students who use Justitia (see section 3.1). Typically the workflow starts with the student receiving the assignment. A program has to be written which solves a certain problem and output its results following a given protocol.

The student will probably think about the assignment and write a solution for the problem (which may or may not be correct). When the student thinks his program meets the requirements he can submit the source code in Justitia. Justitia will judge the submission and provide feedback.

The submission can either be accepted or rejected. If the submission is accepted the student has completed the assignment. If the submission is rejected the student will receive some adequate feedback from Justitia (depending on what the teacher made available). The submission might not compile, crash at runtime or it might not meet some requirement(s) from the assignment. With the given feedback the student can improve or revise his code and submit again until he submits a program that meets all requirements of the assignment.

5.2 Teacher workflow

When a teacher creates a programming assignment using a non-functional test judge, a certain workflow should be followed.

We assume the teacher has already written the assignment specification including a protocol for the output (which should be testable). The teacher now should write a simple C program that will provide the submission with input and validate the output of the submission. The provided `assignment.c` and `assignment.h` files can be used to create an assignment.

assignment.h

```
#ifndef _HEADER_ASSIGNMENT_
#define _HEADER_ASSIGNMENT_

#include "judge.h"

/**
 * Define the maximum runtime in (real life) seconds.
 */
#define DEFAULT_MAXIMUM_RUNTIME 3

/**
 * This function will be called by the judge to start
 * judging the submission.
 *
 * @return AssignmentReturnStatus
 */
AssignmentReturnStatus judgeAssignment();

#endif
```

assignment.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include "judge.h"
#include "assignment.h"

/**
 * This function will be called by the judge to start
 * judging the submission.
 *
 * @return AssignmentReturnStatus
 */
AssignmentReturnStatus judgeAssignment()
{
    allowSubmissionToExit();
    printAndLog("No implementation yet, accept all ↩
        submissions\n");

    return ARS_Success;
}
```

When writing the exercise the teacher can read the output from the submission from `stdin` (using the default Lib-C methods e.g. `scanf`, `getchar`). Providing the submission with feedback/input data is done by writing to `stdout` (which is similiar to using a pipe in a shell). However, it is recommend to use the defined macros `logToFile` and `printLog`, which take the same arguments as `printf`. The `logToFile` macro will write to an output file (which will become visible in Justitia after judging). The `printLog` macro inherits the same functionality but writes the output also to the submission, which is useful to log the input data.

There is only one requirement for the assignment implementation: The assignment must indicate when it is expecting the submission to stop or die. This can be done

by calling `allowSubmissionToExit()`. If this method is not called all submissions will fail when they stop or die, because they stopped or died unexpectedly.

When the teacher is done with implementing the `assignment.c` and `assignment.h` files, they should be tested locally. The template comes with the non functional judge source code and the Lib-C hijack source. Both the judge and the Lib-C hijack library should be compiled, `Makefiles` are provided. During the compilation the assignment implementation becomes a part of the judge. The usage of the judge is a following:

Usage of the judge executable

```
$ ./judge.out
Usage:
./judge.out submission_executable output_file
```

However, to be able to let the judge communicate with the submission without buffering problems the hijack Lib-C library needs to be preloaded. This results in the final commmand to execute the judge locally for testing purposes (on Linux):

Judging a submission locally (Linux)

```
$ LD_PRELOAD=hijack-lib/lib_overrides.so ./judge.out submission.out ↩
    transscript.txt
[Output to stderr (e.g. debug information from the judge)]
JUDGE: Exit status 0
```

All the `logToFile` calls in the assignment will write their output to the transscript.txt file.

When the judge runs successfully, a new exercise can be created in Justitia (see the documentation of Justitia) by creating a new exercise folder in a course. The assignment files should be stored in a sub folder 'assignment' in the exercise folder and the following rules should be added to the exercise's info file:

Exercise info file flags

```
runner:  run_judge
checker: non-functional-checker
compiler files: assignment/assignment.c assignment/assignment.h
```

There is no need to compile files or do any additional setup. The assignment can now be used by students.

Note: The teacher may decide to use a time limit which can be set in the assignment header. This limit should be lower than the default Justitia limit. The default Justitia limit can be overwritten by adding a flag in the exercise info file. The default time limit of the judge can be changed by editing the judge source code in the Justitia bin folder, there is no need to compile afterwards (because each judge will be build on the fly).

Chapter 6

Implementation

The source code of all produced work is placed in the appendix of this thesis. A guide to integrate the non functional judge in Justitia and all needed scripts are included in the appendix as well.

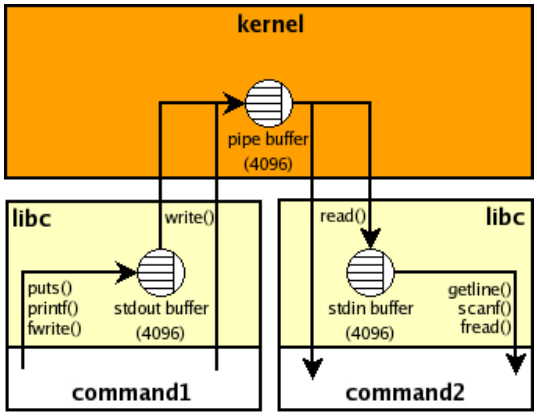
6.1 Communication between Judge and Submission

A fair amount of time in the design of the judge was spent on how to perform I/O operations between two processes. To establish a connection between two processes that communicate through standard I/O (e.g. `printf`, `scanf`), creating pipes is an obvious choice. However, pipes might not behave the way you would expect due to the build-in buffering rules of Lib-C. Lib-C is the library which provides most of the functions that would be considered system calls e.g. `read` or `write`, but also wrapper functions like `printf` and `scanf` which will internally invoke `read` and `write`.

An other solution, which is often suggested when you want to unbuffer your pipes are pseudo terminals. Because Lib-C will be fooled to believe that the program is running in a terminal it will change the default buffering settings to line buffering (in general). Even though pseudo terminals are a usable solution I decided to stick with pipes, because by the time I discovered pseudo terminals I had already found a solution to make sure there was no buffering from Lib-C (while pseudo-terminals cannot guarantee at least line buffering, even though this is almost always the case).

6.1.1 Lib-C and buffering

The Lib-C library, responsible for the functions in `stdio.h`, uses its own buffering to decrease the load on the file-system and kernel. If a program is run in a terminal, Lib-C detects that and uses line buffering. That means that every newline character (marking the end of a line) will trigger Lib-C to actually write the line to the destination.



Source: http://www.pixelbeat.org/programming/stdio_buffering

However, when the program uses pipes Lib-C detects that as well. In the case that you are not writing to a terminal Lib-C uses by default a buffer of 4096 bytes. Thus programs that require a lot of interaction and do not write enough data to flush the buffer will behave unexpectedly when using pipes.

Think of an implementation of a simple game like higher - lower (which is explained in the Introduction). The program requires some input and responses accordingly. However, the response is buffered in Lib-C causing the user to wait for a response while the program waits for new input from the user. The user now does not know what input to send because no useful feedback is received. This is exactly our problem, imagine that the user is our judge. How can we judge a submission without getting proper feedback before sending our new input data?

To disable these buffers is not a problem when you have the control over the source code of the program. Disabling the buffering for `stdout` could be done with the simple command `setbuf(stdout, NULL)` (as long as you put this command before any writing to `stdout`. However, we cannot guarantee a safe way to insert that function call into the source code of a submission. Therefore we need a solution that solves this problem without the need to modify the source code (directly).

6.1.2 Redefine print functions with compile-time macro's

The first thing I tried was redefining the functions that write implicitly to `stdout` by telling the compiler from the command line to define some macros (see the code below). These defines will replace a call to one of the function by a call to that same function with `fflush(stdout)` appended.

This works if you match the commands exactly with the defined macros. However, it is not a reliable method to use, because it will not work if, for example, `printf` gets called with a white space before the arguments. Adding more redefines is not an option, since you could call `printf` using an unlimited amount of combinations by partially defining `printf` (e.g. split `printf` in to `prin` and `tf`).

Redefines in a Makefile

```
REDEFINES= '-Dprintf(...)=printf(__VA_ARGS__); fflush(stdout)'\
           '-Dputs(x)=puts(x); fflush(stdout)'                \
           '-Dputchar(x)=putchar(x); fflush(stdout)'          \
```

6.1.3 Creating an unbuffered version of LibC

Since injecting `setbuf(stdout, NULL)` nor redefining I/O functions is a reliable way to disable buffering I looked at creating my own, unbuffered, Lib-C. This is a reliable option, because whatever the student writes in his source code, the unbuffered Lib-C will be called.

Even though it is simple enough to disable the buffering in the Lib-C source, this does not mean that using this unbuffered version of Lib-C is simple. Since linking to this unbuffered version of Lib-C proved to be quiet hard and labour intensive (since you must recompile Lib-C for all different systems). Another concern might be to determine which version of Lib-C we must use, it is much more safe to use the version that is provided by the operating system. I did some more research until somebody (an anonymous person on the web) pointed me in the direction of dynamic pre-loading a library.

6.1.4 Hijacking LibC with dynamic pre-loading

When a program is executed in Linux, all dynamic libraries on which the program depends are loaded into memory first. When a program calls a function, the function is resolved and invoked. Since the order of loading libraries affects the resolving of functions on a first-come first-serve basis, we could hijack some Lib-C functions by loading our library first. By default LibC is loaded first, so we set the environment variable `LD_PRELOAD=hijackIOmethods.so`. The operating system will now load our library first into the program, thus we can write our own I/O functions. Our functions can still call the Lib-C I/O functions and we can invoke a flush after every call. We do not need to worry about the version of Lib-C since we use the one provided by the system, we only wrap some code around the actual call.

An additional advantage is that, since we can override or wrap all functions from Lib-C, we can also disable a lot of functionality that should not be used by the students (e.g. the `fork` and `system` functions). This enabled us to write a kind of sandbox specifically tailored to our needs.

Chapter 7

Integration with Justitia

As stated in the requirements (requirement 4), the non functional judge should be integrated into Justitia. A teacher should be able to create an exercise which will use the non functional judge and the student should not be bothered by this in any way.

Since the non functional judge requires a buffer free communication with the submission, the Lib-C Hijack library is integrated in Justitia. To be able to use the non functional judge a new runner script was created. This script will run the judge with the assignment and judge the submission. Due to the design of Justitia we also need an additional checker script which will determine whether the submission should be accepted or rejected. The original checker scripts compare the output of the submission with a reference output. The new checker reads the output file of the judge and checks if the judge stopped with an "accept" status (status code 0). Note that there is not a different compile script for the submissions, since the submissions do not need to be modified. Dynamic pre-loading works fine with the existing compile scripts (assuming you do not use static linking). For information about how to use the non functional judge see section 5.2 (Teacher Workflow).

Even though the non functional judge can be used in Justitia, the integration is far from ideal. Justitia is designed based on the idea of a functional judge (comparing output to a reference output). This translates unfortunately into code that does not allow for new systems to easily integrate. This means that the usage of our non functional judge may not be very hard, but implementation is lacking flexibility and is not elegant. However, the integration of our system did not need to modify any existing files, thus it is easy and save to integrate in a live Justitia system.

Chapter 8

Conclusions

The goal of this project was to create a non functional judge that could assess submissions to programming exercises that could not be assessed using the existing judging functionality in Justitia. In this thesis I looked at other automated assessment systems, none of them has the functionality we require.

The non functional judge I described in this thesis meets all requirements. Programming exercises that could not be assessed using the existing functionality, such as search or sorting algorithms, can now be assessed by the non functional judge.

In the requirements it is stated that a teacher with a fair amount of programming experience should be able to use this new judge. For students there should be no difference between running their code from a shell command-line and running it by submitting their code in Justitia. Both of these requirements are met. I spent an afternoon with dr. A. Meijster to try to write a judge for a programming exercise with involves a sorting algorithm. We successfully created a judge for this programming exercise and when this judge is integrated in Justitia the students can just submit their code, which runs fine in the shell command-line.

The most important design problem was the buffering that is handled by Lib-C. Using the hijack method I have described in this thesis we have a reliable way to get rid of the buffering issues when judging a submission. However, to unbuffer all I/O functionality in Lib-C that writes to `stdout` the hijack library might need to be expanded a bit.

I am satisfied with the created work and I have learned a lot about dynamic loading of libraries and systems calls. I have also learned a lot about automated assessment systems in general and what we could improve in Justitia in the future.

Chapter 9

Future work

Plagiarism checker

As mentioned in the chapter Related Work, Justitia does not have a plagiarism checker like Online Judge. A plagiarism checker would be a nice extension for courses where a single exercise is graded by multiple correctors. It is really hard to spot plagiarism for those exercises. Besides, a plagiarism checker should be used for all assignments, to really discourage students to take the risk of cheating.

Sandboxing submissions

The hijack Lib-C library is capable of much more than just disabling output buffering. All functions in Lib-C which should not be used or accessible could be hijacked and mark a submission failed if such a functions is used. Disabling functions like `fork()` and `system()` could prevent fork-bombs or executing system commands. Thus, Justitia could become a much safer automated assessment system.

Rewrite the Justitia backend to make it extendable

As described in the chapters Integration with Justitia and Conclusions, Justitia is not really extendable. A rewrite of the back-end to a modern framework could help to make Justitia extendable, which will increase the usage possibilities. New judging methods could be created an installed more easily, allowing Justitia to become a more powerful assessment system.

Chapter 10

Acknowledgments

I would like to thank dr. G.R. Renardel de Lavalette for his support during my bachelor project. The regular meeting and sometimes hard deadlines helped me to keep on track with the project. The feedback on each draft of my thesis encouraged me to enhance the thesis and keep going.

I would also like to thank dr. A. Meijster for his assistance in my bachelor project. The afternoon we spent on testing the teacher template was really useful, it really made clear to me that the non functional judge can be used in the intended way. The extensive feedback on the last draft version of my thesis really helped me to improve this thesis.

Bibliography

[1] Kirsti M Ala-Mutka. A survey of automated assessment approaches for programming assignments. *Computer Science Education*, 15(2):83–102, 2005.

[2] Brenda Cheang, Andy Kurnia, Andrew Lim, and Wee-Chong Oon. On automated grading of programming assignments in an academic institution. *Computers & Education*, 41(2):121–131, 2003.

[3] Colin Higgins, Tarek Hegazy, Pavlos Symeonidis, and Athanasios Tsintsifas. The coursemarker CBA system: Improvements over Ceilidh. *Education and Information Technologies*, 8(3):287–304, 2003.

[4] Derek S Morris. Automatic grading of student's programming assignments: an interactive process and suite of programs. In *Frontiers in Education, 2003. FIE 2003 33rd Annual*, volume 3, pages S3F–1. IEEE, 2003.

[5] Kenneth A Reek. The try system -or- how to avoid testing student programs. In *ACM SIGCSE Bulletin*, volume 21, pages 112–116. ACM, 1989.

[6] A. Meijster T. van Laarhoven. Justitia documentation, 2009.

Chapter 11

Appendix

Teacher template

Workflow

1. Implement the assignment checker in `assigment.c`.
2. Build the judge by running `make`.
3. Build the Lib-C hijack library (by running `make` in ./hijack-libc).
4. Run the judge with the Lib-C hijack library preloaded.

Judge.out usage:

```
Usage:
./judge.out submission_executable output_file
```

Judge a submission

On Linux:

```
LD_PRELOAD=hijack-lib/lib_overrides.so ./judge.out submission.out transscript.txt
```

On OSX:

```
DYLD_FORCE_FLAT_NAMESPACE=1 DYLD_INSERT_LIBRARIES=hijack-lib/lib_overrides.dylib
            ./judge.out insert-path-to-submission-executable.out transscript.txt
```

Publish exercise in Justitia

To add the exercise to Justitia create a new blank exercise in Justitia (as usual). Then follow these steps:

1. Copy `assignment.c` and `assignment.h` into an assignment subfolder in the exercise directory.

2. Add at least the following lines to the exercise's info file:

```
time limit: 20  // should be higher than the timelimit set in assignment.h
runner: run_judge
checker: non-functional-checker
compiler files: assignment/assignment.c assignment/assignment.h
language: c
```

The exercise should now be available in Justitia.

assignment.h

```c
#ifndef _HEADER_ASSIGNMENT_
#define _HEADER_ASSIGNMENT_

#include "judge.h"

/**
 * Define the maximum runtime in (real life) seconds.
 */
#define DEFAULT_MAXIMUM_RUNTIME 3

/**
 * This function will be called by the judge to start
 * judging the submission.
 *
 * @return AssignmentReturnStatus
 */
AssignmentReturnStatus judgeAssignment();

#endif
```

assignment.c

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include "judge.h"
#include "assignment.h"

/**
 * This function will be called by the judge to start
 * judging the submission.
 *
 * @return AssignmentReturnStatus
 */
AssignmentReturnStatus judgeAssignment()
{
    allowSubmissionToExit();
    printAndLog("No implementation yet, accept all submissions\n");

    return ARS_Success;
}
```

judge.h

```
#ifndef _NON_FUNCTIONAL_JUDGE_HEADER_
#define _NON_FUNCTIONAL_JUDGE_HEADER_

#include <stdio.h>

/* Error statuses */
#define SUBMISSION_SUCCEEDS 0
#define ERROR_SUBMISSON_FAILED 10
#define ERROR_SUBMISSON_NOT_EXECUTED 20
#define ERROR_SUBMISSON_EXCEEDED_TIME_LIMIT 30
#define ERROR_SETUP_EVIRONMENT_FAILED 40
#define ERROR_SUBMISSION_DIED_UNEXPECTEDLY 50

/* Setup writing to logs and output */
#define LOG_FILE _outputFile
#define logToFile(...)   { fprintf(LOG_FILE, __VA_ARGS__); }
#define printAndLog(...) { printf(__VA_ARGS__); fprintf(LOG_FILE, ↩
    __VA_ARGS__); }

FILE* LOG_FILE;

enum
{
    ARS_Success,
    ARS_Failed,
    ARS_Error
} typedef AssignmentReturnStatus;

/**
 * Call this method before the submission is expected to quit / exit / ↩
     die. This
 * will cause the judge allow the submission to die without failing.
 */
void allowSubmissionToExit();

#endif
```

judge.c

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>
#include <libgen.h>
#include <errno.h>
#include "judge.h"
#include "assignment.h"

/**
 * Define maximum running time (real life seconds).
 * Override this in the assignment header
 */
#ifndef DEFAULT_MAXIMUM_RUNTIME
#define DEFAULT_MAXIMUM_RUNTIME 10
#endif

/**
 * Indicates whether the submission should be alive at this moment.
 * Before expecting the submission to die, set this variable in the ↩
     assignment
 * to 0.
 */
int submissionShouldBeAlive;

/**
```

```
27  * Indicates whether the submission did quit. We use this flag to ↩
        prevent the judge
28  * from exiting with failure if the submission did quit on time, but ↩
        the judge itself
29  * takes more time to process.
30  */
31 int submissionDidQuit = 0;
32
33 /**
34  * Store the process ID of the running submission
35  * to be able to kill it when pressing CTRL+C or
36  * when it exceeds the time limit.
37  */
38 pid_t submissionPID;
39
40 static void exitWithStatus(int status)
41 {
42     fprintf(stderr, "JUDGE: Exit status %d\n", status);
43     exit(status);
44 }
45
46 /**
47  * Listens for dying submissions (child process).
48  *
49  * @param signal
50  */
51 static void signalCHLDHandler(int signal)
52 {
53     if (submissionShouldBeAlive)
54     {
55         fprintf(stderr, "JUDGE: Submission died unexpectedly!\n");
56         waitpid(submissionPID, NULL, 0);
57         exitWithStatus(ERROR_SUBMISSION_DIED_UNEXPECTEDLY);
58     }
59
60     waitpid(submissionPID, NULL, 0);
61     submissionDidQuit = 1;
62 }
63
64 /**
65  * Handle SIGINT signals
66  *
67  * @param signal
68  */
69 static void signalINTHandler(int signal)
70 {
71     fprintf(stderr, "JUDGE: Killing submission!\n");
72     kill(submissionPID, SIGKILL);
73     waitpid(submissionPID, NULL, 0);
74     fflush(LOG_FILE);
75     exitWithStatus(EXIT_FAILURE);
76 }
77
78 /**
79  * Listens for alarm signal. If the signal is received, the
80  * submission process will be killed and the judge will exit
81  * (with error ERROR_SUBMISSON_EXCEEDED_TIME_LIMIT).
82  *
83  * @param signal
84  */
85 static void signalALRMHandler(int signal)
86 {
87     /* Only kill the submission and exit with error if the submission ↩
           is alive. */
88     if (! submissionDidQuit)
89     {
90         fprintf(stderr, "JUDGE: Submission exceeded time frame of %d ↩
               seconds!\n", DEFAULT_MAXIMUM_RUNTIME);
```

```
91          submissionShouldBeAlive = 0; /* prevent double warning */
92          kill(submissionPID, SIGKILL);
93          waitpid(submissionPID, NULL, 0);
94          fprintf(stderr, "JUDGE: stop with error code?!\n");
95          fflush(LOG_FILE);
96          exitWithStatus(ERROR_SUBMISSON_EXCEEDED_TIME_LIMIT);
97      }
98  }
99
100 /**
101  * Call this method before the submission is expected to quit / exit / ↩
         die. This
102  * will cause the judge allow the submission to die without failing.
103  */
104 void allowSubmissionToExit()
105 {
106     submissionShouldBeAlive = 0;
107 }
108
109 /**
110  * Setup the enviroment for the judging of the
111  * submission. Setup pipes and redirect stdin/stdout
112  * and stderr (only on submission). Fork this process
113  * and start the client.
114  *
115  * @return int
116  */
117 static int setupEnvironment(char* submission)
118 {
119     /* Create pipes */
120     int pipeJudgeToClient[2];
121     int pipeClientToJudge[2];
122     if (pipe(pipeJudgeToClient) || pipe(pipeClientToJudge))
123     {
124         fprintf(stderr, "JUDGE: Creating pipes resulted in errno: ↩
              %d\n", errno);
125         return 0;
126     }
127
128     /* Indicate that the submission should be alive */
129     submissionShouldBeAlive = 1;
130     if ((submissionPID = fork()) == -1)
131     {
132         submissionShouldBeAlive = -1;
133         fprintf(stderr, "JUDGE: Forking to run submission resulted in ↩
              errno: %d\n", errno);
134         return 0;
135     }
136
137     if (submissionPID == 0)
138     {
139         /* send stdout (client) to the pipe (judge) */
140         dup2(pipeClientToJudge[1], STDOUT_FILENO);
141         close(pipeClientToJudge[0]);
142         close(pipeClientToJudge[1]);
143
144         /* send pipe (judge) to the stdin (client) */
145         dup2(pipeJudgeToClient[0], STDIN_FILENO);
146         close(pipeJudgeToClient[0]);
147         close(pipeJudgeToClient[1]);
148
149         /* Run submission */
150         execlp(submission, basename(submission), NULL);
151         fprintf(stderr, "SUBMISSON: Submission could not be started!\n");
152         exit(ERROR_SUBMISSON_NOT_EXECUTED);
153     }
154
155     /* Register signal handler only on judge */
```

```
156     signal(SIGCHLD, signalCHLDHandler);
157     signal(SIGALRM, signalALRMHandler);
158     signal(SIGINT, signalINTHandler);
159
160     /* send stdout (judge) to the pipe (client) */
161     dup2(pipeJudgeToClient[1], STDOUT_FILENO);
162     close(pipeJudgeToClient[0]);
163     close(pipeJudgeToClient[1]);
164
165     /* send pipe (client) to the stdin (judge) */
166     dup2(pipeClientToJudge[0], STDIN_FILENO);
167     close(pipeClientToJudge[0]);
168     close(pipeClientToJudge[1]);
169
170     return 1;
171 }
172
173 /**
174  * Runs the assignment judger and converts the return
175  * status to an EXIT status for the judge to describe
176  * the result.
177  *
178  * @return int
179  */
180 static int judgeSubmission(char* outputFile)
181 {
182     /* Open the log file for writing */
183     LOG_FILE = fopen(outputFile, "w");
184
185     alarm(DEFAULT_MAXIMUM_RUNTIME);
186     int status = judgeAssignment();
187
188     switch (status)
189     {
190         case ARS_Success:
191             status = SUBMISSION_SUCCEEDS;
192             break;
193         case ARS_Failed:
194             status = ERROR_SUBMISSON_FAILED;
195             break;
196         default:
197             status = ERROR_SUBMISSON_FAILED;
198     }
199
200     fclose(LOG_FILE);
201     return status;
202 }
203
204 int main(int argc, char* argv[])
205 {
206     if (argc != 3)
207     {
208         printf("Usage:\n%s submission_executable output_file\n", ↩
              argv[0]);
209         exit(EXIT_FAILURE);
210     }
211
212     /* Disable buffering, no necessary when using our hijack library, ↩
          but it does not hurt */
213     setbuf(stdout, NULL);
214     int returnStatus = SUBMISSION_SUCCEEDS;
215
216     if (setupEnvironment(argv[1]))
217     {
218         returnStatus = judgeSubmission(argv[2]);
219     }
220     else
221     {
```

```
222         returnStatus = ERROR_SETUP_EVIRONMENT_FAILED;
223     }
224
225     /* Do not exit before the submission is dead */
226     waitpid(submissionPID, NULL, 0);
227
228     /* Exit with a status message, so the checker can verify success */
229     exitWithStatus(returnStatus);
230     return 0;
231 }
```

Makefile (Judge)

```
1  # Set the desired compiler
2  CC=gcc
3
4  # Set the desired flags used by the compiler
5  # during the compilation of each source file
6  CFLAGS=-Wall -g
7
8  # Set the desired output name
9  EXECUTABLE=judge.out
10
11 # Set all sources files using .o as extension
12 OBJS=judge.o assignment.o
13
14 all: $(EXECUTABLE)
15
16 $(EXECUTABLE): $(OBJS)
17     $(CC) $(OBJS) -o $(EXECUTABLE)
18
19 clean:
20     rm -rf *o $(EXECUTABLE)
```

hijack-libc/lib_overrides.c

```
1  #define _GNU_SOURCE
2
3  #include <stdio.h>
4  #include <unistd.h>
5  #include <dlfcn.h>
6  #include <stdarg.h>
7
8  /**
9   * For caching the original methods
10  */
11 static int (*original_putchar) (int) = NULL;
12 static int (*original_puts) (const char *) = NULL;
13 static int (*original_printf) (const char *, ...) = NULL;
14
15 int putchar(int ch)
16 {
17     if (!original_putchar)
18     {
19         original_putchar = dlsym(RTLD_NEXT, "putchar");
20     }
21
22     int result = original_putchar(ch);
23     fflush(stdout);
24
25     return result;
26 }
27
28 int puts(const char * string)
29 {
30     if (!original_puts)
31     {
```

```
32          original_puts = dlsym(RTLD_NEXT, "puts");
33      }
34
35      int result = original_puts(string);
36      fflush(stdout);
37
38      return result;
39  }
40
41  int printf(const char * format, ...)
42  {
43      if (!original_printf)
44      {
45          original_printf = dlsym(RTLD_NEXT, "vprintf");
46      }
47
48      va_list args;
49      va_start(args, format);
50      int result = original_printf(format, args);
51      va_end(args);
52      fflush(stdout);
53
54      return result;
55  }
```

hijack-libc/Makefile (hijack library)

```
1  # Set the desired compiler
2  CC=gcc
3
4  # Set all sources files using .o as extension
5  linux:
6      $(CC) -Wall -c -Werror -fpic -ldl *.c -o lib_overrides.o
7      gcc -shared -ldl -o lib_overrides.so lib_overrides.o
8
9  osx:
10     $(CC) -Wall -dynamiclib *.c -o lib_overrides.dylib
11 clean:
12     rm -rf *.o *.dylib *.so
```

Ingration in Justitia

To integrate the non functional judge follow these steps:

1. Copy the `/bin/non-function-judge` folder to `/backend/bin/`.
2. Run `Make` in `/backend/bin/non-function-judge`
3. Copy the `/bin/hijack-libc` folder to `/backend/bin/`.
4. Copy the `/checkers/non-functional-checker.sh` file to `/checkers/`.
5. Copy the `/compilers/c.sh` file to `/compilers/` (do not static link libraries).
6. Copy the `/runners/run_judge.sh` file to `/runners/`.
7. Make sure all permissions are set correctly, at least read and execute rights. The user that that runs the Justitia judge deamons should own all these files.

compilers/c.sh

```
#!/bin/sh

# C compile wrapper-script.
# Usage: <this-script> <input> <output> <errorfile>

SOURCE="$1"
DEST="$2"
ERROR="$3"
FLAGS="$4"

# -Wall:    Report all warnings
# -O2:      Level 2 optimizations (default for speed)
# -static:  Static link with all libraries
# -lm:      Link with math-library (has to be last argument!)
gcc -Wall -O2 -o $DEST $SOURCE $FLAGS -lm 2>$ERROR
exit $?
```

runners/run_judge.sh

```
#!/bin/sh

# Run wrapper-script

# Usage: $0 <program> <testin> <output> <error> <flags>
#
# <program>   Executable of the program to be run.
# <testin>    File contains the assignment source (c code)
# <output>    File where to write solution output.
# <error>     File where to write error messages.
# <flags>     More flags

PROGRAM="$1";   shift
TESTIN="$1";    shift
OUTPUT="$1";    shift
ERROR="$1";     shift

# Set the location of the unbuffering library
UNBUFFER_LIB="/var/subdomains/justitia/backend/bin/hijack-libc/lib_overrides.so";

# Set the location of the judge template sources
JUDGE_SOURCE="/var/subdomains/justitia/backend/bin/non-functional-judge";

# Get the directory of the submission
SUBMISSION_DIR=$(dirname $PROGRAM);

# Make the judge for this assignment
JUDGE_EXEC=$(tempfile);
gcc -I$JUDGE_SOURCE -I$SUBMISSION_DIR/assignment $JUDGE_SOURCE/*.c ↩
    $SUBMISSION_DIR/assignment/*.c -o $JUDGE_EXEC;
chmod 777 $JUDGE_EXEC;

# Run the actual judge
# echo "LD_PRELOAD=$UNBUFFER_LIB $JUDGE_EXEC $PROGRAM $OUTPUT 2>$ERROR";
#

LD_PRELOAD=$UNBUFFER_LIB $JUDGE_EXEC $PROGRAM $OUTPUT 2>$ERROR

cat $ERROR >> $OUTPUT;
rm -rf $JUDGE_EXEC;

exit $?
```

checkers/non-functional-checker.sh

```
#!/bin/sh
```

```
2
3  # Checker script
4
5  # Usage: $0 <testout> <refout> <diff> <flags>
6  #
7  # <testout>   File containing submission output.
8  # <refout>    File containing reference output.
9  # <diff>      File where to write the diff.
10 # <flags>     Extra flags.
11
12 # Check if the last line contains a "OK" status
13 tail -n 1 $1 | grep -q "JUDGE: Exit status 0";
14
15 exit $?
```