UNSW
AUSTRALIA

School of Computer Science and Engineering
Faculty of Engineering
The University of New South Wales

A Testing Tool for Introductory Programming Courses

By Kyu-Sang Kim

z5208931

Supervised by Andrew Taylor (UNSW)
Assessed by John Shepherd (UNSW)

Thesis C Report — Term 3, 2022
Submitted *December 2022*

Thesis submitted as a requirement for the degree of
Bachelor of Engineering in Software Engineering (Honours)

Abstract

Automated code testing tools are valuable to course staff in the administration of *practical coding assessments* to students. In this thesis, we will evaluate the architecture and implementations of *automated code testing tools* such as *autotest*, and develop a new software package with the proper *technical debt* management procedures. Thus, this thesis aims to design and implement a *automated code testing tool* that is suitable to replace the existing *autotest* at UNSW introductory programming courses and has measures implemented to minimise technical debt into the long term.

Acknowledgements

I would like to thank Andrew Taylor and John Shepherd for their guidance as supervisor and assessor for this project respectively. I would also like to thank the numerous UNSW Computer Science and Engineering school course administrators who have taken interest in the project by providing their thoughts and ideas: Shrey Somaiya, Zac Kologlu and Tom Kunc.

Finally, I would like to thank my family and friends for all their support during both my degree and this thesis.

Abbreviations

UNSW The University of New South Wales
CSE Computer Science and Engineering School
pip PIP Installs Packages - de facto and recommended package installer for Python
MVP Minimal Viable Product
CPU Central Processing Unit
IPC Interprocess Communication
PID Process ID
UTS Unix Time-Sharing
NIS Network Information Service
POSIX Portable Operating System Interface

Contents

List of Figures

List of Tables

Introduction

1.1 Motivation

1.1.1 Student Enrolments & Practical Coding Assessments

In recent years, student enrolments in introductory programming courses at UNSW have increased at an extraordinary rate. Despite the increase in students, course staff must still continue to deliver the best learning experience, fulfil course outcomes and provide a numerical "mark" to students upon course completion.

Course/Year	2017	2018	2019	2020	2021
COMP1511	1351	1655	1874	1905	2381
COMP1521	715	1136	1352	1417	1633
COMP2521	378	1019	1389	1445	1551

Table 1.1: Course enrolments from UNSW Class Timetable (April 2022)

One approach taken by UNSW computing courses to assist in calculating a student's "mark" are *practical coding assessments*. They allow for students to demonstrate their knowledge and understanding of the course content which can then be assessed at a later time by course staff.

In order to support course staff in the administration of these *practical coding assessments*, *automated code testing tools* can be utilised to automatically determine the correctness of a student's submitted code to a given specification with minimal course staff intervention. The results of these tools can then be further processed to generate a grade for the student's code submission.

In some courses, these tools can also be made available to students by course staff to perform basic correctness checks on their code before submission.

1.1.2 Technical Debt

In 1992, Ward Cunningham who first introduced the concept of *technical debt* described it as how "shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite" [2].

Although the meaning of *technical debt* varies on the context and each reader's interpretation, this thesis will assume *technical debt* to be any component or aspect of a project that could be prescribed as a weakness or flaw by a reader or contributor. As an example, *technical debt* could be observed as issues that hinder the ease of understanding and improvement of a project.

1.1.3 autotest

autotest is a Python-based *automated code testing tool* written by Andrew Taylor which is an evolution of a earlier script by Mei Cheng Whale [1]. It was first introduced into the COMP2041 *Software Construction: Techniques and Tools* course in the second semester of 2015 and has since become the standard for introductory programming courses at UNSW to automatically assess the correctness of code to a given specification.

Despite the gradual improvements and fixes made to *autotest* since it's introduction, *autotest* is plagued with *design deficiencies* and non-trivial amounts of *technical debt* which has affected the scalability of *autotest* in response to increasing student numbers as well as complicate development of new features.

In particular, major shortcomings can be identified if we analyse the *autotest* code to the framework defined within *An Exploration of Technical Debt* [3]. A clear example is the design and architectural debt stemming from the ill-considered development of *autotest* which has hindered it from easily implementing performance improvements such as multiprocessing. An additional example is the knowledge distribution debt stemming from the ineffective documentation of *autotest* which has obstructed new users in their adoption of *autotest* and confused code contributors on the tool's inner-workings. Resolving the aforementioned technical debts could reduce the time spent by course staff performing assessment marking and creation respectively from an already constrained time pool.

As a result of these findings, there were efforts undertaken in 2021 by the UNSW CSE community including Andrew Taylor to remediate major sources of technical debt from the *autotest* tool in which some measures were successful. However, there exists issues such as the lack of documentation and fundamental design flaws that have been seen as infeasible to fix without a significant overhaul[4].

1.2 Thesis Problem Statement

Automated code testing tools are valuable to course staff in the administration of *practical coding assessments* to students. In this thesis, we will evaluate the architecture and implementations of *automated code testing tools* such as *autotest*, and develop a new software package with the proper *technical debt* management procedures. Thus, this thesis aims to design and implement a *automated code testing tool* that is suitable to replace the existing *autotest* at UNSW introductory programming courses and has measures implemented to minimise technical debt into the long term.

1.2.1 Thesis Structure (FIXME)

Chapter 1 covers the motivations behind this thesis as well as the thesis problem statement. Chapter 2 explains the background concepts that are relevant to understanding the later chapters of the thesis. Chapter 3 undertakes an evaluation and review of the existing works related to this thesis. Chapter 4 declares design requirements/goals and proposes approaches that fulfil as many of the goals as possible before providing a justification for selecting one approach. Chapter 5 provides detailed information on each component and measures. Chapter 6 evaluates the solution in chapter 5 by comparing performance to the existing *autotest* in a controlled environment. Finally, Chapter 7 summarises the contents of this thesis report.

Background

First, it is important to explore what technical debt is and the various approaches to management, then it is possible to start looking at the concepts of how code marking is manually conducted and it's flaws at scale. This thesis will then evaluate the various approaches to automated code testing and how they can be applied to determine the correctness of code to a given specification. An overview of software containerisation is also included to assist in understanding components of the solution implementation.

2.1 Management of Technical Debt

Technical debt is inevitable in any project but implementing measures to understand, communicate and manage technical debt from the outset can make a big difference in both the short- and long-term success of the project. One particular benefit is that contributors are able to make informed decisions which are aimed towards reducing the time and work required to understand and improve the project in both the short- and long-term [5].

To minimise sources of various technical debt in this thesis project, we will implement the following measures:

- **Extensive project and code documentation** - Extensive project and code documentation will greatly assist a reader in understanding the usage and internals of project faster which will be useful for improvements in the future.
- **Project management tools** - Tools such as Github and Jira allow for the tracking of code and issues respectively to assist in communicating and management of technical debt between multiple project contributors to minimise confusion and allow each contributor to work independently.
- **Modular architectural design** - Approaching the project with a modular architectural design in mind will assist in long term development as decoupling dependencies between major components will make them easier to replace when upgraded.
- **Solution modernisation & focus on feature extensibility** - Similar to modular architectural design, approaching the project with a focus in supporting easier solution modernisation and

feature extensibility will reduce the need for a replacement of the entire project instead of targeted components. If possible, solution modernisation should also be implemented.

- **Automated regression testing** - Automatic regression testing of code will assist in development by allowing contributors to create and improve features whilst enforcing an accepted baseline of program correctness. This ensures that minimal code errors which can also be considered technical debt is deployed for users of the project.

Successful implementation of the above measures will ensure that the project has minimal technical debt in both the development and post-release period.

2.2 Code Testing & Marking

This thesis assumes that *code marking* refers to the determination of whether provided code when executed conforms to some behaviour that is outlined in a specification or similar resource.

As an example, Figure 2.1 highlights a program that concurs to a specification of checking if a given number argument is prime and a program that fails to do so:

Correct Program

```
$ ./is_prime 37
37 is prime
```

Incorrect Program

```
$ ./is_prime 37
37 is not prime
```

Figure 2.1: Example program that conforms to a specification (left) and one that does not (right)

The process of testing submitted code for correctness to a specification can be done manually by course staff but some issues do arise from this [6]:

- Tedious and repetitive work is known to be prone to mistakes as a result of human error.
- Potential inconsistencies in the assignment of marks when distributed between separate markers.
- Increasing marking load per course staff member due to increasing student numbers can exacerbate mistakes and inconsistencies.

To remediate these issues and improve the process of code testing, the thesis will turn to an automated platform to perform testing.

2.3 Automated Testing & Marking Approaches

The earliest implementations of automated code testing on student code was published in 1960 by Jack Hollingsworth of the Rensselaer Polytechnic Institute. The main goal of his tool was to autonomously verify the correctness of student submitted code in relation to a pre-defined expected behaviour [7].

The tool led to some benefits to both students and course staff as follows:

- Time spent on manual code testing by course staff could be utilised in other tasks.
- Students were observed to learn programming and gain confidence better with an automatic grader over dedicated lab groups.
- Increasing enrolments for programming courses per teaching period became economically feasible.

The thesis is based upon the idea that the benefits of automated code testing tools are valuable but repeatable and consistent methodologies of determining whether "submitted code matches expected behaviour" need to be explored. As such, this thesis elaborates and evaluates two possible methodologies, *external program side-effect comparison* and *internal program unit testing*:

2.3.1 External Program Side-effect Comparison

This thesis assumes *side-effects* to be *observable output* from a tested program that has been stored on external resources such as the console via *stdout* or files that store logs and program state.

In external program side-effect comparison, the side-effects of a tested program can be externally compared to the side-effects generated by a sample solution to determine whether expected behaviour has been achieved. This methodology could be compared to the example of manual code testing in Figure 2.1.

As a benefit, comparisons on side-effects after program execution allows for simpler testing of complex program with multiple dependencies to other code sources. However, this benefit is traded off with the necessity of storage space to store the side-effects of the tested program before it can be compared to the expected values.

The interactions between the code tester, the test program and side-effects can be summarised in Figure 2.2.

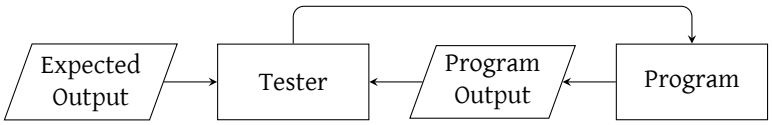

Figure 2.2: External Program Side-effect Comparison Overview

2.3.2 Internal Program Unit Testing

Unit testing is assumed to be the evaluation of *individual discrete code components* within a tested program by comparing internal program state such as a function return value to a known expected state that has been pre-generated by a sample solution.

In internal program unit testing, the execution and collection of results from internal unit test cases can be used to determine whether expected behaviour has been achieved.

As a benefit, testing internal code components permits for finer control on the determination of program correctness by exposing internal functions instead of being reliant on observing side-effects. However, this benefit is traded off with the difficulty of unit testing more complex programs which may not be able the share the same unit testing framework throughout all components.

The interactions between the code tester and test program can be summarised in Figure 2.3.

Figure 2.3: Internal Program Unit Testing Overview

2.3.3 Differences

It is possible to argue that both *side-effect comparison* and *unit testing* seem very similar since each technique checks program "state" after executing some code. However, the thesis was unable to find any other suitable and usable technique in determining program correctness within a reasonable amount of time.

2.4 Software Containerisation

Containerisation is generally accepted as the "packaging together of code with all it's necessary components [such that packages] are isolated in their own container". This allows containers to be run consistently in any environment and on any infrastructure regardless of the operating system. More importantly, containers are isolated from the rest of a host system which will be an important feature for the thesis.[19] In order to understand how *containerisation* works and how it could be of benefit to this thesis, we need to understand how *Linux namespaces* work to provide *containerisation*.

2.4.1 Linux Namespaces

Namespaces are a Linux kernel feature introduced in 2002 which permits the isolation of a process from the rest of the host system. To provide finer control over the individual aspects of a process, there are currently eight *namespaces* that can be created as shown in 2.2. This provides the ability to isolate a process without removing aspects such as network access which a user may wish to keep intact. All of these

namespaces with the exception of the *user namespace* (Linux >= 3.8) require the *CAP_SYS_ADMIN* kernel capability to be enabled. No privilege is required for the *user namespace* with Linux 3.8 or above.[20]

Namespace	Resources Isolated
Cgroup	Cgroup root directory (system resource allocations such as cpu or memory)
IPC	System V IPC, POSIX message queues
Network	Network devices, stacks, ports, etc.
Mount	Mount points (file system mounts)
PID	Process IDs (Process ID table)
Time	Boot and monotonic clocks
User	User and group IDs (ability for a process to have root privilege within the namespace)
UTS	Hostname and NIS domain name

Table 2.2: Linux namespaces and resources isolated by each namespace

Isolating all eight *namespaces* allows a process to be mostly separate from the host system which can serve as the base for a container instance. Most widely used container engines such as *Docker* and *Podman* rely upon *namespaces* among other techniques such as *overlay filesystems* to provide *containerisation*.

2.4.2 Rootless Containers

The majority of popular container engines require the host system to execute the engine with root privileges in order to deliver full functionality. In recent years, the idea of *rootless containers* has gained traction to counter the possible damage to a host system if a container run under root privileges is escaped by a malicious program which would then also assume root permissions. Although good practices by the user can minimise the risk of a container escape, undiscovered vulnerabilities and user error can bypass the efforts taken by the user. *Rootless containers* remove the risk as any malicious programs escaping a container will no longer assume privileged access.

As a result of the benefits, industry standard container engines such Docker and Podman have introduced rootless versions of their engines but are severely lacking in usage documentation and troubleshooting due to poor uptake by a majority of users who simply accept the risk or require features only available with privileged execution.

With this overview, the thesis will refer to “containerisation” as one of it’s main benefits which is the isolation of an executing program from the remainder of a host system. The other main benefits such as software portability are noted but did not have a functional role in this thesis outside of potential future work.

Related Works

It is important to explore some existing work which could influence the design towards a chosen direction. In particular, the thesis explores and evaluates *autotest* - the current automated code testing tool used for introductory programming courses at UNSW [1]; *check50* - another implementation of an automated code testing tool used in the CS50 introductory programming course at Harvard University [8]; and *gtest* - a C++ unit testing framework used by Google and the programming community [9]. Historic and minimally relevant implementations are also mentioned for completeness.

3.1 autotest

As mentioned in Chapter 1, *autotest* is a software package designed by Andrew Taylor to fulfil the purpose of an automatic code testing tool and has become the standard code testing tool for introductory programming courses at UNSW.

3.1.1 Implementation & Usage Overview

autotest is a tool that primarily follows the *external program side-effect comparison* methodology to automatically determine the correctness of a tested program which can be seen in Figure 3.4. *autotest* is written in Python but does expose some Bash shell and Perl script interfaces as a result of it's historical relation with the earlier script by Mei Cheng Whale.

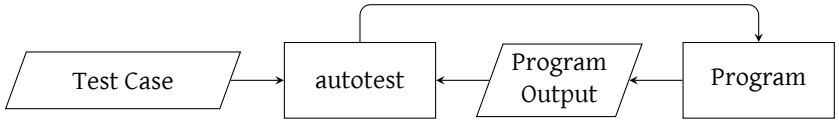

Figure 3.4: autotest High-level Architecture

In terms of usage, *autotest* provides a simplified interface to define multiple test cases with custom test parameters and environment in accordance with the needs of the user. In particular, the ability to set a time limit for each test case is valuable in handling cases where code may have an unintentional infinite loop, a common mistake made by beginners to programming.

```
files=is_prime.c

1 stdin="39" expected_stdout="39 is not prime\n"
2 stdin="42" expected_stdout="42 is not prime\n"
3 stdin="47" expected_stdout="47 is prime\n"
```

Figure 3.5: autotest Example Test Cases

Files containing these test case specifications (Figure 3.5) can be saved to a directory and easily utilised by a wrapper script (Figure 3.6) which generalises and simplifies execution of the tool via abstraction of certain details from the intended end-user:

```
#!/bin/sh

parameters="
default_compilers = {'c' : [['gcc', '-Werror', '-std=gnu11', '-g', '-lm']]}
"

exec <path_to_autotest>/autotest.py -a <path_to_test_dir> --parameters "$parameters" "
    $@"
```

Figure 3.6: autotest Wrapper Script

Upon execution of *autotest* with a correct *is_prime.c*, the output shown in Figure 3.7 is expected which informs the user that the provided test cases have passed. In this particular case, the tested program has successfully provided the expected output as defined by each of the test cases.

```
gcc -Werror -std=gnu11 -g -lm -o is_prime is_prime.c
Test 1 (is_prime) - passed
Test 2 (is_prime) - passed
Test 3 (is_prime) - passed
3 tests passed 0 tests failed
```

Figure 3.7: autotest execution on correct program

In the event that *autotest* is executed with a flawed *is_prime.c*, *autotest* will report to the user on the cause of the failure of a test case whether it is a mismatch in side-effects in the case of Figure 3.8 or other cause such as compilation failure. Over the lifetime of *autotest*, the unique situations that the tool can detect has increased which in turn provides users with additional targeted information and context.

This information is useful to both student and staff as it may be difficult to manually analyse the differences in larger amounts of side-effects created from more complex programs or program executions. As such, it is one of the defining features that separates *autotest* from other implementations.

It is noted that to ensure a test program's full compliance to a given specification with *autotest*, sufficient test cases to cover all edge cases and standard execution will be required. This can be a challenge

```
gcc -Werror -std=gnu11 -g -lm -o is_prime is_prime.c
Test 1 (is_prime) - passed
Test 2 (is_prime) - passed
Test 3 (is_prime) - failed (Incorrect output)
Your program produced this line of output:
47 is not prime

The correct 1 lines of output for this test were:
47 is prime

The difference between your output(-) and the correct output(+) is:
- 47 is not prime
?      ----

+ 47 is prime

The input for this test was:
47
You can reproduce this test by executing these commands:
gcc -Werror -std=gnu11 -g -lm -o is_prime is_prime.c
echo -n 47 | is_prime
2 tests passed 1 tests failed
```

Figure 3.8: autotest execution on incorrect program

depending on the complexity of the specification but it is noted that testing coverage is out of scope for *autotest*.

3.1.2 Benefits

Further analysis of *autotest* and it's usage at UNSW can provide the following list of benefits for this thesis as an inspiration:

- ***autotest* has been proven to be mostly reliable at UNSW** - There have been undocumented reports of rare autotest crashes when being run in a distributed fashion but no documented reports could be found of the issue affecting administration of coding assessments within UNSW [11].
- ***autotest* is able to support any programming language in which *autotest* can detect and compare side-effects to determine program correctness** - This assists greatly in the usability of *autotest* by courses that do not share an identical programming language as *external program side-effect comparison* is a language-agnostic methodology.
- ***autotest* exposes many parameters in the test case specification which allows users to have finer control over the test execution environment** - Some specifications that *autotest* may want to test could require specialised testing environments for it's test cases.
- ***autotest* has a side-effect comparison module that provides meaningful information on differences between actual and expected output in the event of a test failure** - This is one of the most defining components of *autotest* in the assistance it provides to users.

3.1.3 Limitations

Despite the benefits of *autotest*, there are unfortunately some limitations that make *autotest* unsuitable for continued use and have motivated this thesis:

- ***autotest* was not designed for long term maintainability and feature extensibility** - This is evident in the difficulty to change major components of *autotest* without significant and difficult overhaul on large amounts of the code base as seen in the 2021 improvements by UNSW CSE.
- ***autotest* has incorrect, outdated and insufficient documentation** - Although the mistakes in documentation are relatively minor such as typos, the insufficient and outdated information has been difficult to rectify. This has hindered the ability of contributors to understand how *autotest* works at a deeper level and obstructed first-time users from utilising *autotest* to it's fullest ability.
- ***autotest* was not designed for security which is of concern in modern times** - When code testing foreign code as a result of marking, the current iteration of *autotest* has no in-built protections against potentially malicious code such as deletion of system files (eg. "*sudo rm -rf /*"). This can be alleviated via external means such as limiting permissions for the user executing *autotest* but this is only a short term solution.
- ***autotest* has performance issues as a result of it's original design** - *autotest* was not originally intended to be utilised at the scale as it is today. As such, it's most significant performance issue is the single-threaded nature of *autotest* where there is no accepted implementation to execute unrelated tests in parallel [14].

3.2 check50

check50 is a software package designed by Chad Sharp and others at Harvard University to fulfil the purpose of an automatic code testing tool. Similar to *auotest*, *check50* which was first introduced in 2012 has also become the standard code testing tool for the *CS50: Introduction to Computer Science* course at Harvard University.

3.2.1 Implementation & Usage Overview

check50 is a tool that primarily follows the *external program side-effect comparison* to automatically determine the correctness of a tested program which can be seen in Figure 3.9. *check50* is written in Python and receives active maintenance and development efforts via a private repository [10].

Figure 3.9: check50 High-level Architecture

In terms of usage, *check50* provides a simple framework for defining the test cases as well as an easy-to-use API that abstracts common tasks such as compiling and running away from the user. In line with the goals of *check50*, the creation of test cases known as "checks" is simplified into a chain of functions in Python similar to that seen in functional languages such as Haskell. YAML checks can also be defined but are converted into Python automatically on execution of these checks by *check50*.

```
import check50
import check50.c

@check50.check()
def exists():
"""hello.c exists"""
check50.exists("hello.c")

@check50.check(exists)
def compiles():
"""hello.c compiles"""
check50.c.compile("hello.c", lcs50=True)

@check50.check(compiles)
def emma():
"""responds to name Emma"""
check50.run("./hello").stdin("Emma").stdout("Emma").exit()
```

Figure 3.10: check50 Example Test Cases (Checks)

Files containing these checks (Figure 3.10) can be saved to a directory locally or online to be executed as a "slug" which is the relative path to the directory containing checks. Since *check50* is officially recommended to be installed as a pip package, execution of *check50* on a slug is relatively simple as seen in Figure 3.11 [12]. It is also observed that upon execution of *check50* with a correct *hello.c*, the tool reports to the user that the provided checks have passed. In this particular case, the tested program has successfully completed all checks as defined.

```
$ check50 <path_to_example_checks>
:) hello.c exists
:) hello.c compiles
:) responds to name Emma
```

Figure 3.11: check50 execution on correct program

In the event *check50* is executed with a flawed *hello.c*, *check50* will report to the user on the failure of a test case whether it is a mismatch in side-effects in the case of Figure 3.12 or other cause such as compilation failure. *check50* supports the provision of a hint string in the failure of the test case but this must be manually set by the test writer and may potentially distract from the issue if the hint is not appropriate.

```
$ check50 <path_to_example_checks>
:) hello.c exists
:) hello.c compiles
:( responds to name Emma
        expected "Emma\n", not "emma\n"
        <hint_string_here>
```

Figure 3.12: check50 execution on incorrect program

Similarly to *autotest*, to ensure a test program's full compliance to a given specification with *check50*, sufficient checks to cover all edge cases and standard execution will be required which can be a challenge depending on the complexity of the specification. However, evaluation of checks for a specification is out of scope for *check50*.

3.2.2 Benefits

Further analysis of *check50* can provide the following list of benefits for this thesis as an inspiration:

- ***check50* test cases or checks are very easy to create** - There are many examples of checks that are available publicly and the difficulty of creating checks for basic programs is significantly lower than *autotest* [13].
- ***check50* is theoretically able to support any programming language** - This assists greatly in the usability of *check50* by courses that do not share an identical programming language as *check50* is declared to be theoretically language-agnostic assuming the user implements the necessary tooling for the language such as the framework and API.
- ***check50* maintains extensive and up-to-date documentation** - This is a significant benefit as the extensive documentation supports users in their understanding on the more confusing components of *check50* which has made usage of the tool and code contributions much easier.
- ***check50* has a module that converts *check50* output into a HTML page that improves readability of results for users who may find terminal output difficult to read** - This is one of the most defining components of *check50* in the benefit of this thesis as it improves accessibility of the tool to students who may have difficulties with sight.
- ***check50* implements containerisation of checks as a security measure against potentially malicious code and can be configured to run both remotely or locally** - This is an important feature that offers significant long term advantages such as infrastructure safety and scalability by keeping infrastructure secure and permitting users to run checks locally respectively. As such, it is a significant advantage that *check50* has over *autotest*.
- ***check50* supports concurrent execution of tests** - Concurrent execution of unrelated tests takes advantage of the full resources offered by a computer to evaluate all checks in a minimal amount of time which increases performance and efficiency. As an added benefit, *check50* checks have

the ability to define an execution order or dependency graph which ensures that certain checks such as tests on the existence of certain files and compilation are executed first before checks that depend on them.

3.2.3 Limitations

Despite the benefits of *check50*, there are unfortunately some limitations that make *check50* nonviable for immediate use and have motivated this thesis:

Assume *harnessing* to be programs and tooling that is specifically created for the purposes of creating side-effects or other usable information from programs that do not generate such information alone. An example of *harnessing* would be a test program that outputs the return value of a function that is not exposed to the user under normal operating conditions.

- ***check50* has an emphasis on the easy creation of tests which has resulted in larger amounts of abstractions and complicated the execution of complex programs** - This could be mitigated by harnessing the complex programs to work with *check50* but this is a short term solution and could involve significant work by the user to implement.
- ***check50* has no official support for languages outside of C, Python and Python Flask** - Although support can be implemented for different languages, there is likely a non-trivial amount of work that is necessary with no guarantee that the implementation by the user will be correct. This could once again be mitigated by harnessing but the same flaw of being a short term solution and potentially requiring significant work remains.

3.3 gtest

gtest is a testing and mocking framework originally designed by Google to fulfil the purpose of testing Google's internal C++ projects. *gtest* has since been released to the public in 2008 and has become one of the most popular C++ testing tools that is publicly available as open source software.

3.3.1 Implementation & Usage Overview

gtest is a tool that primarily follows the *internal program unit testing* methodology to automatically determine the correctness of a tested program which can be seen in Figure 3.13. It could theoretically support *external program side-effect comparison* but it would require significant harnessing efforts that may not carry over for different specifications. *gtest* is written in C++ and receives active maintenance and development efforts by both Google and the open source software community.

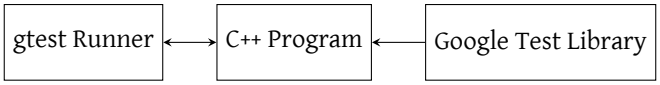

Figure 3.13: gtest High-level Architecture

In terms of usage, *gtest* approaches defining of test cases in a unique way in comparison to *autotest* or *check50*. In particular, test cases are able to specifically target internal components of a tested program such as functions without harnessing unlike *autotest* and *check50* that can only target the tested program as a whole without the use of harnessing.

```
// test.h
#include "is_prime.h"
#include <gtest/gtest.h>

TEST(IsPrimeTest, Positive) {
        EXPECT_FALSE(IsPrime(4));
        EXPECT_TRUE(IsPrime(5));
        EXPECT_FALSE(IsPrime(6));
        EXPECT_TRUE(IsPrime(23));
}
```

Figure 3.14: gtest Example Test Cases

Files containing these test case specifications (Figure 3.14) can be saved to a directory and executed via a *gtest* runner which will automatically collect all tests from linked files when compiled and run (Figure 3.15):

```
// main_test.cc
#include <gtest/gtest.h>

#include "test.h"
// link all other test specification files here

int main(int argc, char **argv) {
        testing::InitGoogleTest(&argc, argv);
        return RUN_ALL_TESTS();
}
```

Figure 3.15: gtest runner Source Code

```
$ ./main_test
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from IsPrimeTest
[ RUN      ] IsPrimeTest.Positive
[       OK ] IsPrimeTest.Positive (0 ms)
[----------] 1 test from IsPrimeTest (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (0 ms total)
[  PASSED  ] 1 test.
```

Figure 3.16: gtest runner execution on correct program

```
$ ./main_test
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from IsPrimeTest
[ RUN      ] IsPrimeTest.Positive
main_test.cc:17: Failure
Value of: IsPrime(23)
Actual: false
Expected: true
[  FAILED  ] IsPrimeTest.Positive (0 ms)
[----------] 1 test from IsPrimeTest (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (0 ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] IsPrimeTest.Positive

1 FAILED TEST
```

Figure 3.17: gtest runner execution on incorrect program

Upon execution of the *gtest runner* with a correct *is_prime.cc* and *is_prime.h*, the output shown in Figure 3.16 is expected which informs the user that the provided test cases have passed. In this particular case, all assertions of the *isPrimeTest* test case have passed.

In the event *gtest runner* is executed with a flawed *is_prime.cc* or *is_prime.h*, *gtest* will report to the user on the cause of the failure of a test case such as a deviation from expected behaviour as seen in Figure 3.17.

To ensure a test program's full compliance to a given specification with *gtest*, sufficient test cases to cover all edge cases and standard execution will be required which can be a challenge depending on the complexity of the specification. However, checks on the creation of tests is out of scope for *gtest*.

3.3.2 Benefits

Further analysis of *gtest* can provide the following list of benefits for this thesis as an inspiration:

- ***gtest* maintains extensive and up-to-date documentation with a large community of users** - This is a significant benefit as *gtest* as a whole can be perceived as complicated and confusing but many community made tutorials and guides exist on the internet to solve most issues.
- ***gtest* supports concurrent execution of tests** - Concurrent execution of unrelated tests takes advantage of the full resources offered by a computer to evaluate all test cases in a minimal amount of time which increases performance and efficiency. However, *gtest* offers no methods to guarantee that tests will be executed in a certain order unlike *check50*.

3.3.3 Limitations

Despite the benefits of *gtest*, there are unfortunately some limitations that make *gtest* unsuitable for use:

- ***gtest* is not designed to test whole programs and can have trouble detecting side-effects such as the existence of files without significant harnessing** - This is expected for software that implements *internal program unit testing* as a methodology but still remains a significant limitation of *gtest* for use as an automated code testing tool.
- ***gtest* does not sufficiently support test execution environment configuration easily** - *Internal program unit testing* frameworks are not expected to manage an external test environment but the difficulty and lack of configuration within *gtest* could prevent the testing of programs that may rely on a specialised environment.
- ***gtest* is generally difficult to use as an automated code testing tool** - *gtest* is difficult to configure and utilise for most users and far better tools such as *autotest* and *check50* exist for a baseline design.

3.4 Other & Historic Works

See these other and historic works that have had minimal but relevant impact on this thesis as an exploration on the progression of each aforementioned automated code testing methodology throughout history: **JUnit** (Internal Program Unit Testing) [15], **BAGS** (Primitive External Program Side-effect Comparison) [16], **Kassandra** (External Program Side-effect Comparison) [17] and **TRY** (External Program Side-effect Comparison) [18].

Approach

4.1 Requirements

The initial problem statement of introducing a replacement automated code testing tool is quite broad, and the required design will depend highly on the chosen automated testing approach. In particular, it would be impossible to have one solution that will immediately satisfy the demands of all users.

Therefore it is important to lay out the desired requirements and design goals for an ideal solution before choosing a proposed approach for the implementation.

4.1.1 Design Goals

In accordance with the thesis aim, the design goals for the project will emphasise the following properties:

- **Accessibility** - The design should allow new users to easily integrate this automated code testing tool into their coding assessments.
- **Familiarity** - Existing users of *autotest* should not observe the proposed solution and it's usage to be completely independent of the former.
- **Performance & Efficiency** - The proposed implementation should be observed to have comparable if not better performance than the existing *autotest*. Benchmarking of *autotest* and the proposed implementation will be performed to verify this goal.
- **Maintainability** - The design should be adequately documented and support an ease of maintainability and possible extension of features via the appropriate architectural decisions.
- **Security** - The design should implement or consider the addition of security as it has been a concern for various implementations of an *automated code testing tool* in the protection of user hardware against potentially malicious code.

It is noted that most of these goals are qualitative and can not be quantitatively measured outside of reader observation. The thesis will attempt to quantify these goals within the properties of each pro-

posed design approach but it will still be dependent on the reader's interpretation.

4.2 Design

From the evaluation of the related works, the thesis evaluates two potential design approaches to the solution that fulfils the design goals outlined above, one of these two approaches is then chosen for the project. Implementation details for each design approach are important but defining a "target" architecture design is considered to be of greater importance as it cannot be easily changed once development begins.

4.2.1 check50 Approach

In this approach, the architectural design will be similar to that of *check50* as shown in Figure 4.18 including the design for definition of test cases and as such, the methodology to determine correctness of a tested program will be *external program side-effect comparison*.

Figure 4.18: check50 Approach - High-level Architecture

This approach allows the project to utilise *check50* as a "baseline" for both performance and correctness testing of the solution which will aid in the development process by being a reference solution that can be compared with.

To further develop the approach, a solution implementing this approach would work like so (at a high level):

1. The *Tester* will read in the appropriate *test cases* and any other relevant information to parse into individual test cases with the relevant parameters set (timeout period, test input, expected output etc).
2. The *Tester* will "spin up"/initialise a configured amount of *Test Runner* processes which will each individually consume a test case for independent execution.
3. When the *Test Runner* completes execution of a test which may involve an external *Program*, the *Test Runner* will return it's result back to the *Tester* and consume another test for execution if one exists. If no more tests exist, the waiting *Test Runner* will "spin down"/shut down.
4. Throughout steps 2 and 3, the *Tester* will report to the user on test execution progress but once all tests results have been gathered, a report will be generated for the user on test successes and failures.

4.2.2 autotest Approach

In this approach, the architecture is similar to the *check50 Approach* in Section 4.2.1 but with the distinct change that the design of the test case definitions will be that of the existing *autotest*.

Figure 4.19: autotest Approach - High-level Architecture

The original architecture of *autotest* has been abandoned in this approach as the *check50* architecture is more suitable to the aforementioned design goals.

Thus, the solution implementing this approach will be identical to the *check50 approach* and has been omitted as it is available in Section 4.2.1.

4.2.3 Chosen Approach

Evaluation of each approach was made by considering fulfilment of the design goals aforementioned above. Table 2.2 lists if each approach satisfies each design requirement.

Requirement	**check50 Approach**	**autotest Approach**
Accessibility	Yes (by design)	Yes (by design)
Familiarity	No	Yes
Performance & Efficiency	Unknown	Unknown
Maintainability	Yes (Technical Debt Management)	Yes (Technical Debt Management)
Security	Planned	Planned

Table 4.3: Evaluation of the *check50* and *autotest* approach against the stated requirements

As seen from Table 4.3, neither approach completely matches the requirements laid out, however the *autotest approach* is close and will serve as an appropriate start for the project over the alternative.

It is noted that some components of the test case definition design and backend design of *check50* could be considered to be superior to that of *autotest*. As these features are implementations details and are not greatly affected in respect to the chosen architecture, decisions on whether these features will be included in the solution can be deferred to a later time in the development period.

It is also noted that the chosen approach has changed since Seminar A due to the inclusion of security in the design goals which had been justified by Seminar A feedback [11].

Implementation

5.1 Core Architecture

5.2 Modules (or Modular Design)

5.2.1 Core Module

5.2.2 Parser Module

5.2.3 Test Case Definition

5.2.4 Test Case Scheduler Module

5.2.5 Test Case Runner Module

5.2.6 Sandbox Container

5.2.7 Test Case Canonical Difference Module

5.2.8 Test Case Report Module

Evaluation

Future Work

Conclusion

The above thesis outlines the possible methodologies of automated code testing, it's usage in relevant existing works and proposes a design approach for achieving the aim of this thesis which is to create an automated code testing tool in respect to the given design goals in Section 4.1. Preliminary work has been completed and a plan for the implementation of the proposed design has been provided to demonstrate the feasibility of this thesis.

Bibliography

[1] Andrew Taylor, 'autotest', URL: https://github.com/COMP1511UNSW/autotest

[2] Ward Cunningham, 'The WyCash Portfolio Management System' in *Addendum to the Proceedings on Object-Oriented Programming Systems, Languages, and Applications (addendum)* (ACM, 1992) 29

[3] Tom Edith, Aybüke Aurum and Richard Vidgen, 'An Exploration of Technical Debt' (2013) 86(6) *The Journal of systems and software 1498*

[4] Andrew Taylor et al., 'autotest issues', URL: https://github.com/COMP1511UNSW/autotest/issues

[5] Eric Allman, 'Managing Technical Debt' (2012) 55(5) *Communications of the ACM* 50

[6] David Jackson, 'Using Software Tools to Automate the Assessment of Student Programs' (1991) 17(2) *Computers and education* 133

[7] Jack Hollingsworth, 'Automatic Graders for Programming Classes' (1960) 3(10) *Communications of the ACM 528*

[8] Chad Sharp et al, 'An Open-Source, API-Based Framework for Assessing the Correctness of Code in CS50' in *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education* (ACM, 2020) 487

[9] Google, 'GoogleTest', URL: https://github.com/google/googletest

[10] Chad Sharp et al, 'check50', URL: https://github.com/cs50/check50

[11] Andrew Taylor et al, Conversation at conclusion of Thesis A seminar for 'A Testing Tool for Introductory Programming Courses', 22 April

[12] Chad Sharp et al, 'check50 Documentation', URL: https://cs50.readthedocs.io/projects/check50/en/latest/

[13] Chad Sharp et al, 'CS50 Problem Exercise Repository', URL: https://github.com/cs50/problems

[14] Kyu-Sang Kim, 'autotest test case execution parallelisation proposal', URL: https://github.com/COMP1511UNSW/autotest/pull/28

[15] Kent Beck et al, 'JUnit', URL: https://github.com/junit-team/junit5

[16] J Hext and J Winings, 'An Automatic Grading Scheme for Simple Programming Exercises' (1969) 12(5) *Communications of the ACM* 272

[17] Urs von Matt, 'Kassandra' (1994) 22(1) *SIGCUE bulletin* 26

[18] Kenneth A Reek, 'The TRY System -or- How to Avoid Testing Student Programs' (1989) 21(1) *ACM SIGCSE Bulletin* 112

[19] Red Hat, 'What is containerization?', URL: https://www.redhat.com/en/topics/cloud-native-apps/what-is-containerization

[20] man-pages Project, 'namespaces(7)', URL: https://man7.org/linux/man-pages/man7/namespaces.7.html