University POLITEHNICA of Bucharest

Automatic Control and Computers Faculty,
Computer Science and Engineering Department

BACHELOR THESIS

Portability and Interoperability Features in Robocheck

Scientific Adviser:
Adrian-Răzvan Deaconescu

Author:
Laura-Mihaela Vasilescu

Bucharest, 2012

When the wind of change blows, some people build walls, others build windmills.

Abstract

Code quality, layout and structuring are of paramount importance in developer communities. Most of the time, a project can be labeled as successful or not depending on how easy can someone delve into the project. Good quality code should be well commented, well designed and easy to browse and understand. A programmer needs time to improve their coding skills. It is an entire process that needs guidance and feedback in order to enhance development techniques.

Robocheck is a framework designed to automatically generate feedback about the quality of a project implementation. It was created for a didactic purpose, but it can also be used to elicit the quality of the code of the application.

This project emerged from the need to automatically generate feedback for the assignments proposed in the Operating Systems class. Robocheck didn't have the maturity of a stable tool and could only run on the Linux platform. The initial functionalities of the Robocheck were improved in order to provide portability and interoperability for the framework. Today, Robocheck can run on the Windows platform and it is easy to integrate with other frameworks.

Keywords: Robocheck, Windows, configuration, interoperability

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context and Motivation

Code quality, layout and structuring are of paramount importance in developer communities. Most of the time, a project can be labeled as successful or not depending on how easy can someone delve into the project. Good quality code should be well commented, well designed and easy to browse and understand. [1]

One of the best periods for acquiring and improving code writing skills is while studying. During high-school or during their time in universities, students gradually learn how to improve their code quality. They need to be able to understand how poorly written code can affect their projects. Such things are learned in time and by gaining experience; reading an article or book on coding style is not enough for actually using the knowledge in practice.

Students need guidance in order to fully understand the importance of code quality. They need feedback and code review to be able to improve their skills. Their assignments must be assessed and reviewed while considering two criteria: code functionality and code quality. It is not enough for an assignment to match the requested functionality if it has an unintelligible structure and doesn't cover all possible corner cases and errors.

In order to achieve balance between functionality and quality, teaching assistants should give complex and punctual review for each student according to the issues encountered.

Reviewing assignments can take quite some time for a teaching assistant. Consider this example: at University POLITEHNICA of Bucharest[1], in the Computer Science and Engineering field, there are approximately 350 students enrolled in a year of study. Reviewing just one of their assignments by only one assistant (with an average of 20 minutes / assignment) would take 116 hours and 40 minutes. That means 5 full days (with no sleeping). One could consider increasing the number of assistants, such that every assistant would have to review 30 assignments (for 2 half-groups); in this case the assignment review process would require 11 assistants and 10 hours for every assistant; however reviews and penalties would not be uniform for all students.

Consequently a need arises to create tools that are able to automatically review student assignments. There already exist tools that can assess code quality. Some of them are focused on code modularity, some of them verify memory inconsistencies, but there is no tool that aggregates all of the above capabilities. Although the idea of developing a tool to aggregate code quality reviewing utilities was considered many years ago, it was only in the recent years that actual effort has been made.

[1] http://www.upb.ro

Robocheck[1] has emerged to address the need for an automatic tool for code quality validation in the Operating Systems class[2] in University POLITEHNICA of Bucharest.

In the Operating Systems class, assignments should be able to run on both Linux and Windows, or only on a single one of them. In the initial version, Robocheck was designed to run only on Linux. To solve the problem of automatic validation code quality for all Operating Systems assignments, Robocheck is required to also run on Windows too.

1.2 History of Robocheck

Robocheck was started in 2003 by Octavian Purdilă (Tavi) to address the need for an automatic tool that could test code quality of student assignments. At that time, Robocheck was tracing memory leaks and used a hand-written module. The goal was to implement this solution in the Operating Systems course. However, due to lack of time and community support, the project was put on hold.

Figure 1.1: Robocheck Logo

The project was revived in 2009 by Andrei Buhaiu when he proposed Robocheck as a summer project in ROSEdu Summer of Code (RSoC)[3]. Five students were selected and they started to redesign the project. During RSoC, they focused on designing a modular architecture for Robocheck in order to aggregate tools used for code quality validation. RSoC lasted only one month and they didn't have too much time to actually implement the new features; as such they insisted on having a complete and extensible design at the end of the program.

The need for a tool with Robocheck capabilities persisted. In 2011, Cezar Socoteanu [2] and Iulia Bolcu [3] made their diploma project for actually implement Robocheck consistently with the design proposed and specified in 2009. Cezar possessed initial experience from the RSoC project. At the end of their project, Robocheck was able to run on Linux platforms.

1.3 Limitations in the Initial Version of Robocheck

Although Robocheck is a useful tool, it has some shortcomings. This work aims to alleviate these shortcomings and push Robocheck as a ready–to–use solution for a variety of classes. This section presents some of the caveats of the initial version of Robocheck.

1.3.1 Platform Dependency Issues

Robocheck aggregates several tools and extracts reported errors. Some tools are cross-platform, while others are not. Because of that, the framework was limited to running only on the Linux platform. In theory, running Robocheck only with cross-platform tools would have been sufficient to run on a Windows platform. Nevertheless, excluding non-cross-platform tools from

[1] http://ixlabs.cs.pub.ro/redmine/projects/robocheck/wiki
[2] http://elf.cs.pub.ro/so
[3] http://soc.rosedu.org

Robocheck limits it capabilities. Some types of errors can not be intercepted because there isn't a tool available to extract them. In order to minimize the impact of using only non-cross-platform tools, other tools should be integrated to provide a similar functionality and capabilities for the framework running on the Windows platform.

On the other hand, on Windows platforms, default compilers (for example `cl`) don't support for C99[1] standard and the initial code wasn't compatible with C89[2] standard; it wouldn't compile on a Windows environment. In order to facilitate platform independence, the code needed to be compatible with C89. A good part of the code needed rewriting to confer compatibility and integration to C89 standard.

Another thing that stood in the way of running Robocheck on Windows was the use of system calls. In Linux, system calls are mostly POSIX compatible, so the code can be easily compiled and run on a Unix platform (like MAC OS X). Unfortunately, on Windows, system calls are not POSIX compatible. Calling such a function (a system call on Linux) in Windows would cause a linker failure: *" undefined reference to ... "*. Creating and using a set of wrappers around those functions was a good solution to confer and support portability.

1.3.2 Non-intuitive Configuration of Penalty Facility

Robocheck offers a penalty facility (decrementing the final grade) for errors discovered. This facility was design to be modular and configurable through the use of an `XML`[3] file. Through the use of the `XML` file, teaching assistants can define which tools to use, what errors to trace and values for the corresponding penalties. Configuration through the `XML` file made Robocheck very extensible. Each teaching assistant could make his own decisions about what type of errors should be downgraded in his class and how much value should be penalized from the final grade of the assignment.

However, changing the `XML` configuration file is not an easy step. There is no documentation about the structure of the `XML` file and browsing through an existing `XML` file offers little information to an non-developer. Adding new errors for tracing was mostly guessed based on the examples for already traced errors.

The usability of an application is a very important aspect. Changing running parameters of a framework should be very intuitive for the user. To easily change the the configuration file, a new tool, capable of automatically generate `XML` files, was requested.

1.3.3 Non-Standard Output Format

At the end of running Robocheck, the framework provides an output with all errors and corresponding penalties discovered. The assistant gets the output and provides it as feedback to the student. One disadvantage was that each integrated tool in the framework was using its own format for writing errors to standard output. The only consistent format was the corresponding penalty for each error.

Having a non-standard format for error output made Robocheck difficult to integrate with other tools already used in classes to check functionality and correctness of the application. The assistant was limited to viewing and interpreting the output manually.

The uniformity gap in errors reported was also reflected as a bug in the framework. Due to the use of multiple tools for errors detecting, one error could be reported by multiple tools. The problem wasn't only the duplicate output, but also the duplicate penalty applied to the same

[1] http://www.iso-9899.info/wiki/The_Standard#C99
[2] http://www.iso-9899.info/wiki/The_Standard#C89_.2F_C90_.2F_C95
[3] http://www.w3.org/XML/

error. To solve this issue, Robocheck required a mechanism to differentiate between such errors and report them only once.

1.3.4 Running Robocheck as an Application

In the previous release, running Robocheck was possible only from the building directory. The compiler script used relative paths to describe all object files and libraries dependencies.

Even if the code compiled successfully, the application couldn't be run from a different directory. Putting Robocheck compilation folder into the system `PATH` (an environment variable specifying a set of directories where executable programs are located) variable wouldn't be sufficient. The code also relied on hard-coded values with relative paths that made Robocheck unable to run from a different directory.

1.3.5 Unexpected Parsing Errors

A parsers is used to extract and interpret tool reported errors. One shortcoming was the fact that the parser had little information on how to interpret unexpected input. Most of the time, the program crashes.

The parser needed to be rewritten to cover all use-cases scenarios and to properly treat all corner-cases and possible errors.

1.4 Project Proposal

This project started from the need of testing code quality for the assignments for the Windows platform in Operating Systems class, but will be, hopefully, used in other classes in the faculty.

At the beginning, porting the framework to run on the Windows platform was the prime objective of the project. After identifying different types of issues in the initial Robocheck implementation, the main goal for the project became transforming Robocheck into a usable tool for the next year Operating Systems class. Solving usability issues in Robocheck (even on Linux) became a track in the project. This project aims to ensure portability for Windows platform and interoperability and integration with tools for correctness checking.

Some parts of the code must be rewritten in order to have stability, portability and interoperability of the Robocheck framework. Another important step in goal achieving is integrating **Dr. Memory**[1] in Robocheck to provide similar functionality on Windows platform and integrate the framework in vmchecker[2] for making a complete solution of assignments testing, usable in next year classes.

[1] http://www.drmemory.org
[2] https://github.com/vmchecker

Chapter 2

Tools of the Trade

This chapter elaborates on the tools that interact some way or another with the current Robocheck implementation. The framework provides a context for each integrated tool to run into.

Each integrated tool has its own purpose and can identify different types of errors.

- valgrind: used for memory corruption detection
- helgrind: used to identify race conditions
- simian: used to identify code duplication
- splint: does a static analysis of the source code in order to find code vulnerabilities and mistakes
- sparse: responsible for semantic parsing
- Dr. Memory: used for memory corruption detection; can run both on Linux and Windows

Integrating these tools in Robocheck increases the capabilities of the framework and makes it a powerful tool that can perform a complex analysis of the submitted assignments.

2.1 Valgrind

Valgrind[1] is an Open Source framework used for dynamic analysis that can be run on different architectures, powered by Linux kernel based operating systems (such us different Linux distributions or Android). The framework is used mostly for profiling and debugging Linux applications. [4]

The source code[2] of **Valgrind** is freely available to download from its `Subversion`[3] repository, because the tool is an open source project.

The Valgrind framework offers different tools with diverse purposes and capabilities according to one's needs:

- **Memcheck**: used to detect memory-management problems
- **Cachegrind**: used as a cache profiler

[1] http://valgrind.org
[2] svn://svn.valgrind.org/valgrind/trunk
[3] http://subversion.tigris.org/

- **Callgrind**: adds information about callgraphs to **Cachegrind**
- **Massif**: used as a heap profiler
- **Helgrind**: used as a thread debugger to find races
- **DRD**: used to detect errors in multi-threaded programs

Running **Valgrind** with no parameters does in fact call the **Memcheck**[1] tool. To be able to run other tools, one must specify the wanted tools as command line parameters.

Memcheck can detect invalid memory accesses (and can distinguish between access to unallocated memory and recently freed areas), memory leaks, double free corruptions and invalid freed addresses, overlapping buffers (when memory copying functions are used) and uninitialized variables usage. This tool is a very powerful instrument and can show the line number that produced the error. To be able to give additional information, the program must be compiled with debugging symbols.

The main disadvantage of this tool is the overhead induced by the dynamic analysis. Statistically, an application runs 30% slower than usually when is coupled with this tool. Because Robocheck runs each tool individually, **valgrind** overhead will be added to Robocheck. If the errors tracked by **valgrind** could be also tracked by another tool, then the time spent by Robocheck will decrease.

This tool is the one responsible in Robocheck with errors related with invalid memory accesses or non-release resources.

2.2 Helgrind

Helgrind[2] is one of the instruments offered by the **Valgrind** tool suite. The tool is capable of identifying and reporting race conditions in multithreaded programs.

Helgrind portability is limited by the architectures and operating systems supported by **Valgrind**.

The concepts applied by **Helgrind** are simple: it traces all memory locations used by more than one thread and checks if the locks had been acquired everywhere in the same order to prevent race conditions. It also runs a topological sorting on the locks used by the application to determine if threads would run into race conditions at some point.

This tool is used by Robocheck in multithreaded assignments to be able to identify deadlocks and race conditions.

2.3 Simian

Simian[3] is a similarity analyser tool. It is able to identify code duplication (in a variety of languages) and even duplication for plain text files.

The main purpose of the tool is to help development teams involved in large projects trace code duplication in a re-factoring stage of the project. Furthermore, it can also be used in searching for plagiarism. By using this tool, one can improve the code quality and modularization of the application one is working on.

[1]http://valgrind.org/info/tools.html#memcheck
[2]http://valgrind.org/info/tools.html#helgrind
[3]http://www.harukizaemon.com/simian

Although **Simian** is not an Open Source application, the developers offer, free of charge, a JAR library for non-commercial and Open Source use.`Java`[1] is a cross-platform programming language with many build-in libraries, offering operating system independence. As **Simian** is written in Java, it also benefits from the programming language's portability and it can be run in any environment.

By integrating this tool, Robocheck can provide feedback about the code modularity of the assignment and can indicate which parts of the code can be restructured as functions.

2.4 Splint

Splint[2] is an application used to trace errors in C programs. The analysis is statically performed in order to inspect the source code for vulnerabilities and coding mistakes.

Splint was developed by the Secure Programming Group at the *University of Virginia, Department of Computer Science*[3]. The development is done in-house and doesn't have a public repository, but the source code is published periodically with each stable release. To install the application, one can either compile it from the source code or install it directly from the binaries available on the web site. Binaries for Linux x86, FreeBSD, OS/2, Solaris and Win32 are provided.

Splint doesn't run the application under test in order to trace errors. The analysis is based only on the source files of the application and can detect mistakes like null dereferences, uninitialized variables usage, invalid memory access, buffer overflows, usage of non-static context in a static context etc.

2.5 Sparse

Sparse[4] is a semantic parser and was written by Linus Torvalds. Initially, his goal was to write a tool that was capable to identify mixed usage of pointers from user space with pointers from kernel space.

Sparse is an Open Source project maintained by the Linux Kernel community. The source code[5] is stored using Git[6] and can be cloned by anyone, since the project is open source.

Nowadays, **Sparse** does a complex analysis of code quality: it identifies the mixing tabs and spaces in indentation, trailing whitespace and much more.

Unfortunately, **Sparse** is only available on the Linux Platform.

2.6 Dr. Memory

Dr. Memory[7] is a memory checking and debugging tool, similar to **Valgrind**. It was developed by *Google* in collaboration with *Massachusetts Institute of Technology*.

[1] http://java.oracle.com
[2] http://www.splint.org
[3] http://www.cs.virginia.edu
[4] https://sparse.wiki.kernel.org
[5] git://git.kernel.org/pub/scm/devel/sparse/sparse.git
[6] http://git-scm.com
[7] http://www.drmemory.org

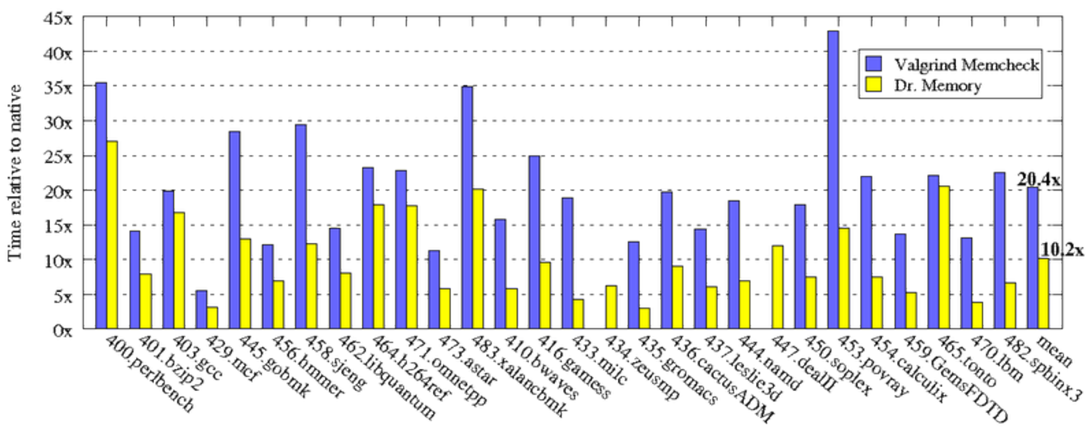


Figure 2.1: The Performance of Dr.Memory compared to Valgrind Memcheck [5]

Dr. Memory is an Open Source application that can be run on both Linux and Windows platforms and comes as an alternative to old solutions, which introduced large overhead. Aldo it can be run on `IA-32` and `x86-64` architectures, currently the main focus is to develop a powerful tool for `32-bits` architectures;

According to Derek Bruening and Qin Zhao [5], **Dr. Memory** is much faster than **Valgrind** and the resulting overhead is very small. The main reason of integrating this tool into Robocheck is that **Dr. Memory** is designed to be architecture independent. Even though **Valgrind** is a popular tool and there are a lot of requests for porting it on the Windows platform, it is not at all a trivial task due to its inflexible architecture. Windows has a different management of the memory as against Linux. Valgrind was designed to run on Unix platform, and makes assumptions about how the memory is managed.

Because feedback about invalid memory accesses is very important for students, a tool similar with Valgrind, but that can also run on Windows platform, was almost mandatory. Dr. Memory performances and capabilities are a perfect match for the Robocheck requests.

2.7 vmchecker

vmchecker[1] is a framework for automated home assignment grading and it was developed by a group of students from *University POLITEHNICA of Bucharest, Department of Computer Science*[2]. It is an Open Source project that runs on Linux. **vmchecker** provides an isolated environment for the assignments to run into. (the guest environment can be specified by a Vmware[3] machine, LXC[4] or KVM[5] that runs any operating system). [6]

The source code[6] is version in a **Git** repository and it is an open source project.

vmchecker runs on two different machines:

- the storer
- the tester

[1] https://github.com/vmchecker/vmchecker
[2] https://csite.cs.pub.ro
[3] http://www.vmware.com
[4] http://lxc.sourceforge.net
[5] http://www.linux-kvm.org
[6] git://github.com/vmchecker/vmchecker.git

The storer is responsible for assignment storage and evaluation for the students. The Teaching Assistants interact only with this module. Furthermore, the environment configuration is also handled by this module, specifying the testing suite and the guest operating system.

The tester is responsible for starting the isolated environment and running the test suite in order to evaluate the assignment. The output is stored in a file named `grade.vmr` that is copied on the storer machine where the Teaching Assistants can review it.

Chapter 3

Overview of The Robocheck Framework

In order to ensure portability and interoperability for the new version of Robocheck, the framework architecture has been updated. The basic core remained intact, but new modules were added and existing modules required modifications.

3.1 Description of the Initial Robocheck Architecture

Robocheck is designed as a modular framework used for running other quality checking tools. It has a powerful mechanism for configuring these tools and a mechanism for defining and computiing penalties for the errors traced. A description of the initial architecture can be viewed in Figure 3.1.

The framework is configured through a specialized XML configuration file, that is read each time the framework is run. This file describes the behavior of the application. Teaching assistants may define what and how tools should be run, which errors to be reported and what penalties should be applied.

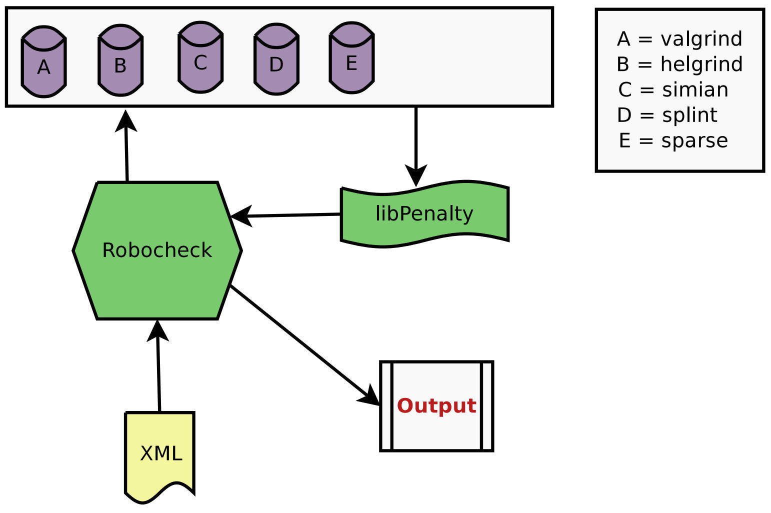


Figure 3.1: Initial Robocheck Architecture

A distinct module of the framework is the penalty module. The penalty module is designed as a standalone library, in order for other applications to easily interrogate it. The library is aware of errors identifiable by Robocheck. Like the framework itself, the penalty library parses the configuration XML file and sets penalties for different types of errors. It is able to differentiate among errors that need to be penalized at each appearance and errors that need to be penalized only once (no matter the number of appearances).

In order to provide modularity Robocheck can run only some of the tools integrated into the framework; some tools may not be available (or may not be installed) on a possible target environment. To separate between the core of the Robocheck framework and code used auxiliary to run a specific tool, each tool is defined in its own shared library.

With this approach the size of the final library of the framework is independent of the number of running and configured tools in the system. It also provides flexibility by allowing the removal or addition of one tool from or into the framework without recompiling the code.

Integration of a new tool is easily achievable by writing a new module as described in Section 3.4. When running a tool, Robocheck is in fact loading the library of the corresponding tool, runs it and gets its output. The library tool is responsible for parsing the output and registering errors into the penalty library.

In the previous version, Robocheck was capable of running five tools:

- **valgrind** (described in Section 2.1)
- **helgrind** (described in Section 2.2)
- **simian** (described in Section 2.3)
- **splint** (described in Section 2.4)
- **sparse** (described in Section 2.5)

After running all configured tools, Robocheck would talk to the penalty library and output all identified errors to standard output. The teching assistant would then be able to interpret the output and formulate feedback to the student.

3.2 Description of the Current Robocheck Architecture

This current project aims to ensure portability and interoperability in the Robocheck framework. Although the main core architecture remained intact, small changes in the modules were required in order to implement the proposed solution. An overview of the new architecture is depicted in Figure 3.2.

A big improvement in the Robocheck architecture was adding a configuration tool, described in Chapter 7). This tool is used for updating the changing the XML file; one can easily list different parts of the XML in a user-friendly format and modify it according to needs. Changes are incremental, so the user needn't provide an extensive list of parameters when running the tool.

An important feature in the previous version of Robocheck is the ability to identify memory leaks in assignments. It was provided by **valgrind** (Section 2.1). **Valgrind** is not a portable tool and can only run on Linux. The lack of this tool on Windows is a minus for the Robocheck framework. To replace the gap, a new tool (Dr. Memory) has been integrated to run on both platforms (Linux and Windows). Even if this means to have two tools doing the same thing on the Linux platform, the performance it is not affected as memory checking tools are already quite slow when compared to the simple run of a program.

Dr. Memory (described in Section 2.6) is a memory monitoring tool that can run on Linux, Windows and Cygwin [1]; it has similar capabilities to **valgrind**. This tool was introduced to provide memory leak detection on the Windows platform, and was also included in the build for the Linux platform. The only constrains are that, for the moment, even if **Dr. Memory** is designed to run on IA-32 and X86-64 hardware, it is recommended to be installed only on 32-bits systems.

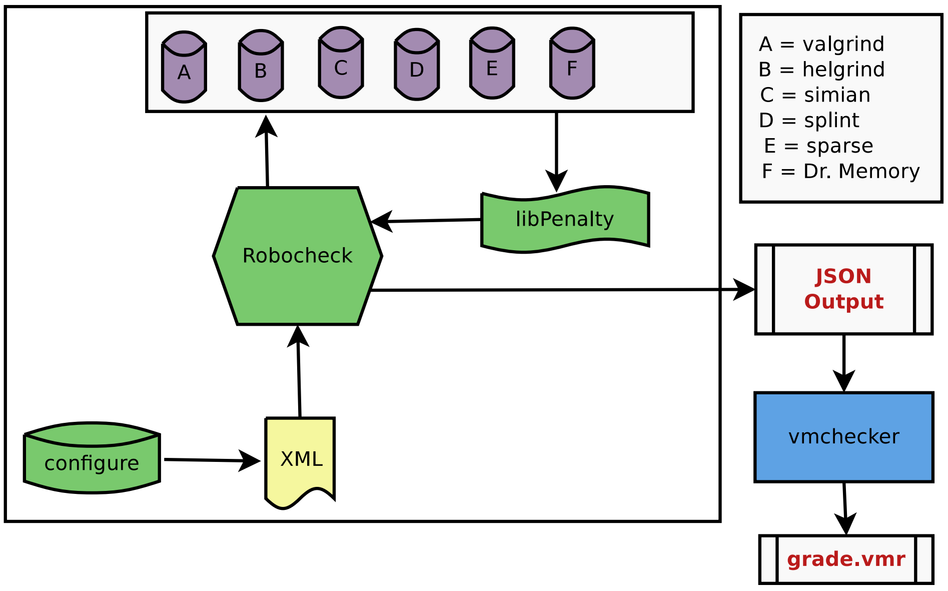


Figure 3.2: Current Robocheck Architecture

The penalty module was updated to offer support for developing a common method of error reporting. This feature was requested in order to determine duplicate errors and make the necessary changes to report them only once.

The output format is generated by the penalty module. The penalty module aggregates reported errors from tools and groups them by type. The grouping simplifies duplicate error checking and to easies penalty computation for identified errors.

The generated output was changed to provide uniformity and easy integration with other tools. In the new version, the output is exported in a JSON (*JavaScript Object Notation*) format as described in Section 6.1.

As Robocheck core provides output in a standard format (JSON), it is easy to integrate it with other tools. The first step in this direction was integrating the framework with **vmchecker** (Section 2.7). **vmchecker** throws the output in a special file named `grade.vmr` with a specific format, easy to understand and view by students. More on this topic is described in Section 6.2.

From now on, everything is described from the point of view of how things are done and how things work in the new version of the Robocheck framework.

3.3 Penalty Library in Robocheck

The Robocheck framework uses a shared library to manage and provide penalties for detected errors.

[1] http://www.cygwin.com

This library exports three functions, each with its own functionality:

- initializing;
- freeing resources;
- applying penalties.

We will discuss each each of the above functionalities in the sections below.

3.3.1 Initializing

```
int init_penalties(rbc_xml_doc);
```

The initialization of the penalty module from the user configuration file is performed by calling `init_penalties`. This is provided by the library and ensures that all the necessary information is available to Robocheck after startup.

The user can select between two ways of penalising errors of certain type: one that considers the number of reported errors in scoring and one that doesn't. In more detail, the first one amends the score for each n occurrences of that error type. Commonly, `n` is set to one, but one can also choose to decrease the score for once in n occurrences.

The other way of applying penalties is to penalize only once, no matter how many errors of that type are reported. For example, if someone used trailing whitespace, it doesn't make sense to penalize for each additional space.

3.3.2 Freeing Resources

```
void free_penalties();
```

The purpose of this function is to release all resources used by the library. This function is usually called after the library has been used (just before closing); it may also be used if one would like to clean the setup and create another one, with a different configuration.

3.3.3 Applying Penalties

```
struct rbc_out_info * apply_penalty(enum EN_err_type, int);
```

This function is responsible for transforming errors into a specific format, depending on the error type. It iss useful in avoiding duplicate errors and in having an uniform and scalable output format.

The first parameter represents the error type and the second one represents the number of the errors in the pool to aggregate. Output messages are parsed while taking into account the error type to eliminate duplicate errors from the pool. Each type of error has its own format and is independent of the tool that generated it.

This call returns the polished errors in a specific format that will be transformed in order to obtain the JSON structure. The transformation is not the responsibility of the penalty module; however, the output provided by this call is similar and very easy to transform into JSON.

3.4 Adding New Modules in Robocheck

Robocheck is designed for modularity and extensibility. Each module is defined as a shared library that is dynamically loaded upon request. In order to load a tool module, the library name must be specified in the XML configuration file.

3.4.1 Implementing a New Module

Every module integrated in Robocheck has its own implementation of how to run the tool. To provide uniformity, modules must export a `run_tool` function that runs the tool, parses its output and provides results in a specific format to the caller.

On Linux, exporting a function in a shared library [7] is done automatically, but it's not the case on Windows [8]. To ensure code portability for a module that should be able to run on both platform, the action of exporting functions needs to use a set of macros (see Listing 3.1). There is a special caution to be taken when compiling on the Windows platform: a symbol such as `DLL_EXPORTS` must be defined. This macro should be defined in the `lib/tool.h` header and the function header must be preceded by `DLL_DECLSPEC` keyword. This modification was already done and the only thing a new module developer should take in consideration is defining the symbol `DLL_EXPORTS`.

```
#ifdef _WIN32
    #ifdef DLL_EXPORTS
        #define DLL_DECLSPEC __declspec(dllexport)
    #else
        #define DLL_DECLSPEC __declspec(dllimport)
    #endif
#else
   #define DLL_DECLSPEC
#endif
```

Listing 3.1: How to Export Functions in Shared Libraries

There are two types of modules that may be integrated into Robocheck:

- static analysis modules: run tools in order to analyze source files;
- dynamic analysis modules: run tools in order to analyze executable files.

Listing 3.2 is an overview of a module's `run_tool` function.

```
#include <lib/tool.h>

struct rbc_output *
run_tool (struct rbc_input *input,
          rbc_errset_t flags,
          int *err_count)
{
    /* Actually implementation of this function */
}
```

Listing 3.2: Tool Module Implementation Structure

The first parameter stores input information. The module uses it to find paths to source files for static modules and paths to binary executable files for dynamic modules.

The second parameter is a set of flags that enable different fields from the input structure. In combination with the first one it will be able to tell what errors should be traced and what parameters should be passed to the tool.

The third parameter returns the number of identified errors. This is used in combination with the function return value in order to specify the size of an array. The function return value is an array of identified errors and it is dynamically reallocated each time a new error is identified.

Some tools may only be available on Linux. In that case, Robocheck framework will search for a file named `Makefile` to pass to `make`. Other tools may be only be available on Windows, in which case Robocheck will search for a file named `NMakefile` to pass to `nmake`.

Availability of a tool on Linux and Windows is defined by the existence of the files `Makefile` and `NMakefile` respectively. If one doesn't exist it means the module isn't available on that platform.

Chapter 4

Updating Tools in the Linux Implementation

The previous version of the Robocheck framework had some bugs and missing features in the implementation of **Splint** and **Simian** modules.

The Splint module implementation only provided support for memory leaks error detection. The problem that arose was that this code was not being called in any scenario. As in all unit tests Splint was run along with Valgrind, this bug was not obvious. The first step in solving this problem was to make the parser actually running; the second step was to extend the parser to report other errors such as assignment of signed values to an unsigned variables and unused exported variables (the lack of the static keyword) definition.

Simian needed to be changed in order to provide an uniform error reporting format, as it was reporting error in a variety of styles. It was modified to provide uniformity and easy integration with the JSON format. Another problem was that if more than one set of duplicate code was detected, the parser would crash with *Segmentation Fault*. Consequently, the parser has been improved to correctly filter the output.

Another important step was to integrate a new tool to be responsible for detection of memory related errors. The tool is **Dr. Memory** and the integration process is presented in the section 4.1.

4.1 Integration of Dr. Memory

As mentioned in section 2.6, **Dr. Memory** is a memory checking and debugging tool.

One of the important improvements for the Linux implementation in the Robocheck framework was the integration of **Dr. Memory**. This tool was requested in order to provide portability of the framework and replace the absence of **Valgrind** on the Windows platform.

Dr. Memory source code was added to Robocheck repository and the `Makefile` was updated in order to also provide compilation of this tool. A new module was added following the steps described in section 3.4.1.

After running **Dr. Memory**, a new process displays to standard error a summary of the errors traced by the tool. The output also contains the path to the file where the detailed output is stored. The parser needed to open the specified file in order to continue to extract details, but a race condition could be possible. **Dr. Memory** displays the output and only after that it starts to write informations to the file. To avoid opening the file and not getting all the wanted output, the parser polls for the number of processes that have the specified file opened. When the number become zero, it means nobody is writing to the file and it can be safely open.

An example of the generated output can be found in listing 4.1.

```
~~Dr.M~~ ERRORS FOUND:
~~Dr.M~~       0 unique,     0 total unaddressable access(es)
~~Dr.M~~       3 unique,     3 total uninitialized access(es)
~~Dr.M~~       0 unique,     0 total invalid heap argument(s)
~~Dr.M~~       0 unique,     0 total warning(s)
~~Dr.M~~       1 unique,     1 total,     10 byte(s) of leak(s)
~~Dr.M~~       0 unique,     0 total,      0 byte(s) of possible
    leak(s)
~~Dr.M~~       0 unique,     0 total,      0 byte(s) of still-
    reachable
allocation(s)
~~Dr.M~~ ERRORS IGNORED:
~~Dr.M~~       0 still-reachable allocation(s)
~~Dr.M~~          (re-run with "-show_reachable" for details)
~~Dr.M~~ Details:
/data/work/projects/robocheck/drmemory-read-only/build/logs/DrMemory
    -simple.11940.000/results.txt
```

Listing 4.1: Dr. Memory Generated Output

After determining that no process has the output file opened, the parser can proceed to opening in order to collect more information. An example of how the output looks like can be found in listing 4.2. For each error that is detected and listed, **Dr. Memory** prints a stack trace, containing details about what triggered that error. It also provides the possibility to extract the function name, the source file and the line number. Each type of error has its own format of printing and can be easily differentiated by a specific string (eg: *UNINITIALIZED READ* or *LEAK*). The parsing produces a temporary output directory that is removed when the process is complete.

```
Error #3: UNINITIALIZED READ: reading register ecx
# 0 libc.so.6<nosyms>!?
# 1 libc.so.6<nosyms>!?
# 2 libc.so.6<nosyms>!?
# 3 simple!main   [/data/work/projects/robocheck/tests/simple.c:13]
# 4 simple!_start
Note: elapsed time = 0:00:00.072 in thread 11940
Note: instruction: movzx  (%edi,%ecx,1) -> %eax

Error #4: LEAK 10 direct bytes 0x08949018-0x08949022 + 0 indirect
    bytes
# 0 simple!main   [/data/work/projects/robocheck/tests/simple.c:8]
# 1 simple!_start
Note: elapsed time = 0:00:00.072 in thread 11940
```

Listing 4.2: Snippets from Dr. Memory Results File

Even if **Dr. Memory** is much faster then **valgrind**, it does not have its maturity. The tool is a new one and doesn't have so many users as **valgrind** to be able to collect a significant number of bug reports. Because of that, it is highly recommended to configure Robocheck to also run **valgrind** on Linux systems.

Chapter 5

Porting Robocheck on Windows

An application is designed to fulfill a certain set of requirements. In the implementation phase, it is very common to change the architecture of the application to fulfill new needs.

Robocheck was not designed to be a platform independent application. It was designed to be compiled and used under the Linux platform. Lacking the intention of running the application under different environments makes the written code difficult to port and rather inflexible.

The software developers write code in a high-level programming language that needs to be translated into machine language, in order to be understand by computers. A compiler is a software that coverts programs from high-level programming language into machine language. Compilers are architecture dependent and can differ upon the operating system used. Because anyone can develop their own compiler, American National Standards Institute (ANSI) published a family of successive standards for the C programming languages. Developers are encouraged to conform to the standards to fulfil portability requests for their programs.

The compiler used under Linux is `GNU Compiler Collection (gcc)` [1]. Since version 3.0, `gcc` provided initial support for C99 features. The latest `gcc` version, released on the 14th of June 2012, is 4.7.1.

As the most popular and most used compiler under Linux, `gcc` allows programmers to take for granted some of the features available, not taking into consideration that other compilers provide little or no C99 support.

The standard compiler on Windows is `cl` [2], part of the Visual Studio development suite. It provides little C99 support.

5.1 Missing Libraries

Because Robocheck takes the configuration from an XML file, some libraries were used to parse the file and extract relevant data. Those libraries can be easily installed under Linux through the default package manager provided by the distribution. To make the code work under the Windows environment, those libraries need to be replaced with corresponding versions specially built for Windows.

Fortunately, libraries in use were pretty common and one can easily find binaries compiled for the Win32[3] environment. Not all version are compatible with the Robocheck code, so it was

[1] http://gcc.gnu.org/wiki
[2] http://msdn.microsoft.com/en-us/library/9s7c9wdw.aspx
[3] http://en.wikipedia.org/wiki/Windows_API

mandatory to search for the compatible ones. The compatible libraries versions was included in the repository (under the `lib-win/` directory) to avoid subsequent time spent for searching.

Robocheck uses dymanic libraries as part of its design. The loader is the part of an operating system that is responsible for loading libraries and programs from executables into memory. The programming interface for dynamic linking and loading is different on the Windows environment than on the Linux environment. In order to provide code compatibility, the solution we thought suitable was to write a wrapper similar to Linux interface that woul be used on Windows. The wrapper used was written by *Ramiro Polla* in 2007 and it can be used under the GNU Lesser GPL [1].

Robocheck analyze the output generated by the integrated tools and extract tokens and different information. String manipulation can be done by using different functions specified in the `string.h` header. The `string.h` header available for `cl` compiler doesn't have support for functions like `strcasecmp`, `strncasecmp`, `strcasestr` and `strdup` (mostly because these function are not standard). Those functions requested to be manually implemented to provide interoperability with the Windows environment.

5.2 Making the Code Compatible with C89 Standard

The first step required to have Robocheck run on Windows is to be able to compile it. As GCC is using C99 features, while cl is not, the code has to be updated.

In the C99 standard the compiler supports initialization of selected fields of structures and unions. Initializations can be done similar to the example in Listing 5.1.

```
typedef struct {
    char b;
    union example_union_value {
        char *p;
        char c;
    } union_value;
    int a;
} example_t;

example_t var = {.a = 2, .union_value.c = 'b'};
```

Listing 5.1: Initialization of Fields for Structures and Unions in C99

C89 doesn't have support for naming the structure field to be initialized. In order to initialize a structure, one must initialize all fields, preserving the order under which they were declared. There is no support for skipping some of the fields. A workaround would be to initialize them with random values.

The standard mentions that union initialization is possible only for the first field from the definition. To initialize another field, one must initialize it with a random value, and after that explicitly modify the value. Even if an union uses the maximum size of the fields it contains, an automatic cast is made at initialization and some data could be lost. The best solution for initialization is to put some junk value and explicit modify it after the initialization. (as shown in Listing 5.2)

```
typedef struct {
    char b;
    union example_union_value {
```

[1] http://code.google.com/p/dlfcn-win32

```
        char *p;
        char c;
    } union_value;
    int a;
} example_t;

example_t var = {'#', NULL, 2};
var.union_value.c = 'b';
```

Listing 5.2: Initialization of Fields for Structures and Unions in C89

As Robocheck made use of initializing structures and unions when they were declared, an important part of the code needed to be rewritten. The initializations currently follow the C89 standard, supported by virtually all modern compilers.

5.3 Loading and Unloading Dynamic Libraries

Using dynamic libraries makes executable files much smaller because the library code isn't included into the application. Dynamic libraries are linked when the application is running. There can be only one instance of the library loaded into memory; the instance is shared among all the application that use it. The advantage is not only the smaller executables; if one runs 100 application that require the same library, the library will be mapped only once into the memory.

The code in a dynamic library is loaded only once (just the first time) and is shared among all processes that use it. The occupied physical memory it is much smaller than if static library would be used. A static library is actually included in the generated object file of the compiled application and makes the executable larger. The data section of a DLL (Microsoft implementation of the shared library) is private for each process that uses the library; as such processes have little idea about each other and can't alter the other one's behaviour.

The loader is responsible for mapping the library address space into the running process space by using relocation techniques. When creating a shared library, the linker performs relocation to assign runtime addresses to each section and symbol generated in the object file (code segment, data segment, etc.). Each element in the relocation table is an address in the object code that must be changed when the loader relocates the program.

5.3.1 Passing Objects Across DLL Boundaries

The concept of dynamic libraries is common to all modern operating systems. No matter what operating systems is used, the concept is the same: a dynamic library is a mechanism for sharing the code of a library among multiple processes without integrating the code into application and through the use of relocation techniques.

Even if the concept is the same, the implementation might slightly differ and the behaviour may not be the same for different platforms. This section will present an issue that appeared while porting Robocheck on Windows, by assuming the shared libraries behaviour is the same.

Scenario

Let's assume there is a shared library named `dll_example` that was generated from the source file defined in Listing 5.3. The library exports the function `print` that uses, as its first parameter, a pointer to an output stream and, as the second parameter, a string to be written.

```c
#include <stdio.h>

__declspec(dllexport) void print(FILE *out, char *string)
{
    fprintf(out, "%s", string);
}
```

Listing 5.3: `dll_example` Dynamic Library

In this scenario, an application makes use of the library in order to print data to a specific stream. To simplify the scenario, let's assume that the application is only printing `Robocheck is awesome` to standard error (`stderr`).

```c
#include <stdio.h>

__declspec(dllimport) print(FILE *out, char *string);

int main(void)
{
    print(stderr, "Robocheck␣is␣awesome!");

    return EXIT_SUCCESS;
}
```

Listing 5.4: `dll_example` Library Usage

Robocheck Behaviour

Robocheck was using a configurable logger system to track important messages for debugging and future analysis reports. The application was using the standard error (`stderr`) for logging purposes, but it did provide the mechanism to update the output stream.

The logging module in Robocheck was working similarly to the one described in the scenario above. The main application could provide, as a parameter, an output stream to the initialization function of the Robocheck library.

Even if this looks pretty simple, the Robocheck framework was crashing at the first call of the logging system. The first approach was to think of uninitialized strings or improperly formatted strings; this wasn't, however, the case.

The application was crashing because the address used as an output stream was invalid. Robocheck was trying to write output to an address that was not attached any output device. The address of `stderr` in the application and the address received in the library were identical.

Using a file instead of `stderr` allowed Robocheck to work properly. So what was the problem?

The rather suprising discovery was that, within a DLL, `stderr` is pointing to a different address than the one received from the application. This behaviour is not part of a Linux Environment.

Explanation

The C standard says that `stdin`, `stdout` and `stderr` are defined as macros that are expanded in the preprocessing stage. The standard doesn't says how these macros should be expanded because every operating systems has its own abstraction for representing file descriptors and opened devices. These macros are defined in the C standard library.

An output stream is in reality an abstraction for the file descriptors (in Linux) and handles (in Windows). Each process has three default files opened: standard input, standard output and standard error. Each process has its own standard devices opened.

In the scenario presented above, the shared library maps the standard C library at a different address. When expanding the macro `stderr`, the address obtained is different from the one obtained in the application. By using the address returned in the application, the framework crashes because inside the library, the address is not valid.

According to this [1], the dynamic library is loading a different C runtime library (because of the dynamic path search) than the application. If the C runtime library loaded by the library, would be the same as the one loaded by the application, the application wouldn't fail because the runtime library is a shared one.

Solution

In order to solve this problem, two possible solutions were identified. One was to make the application load the same runtime library each time; the other was to make things work even if the loader used different libraries.

The first approach didn't scale. In order to make the application load the same runtime library, some changes needed to be done in the environment system of the Windows platform. Those changes needed to be done manually for each Windows platform where Robocheck would be installed. Different platforms mean different configurations and different numbers of possible runtime libraries installed. The workaround would be specific to each platform.

As such, our solution relied on removing the configuration of the output stream. Currently the logger writes messages to standard error. If one needs the output generated by the logger, it can use redirection from the command line interface. This approach doesn't need any changes in the environment system and provides consistent behaviour on all platforms.

5.3.2 Life of a Dynamic Memory Block Allocated Inside a DLL

In Windows, a dynamic memory block allocated inside a DLL will be kept until the memory is freed or the library is unloaded.

In Linux, if one allocates memory in a library call and then returns it to the application, even if the library is unloaded, the memory is still available for the user. This is done because the library is mapped in the process address space and doesn't use a different data segment.

In Windows, if the library is unloaded, accessing the memory at an address allocated by a library call results in memory corruption and usually terminates the program.

Robocheck is running tools as modules, loading corresponding shared libraries at runtime. After running a tool and collecting results, Robocheck unloads the used library. Because the output was collected by allocating space in the heap of the application (in the `run_robocheck` function), all collected data would become invalid at unload time. To solve this issue, Robocheck is keeping now a trace of the loaded libraries and doesn't unload them until the end of the application.

[1] http://msdn.microsoft.com/en-us/library/ms235460%28v=vs.80%29.aspx

Chapter 6

Providing Interoperability in Robocheck

When it comes to automatically generating feedback on the code quality of an application, Robocheck is a very useful tool. However, the generated output could be easily interpreted only by humans, as a standard way of reporting errors was not provided. In order to enhance its interoperability with other tools, the output format has been changed. A standard format has been designed based on `JSON` format.

The unified output was required to be able to integrate Robocheck with other tools, such as `vmchecker` (section 6.2). The output generated by Robocheck needs to be easily parseable by other tools.

6.1 JSON Format for Output of Robocheck

`JSON` is a text-based open standard designed for data exchange between processes or applications. It was derived from `JavaScript` to represent data structures, but it is widely used by many applications because of its simple format. Many languages support parsing `JSON` files.

`JSON` was choose to represent Robocheck output because it is very simple, human-readable and can be parsed easily by many programming languages.

In listing 6.1, one can see an example of the `JSON` output generated by Robocheck.

The output generated by Robocheck is easy to understand. There are also many `JSON` parsers, written for different programming languages, that can be used to extract information from the output.

Errors are grouped by their type. Each error type has:

- a `name` (which briefly describes the error type)
- a `key` (which gives a more detailed description of the error)
- a `value` (which represents the total sum of the penalties for that type of error)
- a field named `where` (a vector of line describing the occurrences of this error type, giving information about where and how it has triggered)

```
{
  "result":
```

```
  [
    {
      "name" : "Error␣type␣no.1.",
      "key" : "Error␣description.",
      "value" : "0.66",
      "where" :
      [
        { "line" : "Details␣about␣where␣and␣how." },
        { "line" : "Details␣about␣where␣and␣how." }
      ]
    },
    {
      "name" : "Error␣type␣no.5.",
      "key" : "Error␣description.",
      "value" : "0.33",
      "where" :
      [
        { "line" : "Details␣about␣where␣and␣how." }
      ]
    }
  ]
}
```

Listing 6.1: Robocheck - JSON output

The output is generated at the end of Robocheck run, because errors must be summed and interpreted in order to remove duplicate errors and correctly format them. Some tools may report the source file where the error occur as a full path, whilst other may report the source file relative path. This kind of inconsistencies should be fixed so that the parser can correctly interpret if two different errors are duplicates or not.

6.2 vmchecker Integration

One of the main purposes of the new version of the Robocheck is to be used in the Operating Systems class starting as soon as the next academic year.

vmchecker is used in many classes at University POLITEHNICA of Bucharest. It is a framework that provides storage and automatically checking of the functionality for the submitted assignments. It is used to simplify the process of grading assignments because the students can verify the correctness of their assignments before the deadline and assistants doesn't have to manually run tests to verify the submissions.

Robocheck needed an infrastructure to automatically check code quality for the assignments. Because **vmchecker** already exists and it is already widely used in the faculty, a need for integrating these tools arises.

The first approach was to change the **vmchecker** architecture in order to support integration of various modules. This could have been done by creating new configuration options and designing support for integrating modules. **vmchecker** would run the basic test (as before) and, after that, would start running the enabled modules one by one.

Even though this approach looks clean and simple at first sight, the actual redesign, along with adding modules into it, would be quite difficult given the existing architecture. **vmchecker** is designed to create a virtual environment and run a set of tests on the application. This mechanism is simple and straightforward as the framework doesn't need to be aware of what it is testing, it only depends on the tests and assignment implementation.

The solution that was implemented approaches the problem in a different manner, to minimize the awareness of **vmchecker** about the existence of other tools. To achieve this, one can simply modify the testing script in order to extend the testing scenario and add quality checking by running Robocheck.

After running the tests, **vmchecker** collects the output files from the testing environment and stores them in the storer machine (section 2.7). The output of the Robocheck framework is stored in a file named *robocheck.vmr*. The Graphical User Interface (GUI) grabs the output of Robocheck (the `robocheck.vmr` file) and parses it and appends to the `grade.vmr` file. In the end, the `grade.vmr` file will contain the feedback on the assignment seemingly given by a Teaching Assistant.

Section 8.3 contains an example of how a `grade.vmr` file looked without the addition of Robocheck and how it looks after.

Chapter 7

The Robocheck Configuration Tool

As mentioned before, Robocheck is an extensible framework and provides mechanisms to configure what tools to be run, with what parameters, what files to test and how the penalty module should work.

The configuration tool starts from a basic template (presented in listing 7.2, with no configurations made) of the XML file. This tool is built simultaneously with the Robocheck framework and it is exported as a command line program with the name `robo_config`. `robo_config` provides incrementally changes for the configuration file.

The configuration file must be named *`rbc_config.xml`*. The tool reads the XML file and interprets its parameters in order to add new settings. Listing 7.1 shows the tool's options.

```
--list-all-tools
--list-startup-tools
--list-tool-info [tool name]
--list-err-info [error ID]
--list-all-errors
--create-tool [Tool Name] [Tool Path] [Tool type]
--register-tool [Tool Name]
--register-error [error ID] [tool name]
--register-parameter [tool running parameter] [tool name]
--add-static-parameter [source name]
--set-executable [executable]
--add-dynamic-parameter [executable parameter]
--add-error-details [error ID] [penalty additional info] [error
    count] [penalty value] [penalty value type]
--set-penalty-info [true/false] [libpenalty path].
```

Listing 7.1: `robo_config` Options

The command `robo_config --list-all-tools` is printing all the tools supported by Robocheck.

To view all the errors that Robocheck can identify, one can simply use the command `robo_config --list-all-errors` to print all the errors and will associate a number for each error (an `error ID`).

For example, to configure Robocheck to also run **Dr. Memory** and be able to identify memory leaks, invalid access exceptions, usage of uninitialized variables and invalid frees, the following commands must be run:

```
robo_config --create-tool drmemory libdrmemory.so dynamic
robo_config --register-tool drmemory
robo_config --register-error 1 drmemory
robo_config --register-error 2 drmemory
robo_config --register-error 3 drmemory
robo_config --register-error 19 drmemory
```

Incrementally adding settings to Robocheck provides flexibility and sturdiness. It is very simple to change small things, but in order to achieve a complete setup, there are many commands that must be run.

In the previous version of the Robocheck framework, errors could be added manually, from the configuration tool. The addition was made by changing the header file where the error types are defined and Robocheck needed to be rebuilt. There is little flexibility in such option, because one can not add an error that does not exist in the implementation of the Robocheck. Adding random errors would only bewilder the framework, because it wouldn't have the capabilities to treat them and would also introduce a big overhead by rebuilding the framework each time a new error was added.

The configuration tool was changed, and the `--create-error` option was removed. Robocheck is now capable of identifying 19 types of errors. By removing this option, the unnecessary 19 builds became history. In the current implementation, all errors are added by default. `robo_config` must start from a simple XML where all possible errors are already added. The header file where the corresponding macros are defined is now static, it does not change in time.

```
<?xml version="1.0" encoding="utf-8"?>
<appSettings>
  <init output="NULL">
    <tools count="0">
    </tools>
    <input/>
    <penalty load="false" lib_path="libpenalty.so"/>
    <err_count value="19"/>
  </init>
  <installed_tools count="0">
  </installed_tools>
  <errors>
    <err_1 id="1" name="Memory␣leak"/>
    [... 18 more errors ... ]
  </errors
>
</appSettings>
```

Listing 7.2: Robocheck - Simple XML Configuration File

Incrementally changing the `XML` file might be a time consuming action. In order to minimize the time spent to create a full configuration, some additional scripts were created. These scripts must be included in the tests archive when running Robocheck as part of the `vmchecker` tool.

The scripts contain a maximum configuration of the Robocheck, with all tools available for that platform configured and with all possible errors to be traced. They also contain an example of how the penalties can be configured. The script for the Linux platform is called `gen_config.sh` and the script for Windows is called `gen_config.bat`. Both of them are located in the root of the Robocheck repository.

Chapter 8

Results

At the moment, all of the objectives for the new release of the Robocheck framework are accomplished. The framework is now capable of running on Windows and can be easily integrated with other tools, `vmchecker` included.

It was a great and rewarding experience to work with this tool and the biggest achievement is that the tool can now be finally used, at least, in the Operating Systems class.

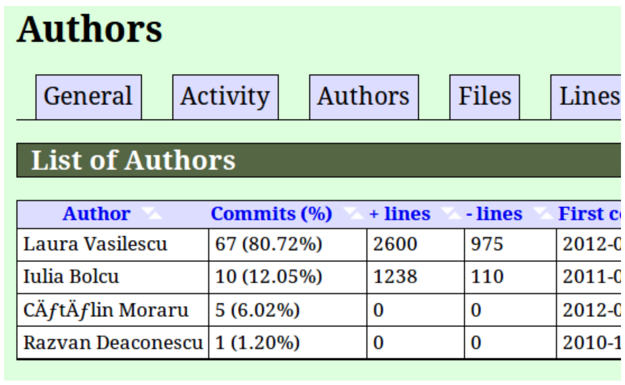
8.1 New Robocheck Release in Numbers

`gitstat` is a useful tool that can be used to generate different statistical informations about git repositories. In order to generate clean reports, a new clone of the repository was required in order to eliminate (using `git filter-branch`) binary files from the commit history.

As one can see in figure 8.1, the new version of the Robocheck framework meant:

- adding 2600 lines of code
- removing 975 lines of code

An important part of the previous version was rewritten in order to provide compatibility with C89 standard.



List of Authors

Author	Commits (%)	+ lines	- lines	First c
Laura Vasilescu	67 (80.72%)	2600	975	2012-0
Iulia Bolcu	10 (12.05%)	1238	110	2011-0
CÄƒtÄƒlin Moraru	5 (6.02%)	0	0	2012-0
Razvan Deaconescu	1 (1.20%)	0	0	2010-1

Figure 8.1: `gitstats` Results on Robocheck Repository

8.2 Assignments Tests

This section reviews statistics about the penalties applied by the Robocheck framework. The tests were run by using `vmchecker` tool. Table 8.1 describes the configured penalties for the test suite.

Table 8.1: Penalties Configuration

Error Name	Penalty Value	For How Many
Memory leak	0.2	1
Invalid access exception	0.3	INF
Uninitialized	0.2	INF
File descriptors error	0.1	INF
Data race	0.1	INF
Dead lock	0.1	INF
Unlock	0.1	INF
Destroy	0.1	INF
Condition variable	0.1	INF
Hold lock	0.1	INF
Duplicate code	0.2	INF
Static variable	0.1	INF
Signed unsigned	0.1	INF
Unverified function return cal	0.3	INF
Function line count exceeds maximum admit	0.2	INF
Function indentation exceeds maximum admit	0.2	INF
Trailing whitespace	0.2	INF
Mixed tabs with spaces	0.2	INF
Invalid free	0.1	INF

8.2.1 Assignment no. 0 - Operating Systems

There were `175` assignments uploaded in `vmchecker`. `4` of them didn't compile, so basically there were only `171` valid assignments submitted.

The smallest penalty was `0`. There were actually `7` assignments with no penalty.

The biggest penalty was `2.7`. There was only one assignment with that much penalty.

The average penalty was `0.71`.

Table 8.2 describes a more precisely overview on the penalties applied by Robocheck.

Table 8.2: Assignment no. 0 - Overview

Error Name	No. of assignments	No. of appearances
Memory leak	120	354
Invalid access exception	24	24
Uninitialized	71	212
Duplicate code	72	143
Static variable	141	1158
Signed unsigned	7	9

8.3 `grade.vmr`. Before and after Robocheck

This section contains examples of the feedback provided by an assistant (without using Robocheck) and the feedback provided by Robocheck framework.

Case 1 - student Alexandru B.

One can see the feedback provided, manually, by an assistant for the assignment in the `grade.vmr` file, in **vmchecker**:

```
-0.4:  memory leaks
+0.0:  good coding style
-- Traian
```

Further down, one can see the feedback provided automatically by the Robocheck framework, when running in combination with **vmchecker**:

```
-1.20:  New memory leak modification (In function ht_find, in file
hash_table.c, at line 80; In function ht_print_bucket, in file
hash_table.c, at line 125; In function get_input_files, in file
tema0.c, at line 64; In function exec_commands, in file tema0.c, at
line 99; In function exec_commands, in file tema0.c, at line 88;
In function parse_command, in file tema0.c, at line 129;)
-0.20:  Use of unitialized variable (In function list_remove, in file
linked_list.c, at line 83; In function open_file, in file util.c, at
line 57;)
-0.10:  Use of non-static variables or functions in a single module
(In file hash_table.h, at line 49; In file tema0.c, at line 22; In
file tema0.c, at line 23; In file tema0.c, at line 24; In file
tema0.c, at line 26; In file tema0.c, at line 27;)
-0.10:  Assignment from signed to unsigned (In function trim, in file
util.c, at line 113;)
```

Case 2 - student Valentin I.

One can see the feedback provided, manually, by an assistant for the assignment in the `grade.vmr` file, in **vmchecker**:

```
+0.0:  nice coding style :)
-- Irina
```

Further down, one can see the feedback provided automatically by the Robocheck framework, when running in combination with **vmchecker**:

```
-0.20:  New memory leak modification (In function execCommand,
in file tema0.c, at line 189;)
-0.20:  Use of unitialized variable (In function add, in file
tema0.c, at line 61;)
-0.10:  Use of non-static variables or functions in a single
module (In file tema0.c, at line 21; In file tema0.c, at line 32; In
file tema0.c, at line 46; In file tema0.c, at line 55; In file
tema0.c at line 79; In file tema0.c, at line 103; In file tema0.c, at
line 113; In file tema0.c, at line 124; In file tema0.c, at line 134;
In file tema0.c, at line 143; In file tema0.c, at line 162;)
```

Chapter 9

Future Development

At the time of this writing, Robocheck works and is able to provide useful functionality, its stability needs to be further improved. There are a lot of features that need to be added for it to become a mature project.

9.1 TUI for Generating XML-Configuration File

One useful tool would be a TUI (Text-based User Interface) for the configuration tool. At the moment, a sample script is present in the repository to help one to understand the capabilities provided by the Robocheck framework. The script can be easily modified to produce configuration changes.

Nevertheless, it would be more convenient if the tool had a TUI to guide the assistants through the entire configuration process. The interface could be developed using the `ncurses` library.

9.2 Multiple Language Support

At the moment, the Robocheck framework can only verify assignments written in `C`. It would be very useful to support other programming languages tools.

Extending the framework with other language support would be useful because this will permit for other classes to integrate automatically checking of the code quality.

`Java` is the next big target for the framework, because there are many assignments written in this language for the Algorithms Design and Object-Oriented Programming classes. The first step would be identifying tools that can be run for checking coding style in Java. `Simian` for example could be also used here, to check for modularity of the written code.

Appendix A

Compiling, Installing and Running Robocheck

The source code of the Robocheck framework is available in `Git` repository at git://ixlabs.cs.pub.ro/robocheck.git. The repository also contains additional source files for integrated tools, but some of them must be manually installed following the provided instructions.

A.1 On the Linux Platform

It is recommended to clone the repository in `/robocheck/repo` directory. One may also clone the repository somewhere else, but the modification of some of the paths listed below must take into consideration.

Some packages are needed to be installed before trying to compile the project. The needed packages are listed in the `install` file script. Run the script.

Copy the `.jar` files from `modules/simian/` to `/lib/`.

Append the following lines to `~/.bashrc`:

```
export PATH=/robocheck/repo:$PATH
export LD_LIBRARY_PATH=/robocheck/repo
```

Run `source ~/.bashrc` to update the environment variables accordingly.

The configuration binary can be accessed by using the command `robo_config`. After generating the complete configuration into a file named `rbc_config.xml`, use the `robocheck` command to run the framework.

Run `robocheck 2> /dev/null` to filter the output and only display the JSON output file.

A.2 On the Windows Platform

It is recommended to clone the repository in `C:\\robocheck\repo` directory. One may also clone the repository somewhere else, but the modification of some of the paths listed below must take into consideration.

Go to `win-install` directory.

Install `DrMemory-Windows-1.4.6-2.exe`.

Extract `splint-3.1.1.win32.zip` archive to `C:\\splint-3.1.1\` .

Install `jxpiinstall.exe`.

Change the `Path` environment variable. From `Desktop`, right-click `My Computer` and click `Properties`. In the `Advanced` section, click `Environment Variables` button. Edit `Path` variable in the `Systems Variable` by appending the following:
`;C:\\robocheck\repo;C:\\robocheck\repo\lib-win`

The configuration binary can be accessed by using the command `robo_config.exe`. After generating the complete configuration into a file named `rbc_config.xml`, use the command `robocheck.exe` to run the framework.

Run `robocheck.exe 2>NUL` to filter the output and only display the JSON output file.

Bibliography

[1] S. Oualline, "*C Elements of Style.*" M & T books, November 1992.

[2] C. Socoteanu, "*Robocheck - Integrated Code Validation Tool.*" Diploma Project, University POLITEHNICA of Bucharest, July 2011.

[3] I. Bolcu, "*Robocheck - Integrated Code Validation Tool.*" Diploma Project, University POLITEHNICA of Bucharest, July 2011.

[4] J. Seward, M. Nethercote, and J. Weidendorfer, "*Valgrind - Advanced Debugging and Profiling for GNU/Linux applications.*" Network Theory Ltd., March 2008.

[5] D. Bruening and Q. Zhao, "*Practical Memory Checking with Dr. Memory.*" International Syposium on Code Generation and Optimization, April 2011.

[6] L. A. Grijincu, A. Moșoi, C. Gheorghe, and I. M. Stănescu, "*vmchecker.*" Scientific Student Projects Session, University POLITEHNICA of Bucharest, May 2009.

[7] J. R. Levine, "*Linkers and Loaders.*" Morgan Kaufmann Publisher, October 1999.

[8] M. Klein, "*Windows Programmer's Guide to DLLs and Memory Management.*" Sams Publishing, September 1992.