Automated assessment of C++ programming exercises with unit tests

Tom-Michael Hesse, Axel Wagner and Barbara Paech
University of Heidelberg, Im Neuenheimer Feld 326, 69120 Heidelberg, Germany
{hesse, paech}@informatik.uni-heidelberg.de, axel.wagner@stud.uni-heidelberg.de

Keywords: Automated testing, C++ testing, Automated source code assessment, E-learning

Abstract

C++ is subject to many programming lectures. To ensure the functionality of handed-in solutions for coding exercises, a dynamic test has to be performed. However, almost no systems can be found, which allow students to have C++ coding exercises automatically built, tested and evaluated through an e-learning system. A major reason is the difficulty to provide an autonomous and secure build process. In addition, it is not obvious in which way the programs shall be tested and scored. In order to tackle those challenges, we developed a test system for C++ source code that employs CppUnit. Test suites for component tests can be created for an exercise in order to automatically assess its functionality. The test cases can be enriched with scoring details and feedback content, so that the students do receive the test results and additional explanations. Moreover, our test system uses different mechanisms like sandboxing and isolation to prevent the system or solution results from being harmed or hijacked.

Introduction

Currently, many systems are available to test and evaluate programming exercises for Java. For example, Java source code can be executed and tested automatically with given test cases. However, systems enabling the automatic testing of C++ source code in the same way as for Java code are mostly not available for public. When asking via mailing lists and personal contacts, we found that some universities have written a bunch of scripts to perform the compile process and log evaluation for C++ in order to ease the correction process of exercises. But to the best of our knowledge, no automatic test system integrated with an online e-learning system could be found that enables students to hand in C++ source code and get a feedback and score for it based on defined test cases. Existing approaches either focus on training basic object-oriented concepts (i.e., ClaRA [[1]]) or provide IDE-like programming support to students on their local PCs (i.e., CPPTutor [[2]]).

In comparison to Java, testing C++ source code automatically introduces two major challenges to the test system. The first challenge is to secure the build process of the code as there is no virtual machine to run the program in a controlled environment. The second challenge is to define suitable test cases that can be executed and assessed automatically. Moreover, there is the general question how to integrate such a test system in an e-learning environment.

Design of the Test System

In order to tackle these challenges, we decided to design a test environment for C++ programs as a standalone test server that can be connected to other e-learning systems. This test server, also called

CppTestServer, gets requests by the e-learning system and is able to build C++ programs from handed in source code. Then, the server tests the programs with test cases. We decided to employ CppUnit to specify these tests, as it is a rich and platform-independent unit test framework for C++. After the tests are executed, the results of the build process and of each test case are returned to the requesting e-learning platform.

An important requirement for the employed e-learning system is that is able to provides code skeletons to the students, which contain the method interfaces where the students can put there custom solution code in. So it is possible to test the specified methods for each exercise with given test cases for these defined methods.

To keep the communication with the server light-weight and robust, we decided to have the input and output of the server in JSON format. The input contains the source and test code files in a Base64-encoded string in order to prevent any interpretation problems due to codepage mismatches. The output is structured according to the Test Anything Protocol (TAP) and can therefore by processed by many different test harnesses and other systems without C++-specific dependencies.

We demonstrate the use of our server by integrating it with the e-learning system *JACK*[1]. It is a web-based e-learning system that enables students to have their solutions assessed automatically for exercises in Java and Mathematics. As JACK is easily extendible and currently does not provide support for C++ programming exercises, we decided to develop a module integrating our C++ test server to JACK. We call this module *CppDynamicChecker*.

We defined a socket-based connection between JACK and our server, specified the transformations needed for data exchange and established a scoring mechanism for the test results retrieved from the test server. For each exercise, a total score of 100 points can be reached. Mapping points to test cases can also be individually defined for each exercise using relative weights. This makes it easy to add new test cases to exercises, as no absolute values of points have to be adapted for the test cases already given. Moreover, we integrated a parser for TAP that replaces the compiler messages from test execution by more detailed explanations for test failures to increase the users' comprehension of the failure. The detailed process of interaction between JACK, the integration module and the test server is depicted in Figure 1.

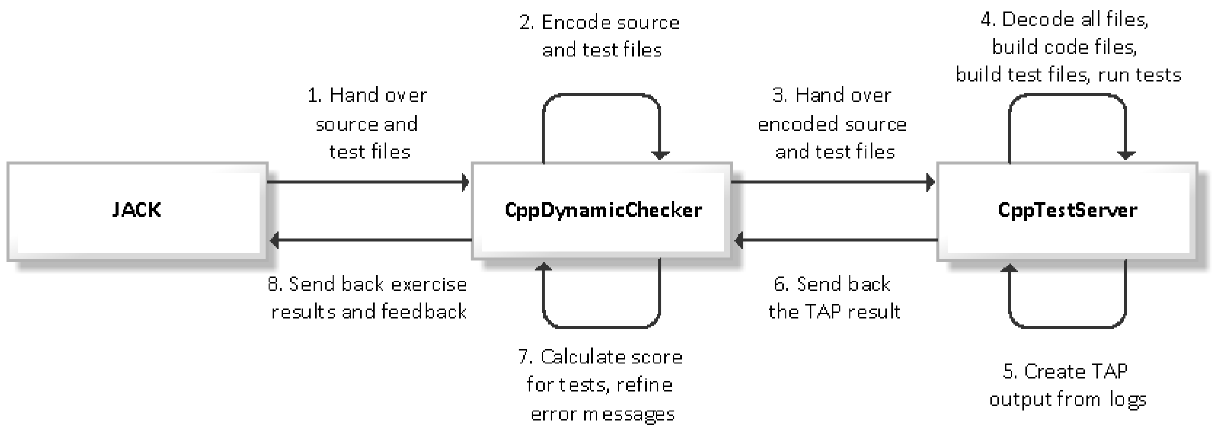
JACK
CppDynamicChecker
CppTestServer
1. Hand over source and test files
2. Encode source and test files
3. Hand over encoded source and test files
4. Decode all files, build code files, build test files, run tests
5. Create TAP output from logs
6. Send back the TAP result
7. Calculate score for tests, refine error messages
8. Send back exercise results and feedback

Figure 1: Interaction between JACK, CppDynamicChecker and CppTestServer

[1] http://www.s3.uni-duisburg-essen.de/research/jack.html

Special Implementation Issues

In this section, we want to sketch those implementation issues in more detail, which are important for security or functionality of the system.

The first issue we want to highlight is the clean execution of code within the test server. To ensure this, the server stores all files for one exercise in a separate temporary directory and builds it independently from other solutions via make. Sets of solution files and their corresponding tests are bundled in a test suite. Those test suites are executed sequentially for each solution. Each solution has its own build and execution process.

A second issue is a solution trying to manipulate the build and test process. To prevent this, we included multiple countermeasures. The solution code is built and executed in a process using the sandboxing-mechanism "EasySandbox"[2], which is an implementation of SECCOMP as a shared library. To ensure that the build and execution process runs in SECCOMP mode, EasySandbox is loaded via LD_PRELOAD. Then, the library makes the SECCOMP call. A major advantage of using this sandbox mechanism is the protection from unwanted system calls. The only allowed system calls are read/write, exit and sigreturn. All other system calls are prevented from being executed by sending sigkill before their execution. In consequence, the test server should be effectively protected from being harmed by the solutions' code. A potential downside is that EasySandbox overwrites the implementation of malloc using a heap of fixed size, before entering the SECCOMP mode. This may lead to insufficient heap space, if the memory needs of solutions are not well estimated in before.

Another countermeasure against manipulation is the systems' option to limit the maximum execution time for the build and run processes of solutions and tests. This limit can be configured and will, for instance, stop solutions that run long time due to an infinite loop. Also an unexpected termination within a students' solution (for instance via exit(0); within tested functions) will neither harm the system nor the test results. When a termination appears while a test case is executed, test results are set to null. This result value will cause no points for this test and therefore is likely to be avoided by students.

Conclusion

The contribution of this paper is threefold. First, we provide a solution for an automated test system for C++ source code. As the server is separated from the productive e-learning system, it can be run in a virtual machine itself in order to prevent security issues. Second, we show that CppUnit is a suitable test framework to enable the automated assessment of unit tests in C++. Third, we have performed the integration of our test system with a typical e-learning system.

The code for the test server is already available at GitHub[3]. As soon as a stable version of code is reached, we will also make our connection module to Jack available for public. Moreover, we will use and evaluate our approach in practice in the upcoming winter semester 2013/14 by supporting a programming lecture.

[2] http://github.com/daveho/EasySandbox
[3] https://github.com/Merovius/bor

Literature

[1] A. Hermanns, V. Jaenen, A. Heide and K. Henning: "ClaRA (C++ learning at RWTH Aachen) Change from classical teaching to e-learing", in *Proceedings of the 7th International Conference on Information Technology Based Higher Education and Training (ITHET '06)*, 2006, pp. 185 – 190

[2] S. Naser: "Evaluating the Effectiveness of the CPP-Tutor, an Intelligent Tutoring System for Students Learning to Program in C++", in Journal of Applied Sciences Research, 2009, vol. 5, no. 1, pp. 109 - 114