A new Testing Framework for C-Programming Exercises and Online-Assessments

Dieter Pawelczak, Andrea Baumann, and David Schmudde
Faculty of Electrical Engineering and Computer Science,
Universitaet der Bundeswehr Muenchen (UniBw M),
Neubiberg, Germany

Abstract - *Difficulties with learning a programming language are wide spread in engineering education. The use of a single integrated programming environment for coding, debugging, automated testing and online assessment lowers the initial burdens for novice programmers. We have developed the Virtual-C IDE especially for learning and teaching the C programming language with an integrated framework for program visualizations, programming exercises and online assessments. A new enhancement of the IDE is a xUnit like testing framework allowing on the one hand larger sets of small, test-based programming exercises and on the other hand simplifying the development of programming assignments. The integration of the new testing framework in the assessment system gives students a better and direct feedback on their programming achievements and helps to find syntactic and semantic errors in their source code.*

Keywords: C-programming, teaching programming, unit testing, static code analysis, dynamic code analysis

1 Introduction

Difficulties with learning a programming language are a well-known challenge for students and lecturers in undergraduate courses [1]. As several studies show, continuously practicing programming by starting from small problems shows respectable success, e.g. see [2]. For small classes, training might be part of the lectures and additional tutors can give a hand to prevent students from falling behind. Although the same could be done for large classes, training during the lecture becomes less effective due to the high diversity of previous knowledge of the students, and finding sufficient and appropriate tutors is an extensive task. Luckily an advantage of learning programming is that – after managing an initial burden – students can directly grasp, what happens by debugging or testing their programs. A direct integration of testing in an adequate IDE further lowers the burdens for novice programmers. Ideally students work continuously on small programming exercises and receive directly feedback from the IDE with respect to syntactical and semantic issues; and finally students can submit larger programming assignments from that IDE to receive their credit points.

We have developed the *Virtual-C IDE*[1] (especially designed for learning and teaching the C programming language) over the last years [3]. We use the IDE for automatic assessment and grading of programming assignments in the third year now. However the original aim to have many small accompanying programming exercises for self-learning could not be established yet, due to the high effort for writing tests. In this paper we present a new testing framework, which enormously reduces the effort for test development. Although this framework allows students to write their own tests, we do not plan to integrate test writing in the primer C programming course at the moment, as our curriculum covers software testing in the software engineering courses in the major terms.

2 Review of related work

Software testing is a core topic in computer science. Even though software testing is often taught in conjunction with software engineering, it becomes more and more important for programming courses: besides testing first approaches, tool-based testing is widely used today in programming primers [4]. The benefit of testing for students is obvious: with test programs provided by the lecturer, students can train programming outside classroom and have immediate feedback on their exercises. For the Java programming language, jUnit tests are widely spread. The language independent testing concept is typically named *x*Unit tests [5]. Based on these, systems for automated grading of programming assignments like e.g. *AutoGrader* have evolved [6]. As *x*Unit testing is aimed more at professional developers, several tools to simplify test specifications or handling of tests have been introduced for teaching programming, as for instance *Web-CAT* [7]. While *Web-CAT* additionally focuses on tests written by students, other platforms analyze beyond unit testing the data and control flow of programs, like e.g. *ProgTest* [8]. *AutoGradeMe* works independent from unit testing and is based on static code analysis and flow analysis of Java programs [9].

An important aspect is the actual purpose of testing: while assessment tools typically test, evaluate and grade a program source *after* its submission, *x*Unit tests provide immediate feedback without formal grading. The new testing framework presented in this paper covers both: an *x*Unit like test system for offline training and evaluation of programming exercises as well as an automated assessment system for programming assignments. As the test framework is directly integrated in the IDE, students benefit from a single environment for coding, debugging and testing.

[1] https://sites.google.com/site/virtualcide/

3 Testing framework

The testing framework (TF) is generally based on the well-known *x*Unit frameworks and its test dialogs [5]. It is syntactically adapted from the *Google C++ Testing Framework* [10] with regards to the C programming language and for educational scope. A test suite (TS) is a single test file and consists of test cases (TC) based on one or more tests. Main differences (despite the programming language) compared to the *Google C++ Testing Framework* are:

- Random test data via the macro `ARGR`.
- Simplified verification of output parameters with the `ARG` macro.
- Predefined tests: function and reference function tests (Section 3.4.2), performance tests (Section 3.4.3) and I/O tests (Section 3.4.4)
- No test fixture macro `TEST_F`. Instead, test fixtures are provides in the test prologue.
- Test and test case names can be string literals
- Heap and data segment re-initialization per test case for full application tests, i.e. execution of `main()`.
- Dynamic re-linking of C functions (Section 3.4.5)
- Automatic prototyping for functions under test.
- GUI based selection/ de-selection of test cases.

3.1 Constraints for the educational scope

It is a challenge for automated testing of students' source codes on the one hand to support students in finding and fixing of programming mistakes in their code, and on the other hand not to reveal too much information about the solution. The use of a reference implementation inhibits disclosure of the test definition. Still a reference implementation is a common and efficient way to test students' code, compare e.g. [6] [8]. Revealing the test definition (as it is self-evident in software engineering) can help students in understanding the task description and train their testing skills, see e.g. [4]. The TF in general allows to open and edit test definitions except for automatic assessment of programming assignments, i.e. for graded submissions. Theoretically a student can adapt her/ his code according to the test results, thus creating a solution, which fits to the tests but not to the task description. Likewise undesirable is, that students add workarounds in their existing solution to fit the tests. This raises the question about the level of detail for the test results. Our former approach was to give a programming description with detailed specification on the functions and the program's I/O. The tests performed for the assessment ran locally in the *Virtual-C IDE* with direct feedback. However the test input was not revealed in the test results. Although most errors could be fixed easily comparing the test report with the exercise description, student's missed the lack of test data. The new testing framework offers the opportunity to reveal test data in the test results as we introduced random data. This highly increases the number of tests and hardens students to program according to the test results instead of the specification. To check on specific failures, the student can deselect tests that already passed to focus on his/ her errors.

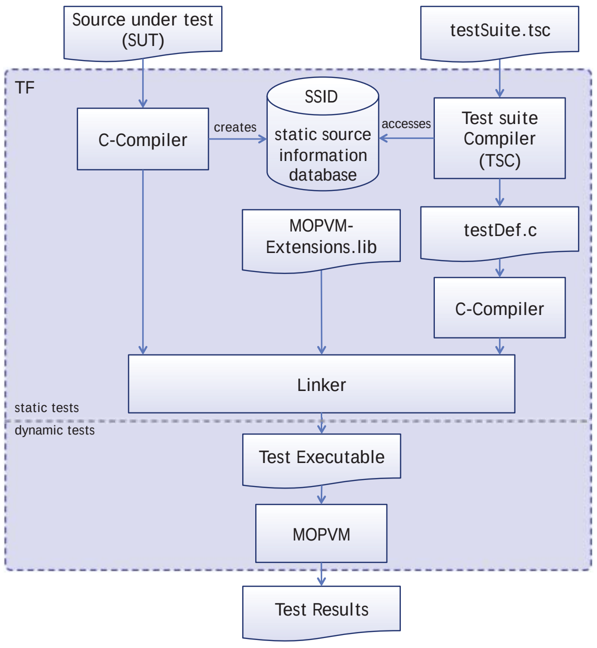


Figure 1. Structure of the testing framework (TF)

3.2 Structure of the testing framework

Figure 1 shows the structure of the TF. The first step covers compiling the student's source file under test (SUT). In case of a compiler error, the test execution is aborted, as a syntactical correct file is expected. Static source code information is stored on a function base in a database (SSID). Static information covers for instance the parameters, number of integer or float-operations, the maximum loop depth, recursion, etc. Afterwards the test suite is compiled by the test suite compiler (TSC), which generates a C file of the corresponding test cases. This file is compiled and linked with the SUT and together with the virtual machine (MOPVM) extensions library. Finally the test is executed in the MOPVM and the results are displayed in the test dialog, see Section 4.1. The TSC is a stand-alone tool, which is able to generate test files according to a test suite description; still it relies on the SSID and the MOPVM features integrated in the *Virtual-C IDE*.

3.3 Static tests

The compiler issues typical warnings during its semantic analysis phase with respect to typecasts, unused or un-initialized local variables, dead-code, accessing NULL-pointers, etc. Another important static test is performed by the TSC, as it checks, if functions under test (FUT) are properly used in the SUT, e.g. if count and types of arguments are correct. This measure prevents linker errors, which are typically difficult to trace for students. The results of the static tests are printed to the console and visualized in the test dialog, see Section 4.1.

3.4 Dynamic tests

Dynamic tests are the main focus of the framework. The test suite can run multiple test cases on a function basis as well as on a program basis. In both cases, a test fixture is set up. Instead of mock objects as specified by the Google C++ Testing Framework [10], the TF provides per test case a *test prologue* and *test epilogue*, as you can see in the following examples Figure 2-6. The test fixture is defined in the prologue; in between prologue and epilogue each test will use a clean test fixture. Optionally TCs can share local variables between tests. For each test case, even the data and heap segments are restored. Consecutive tests inside a test case share data and heap segments, which is for instance important when testing a set of functions with respect to linked lists. The epilogue allows modifying the overall test case result, adding text to the test report or performing clean-ups as, e.g. freeing system resources. The test is run and results can be evaluated afterwards (black-box testing) or during the execution (white-box testing). Black-box testing is preliminary done by evaluating return values, output parameters or console output on given parameters or console input. White-box testing can be achieved by function injection: the linker uses dynamic linking in test mode; thus every function can be re-linked during run-time to a test, a mock or a test-and-mock function, see Section 3.4.5.

3.4.1 Assertions vs. expectations and warnings

In accordance with the Google C++ Testing Framework [10] the TF distinguishes between assertions and expectations as expressed by the macros `ASSERT_*` and `EXPECT_*`. An assertion must be met and contradiction leads to an immediate failure of a test. An expectation might not be fulfilled. This will lead to a failure of the test, but its execution is continued. A typical example for expectation vs. assertion is a function modifying a pointer passed as parameter. It is wrong, if the student does not test for a NULL pointer; still the functional part might be implemented correctly for valid pointers. In case the programming assignment does not rely on the NULL pointer test, this test could use `EXPECT_*`, whereas the proper functionality is tested by assertions. The behavior of assertions and expectations can be expressed with the macros `FATAL()` and `ERROR()` respectively, to print a corresponding message in the report. Additionally, a test can always print warnings with the macro `WARN()`.

3.4.2 Function tests

In addition to the `TEST()` macro as specified in [10], the TF defines the two macros `_funcRefTest()` and `_funcTest()`. Both macros allow a simple but powerful notation for function tests; the first requires a reference implementation for comparing the results. This short test description is possible by implicitly invoking assertions for given parameters and return values and by adding functional extensions to C. For every function test the TSC uses reflection by querying the SSID for function return and parameter types. Figure 2 shows an implementation of a simple TC including four test descriptions. The `_funcRefTest()` macro expects the name of the FUT, a corresponding reference function, a factor specifying the count of allowed instructions compared to the reference function and the arguments for the function call. The `ARGR()` macro generates random test data in a given range for a specified type. Per default, each `ARGR()` adds three tests (additive, not combinatorial); an optional fourth argument can specify the number of tests. Thus the `_funcRefTest()` example in Figure 2 actually creates six tests. For pointers a pointer to the specified value is created. Strings are treated different, as `char` or `wchar_t` pointers are commonly used for character arrays; thus `ARGR()` creates a modifiable null-terminated string array of printable ASCII characters with a length corresponding to the given range (actually the allocated memory will always refer to the maximum range, only the string length varies). In a function tests with `_funcTest()` you provide the number of allowed instructions together with a list of arguments. For functions, the last parameter is an expression of the expected return value, compare Figure 2. This macro can be easily used to test fix values or to forgo a reference implementation.

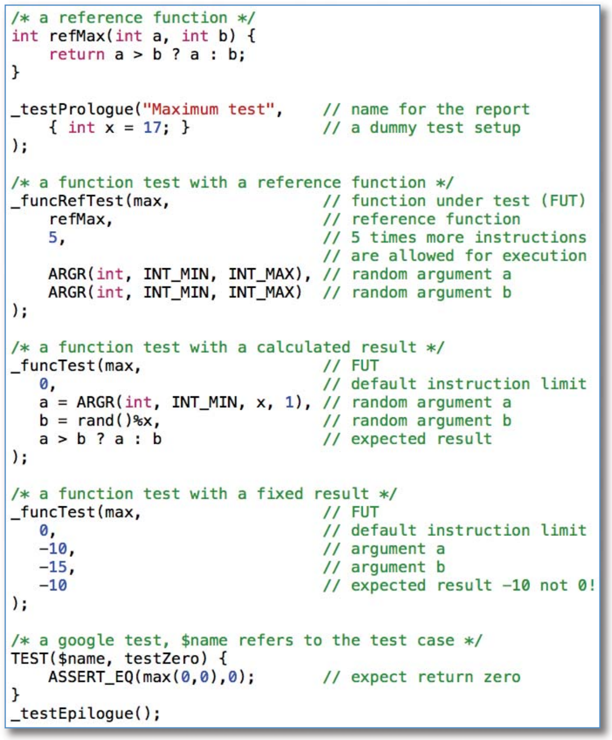

```c
/* a reference function */
int refMax(int a, int b) {
    return a > b ? a : b;
}

_testPrologue("Maximum test",    // name for the report
    { int x = 17; }              // a dummy test setup
);

/* a function test with a reference function */
_funcRefTest(max,                // function under test (FUT)
    refMax,                      // reference function
    5,                           // 5 times more instructions
                                 // are allowed for execution
    ARGR(int, INT_MIN, INT_MAX), // random argument a
    ARGR(int, INT_MIN, INT_MAX)  // random argument b
);

/* a function test with a calculated result */
_funcTest(max,                   // FUT
   0,                            // default instruction limit
   a = ARGR(int, INT_MIN, x, 1), // random argument a
   b = rand()%x,                 // random argument b
   a > b ? a : b                 // expected result
);

/* a function test with a fixed result */
_funcTest(max,                   // FUT
   0,                            // default instruction limit
   -10,                          // argument a
   -15,                          // argument b
   -10                           // expected result -10 not 0!
);

/* a google test, $name refers to the test case */
TEST($name, testZero) {
    ASSERT_EQ(max(0,0),0);       // expect return zero
}
_testEpilogue();
```

Figure 2. Function test definitions

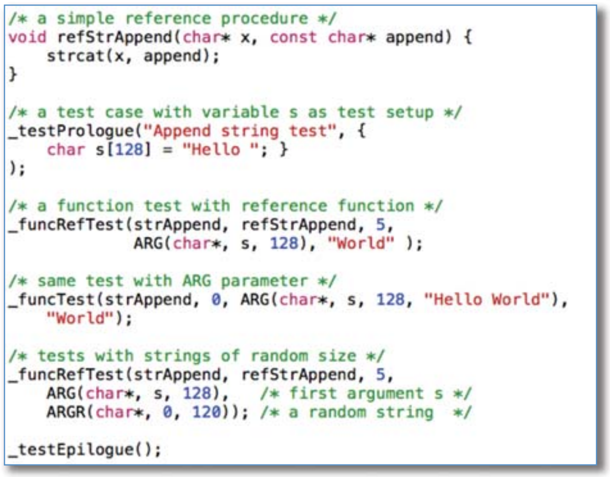
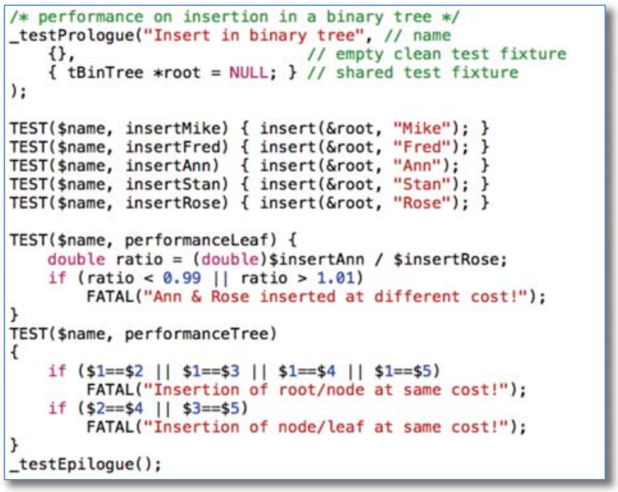

```c
/* a simple reference procedure */
void refStrAppend(char* x, const char* append) {
    strcat(x, append);
}

/* a test case with variable s as test setup */
_testPrologue("Append string test", {
    char s[128] = "Hello "; }
);

/* a function test with reference function */
_funcRefTest(strAppend, refStrAppend, 5,
             ARG(char*, s, 128), "World" );

/* same test with ARG parameter */
_funcTest(strAppend, 0, ARG(char*, s, 128, "Hello World"),
    "World");

/* tests with strings of random size */
_funcRefTest(strAppend, refStrAppend, 5,
    ARG(char*, s, 128),   /* first argument s */
    ARGR(char*, 0, 120)); /* a random string  */

_testEpilogue();
```

Figure 3. Tests for output parameters

Function output parameters can be tested with the `ARG()` macro. In case a non-constant pointer parameter is passed via the macro, the result is compared with the argument of the reference implementation or the optional forth argument of `ARG()`; e.g. `ARG(char*, s, 128, ”Hello World”)` checks, if the contents of `s` is “Hello World” after the function call. The third parameter defines the maximum allocated memory size. Figure 3 shows a test case with three different simple tests on strings. The second test uses the `ARG()` macro to feed an in-/ output parameter and to verify its contents afterwards. The third test uses `ARG()` in combination with a reference function.



```
/* performance on insertion in a binary tree */
_testPrologue("Insert in binary tree", // name
    {},                           // empty clean test fixture
    { tBinTree *root = NULL; } // shared test fixture
);

TEST($name, insertMike) { insert(&root, "Mike"); }
TEST($name, insertFred) { insert(&root, "Fred"); }
TEST($name, insertAnn)  { insert(&root, "Ann");  }
TEST($name, insertStan) { insert(&root, "Stan"); }
TEST($name, insertRose) { insert(&root, "Rose"); }

TEST($name, performanceLeaf) {
    double ratio = (double)$insertAnn / $insertRose;
    if (ratio < 0.99 || ratio > 1.01)
        FATAL("Ann & Rose inserted at different cost!");
}
TEST($name, performanceTree)
{
    if ($1==$2 || $1==$3 || $1==$4 || $1==$5)
        FATAL("Insertion of root/node at same cost!");
    if ($2==$4 || $3==$5)
        FATAL("Insertion of node/leaf at same cost!");
}
_testEpilogue();
```

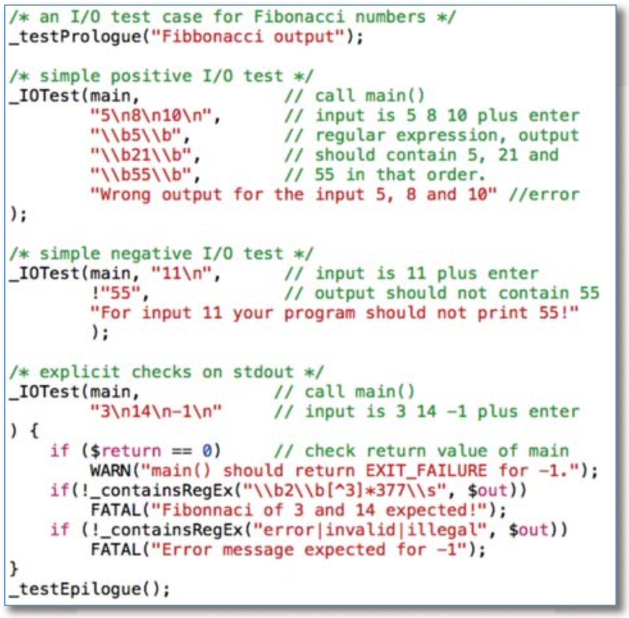
Figure 4. Performance tests (insertion in binary tree)

3.4.3 Performance tests

Performance tests evaluate the number of instructions required for the execution of a FUT; the instruction counter can be queried with the MOPVM extension library function `_getExecutionCount()`. Each tests initially resets the counter, so that the counter can be evaluated in a `TEST()` macro. To access the execution counter from other tests within a TC, the instruction counter is additionally stored in the pseudo variables $1 … $n for n test cases. So each test can compare the performance of the previous tests. The execution count of a test can also be queried by $*testName*, as long as the test name is specified as a regular identifier, compare e.g. `$insertAnn` in Figure 4. These variables can be evaluated either in a `TEST()` macro or in the epilogue. Figure 4 shows a simple and far not complete test case checking on the performance of a binary tree insertion. The test case expects, that insertion of leafs at the same depth require about the same count of instructions. The insertion of the root, nodes or leafs in different depth cannot be performed with the same count of instructions, as an insertion in an array for instance would allow.

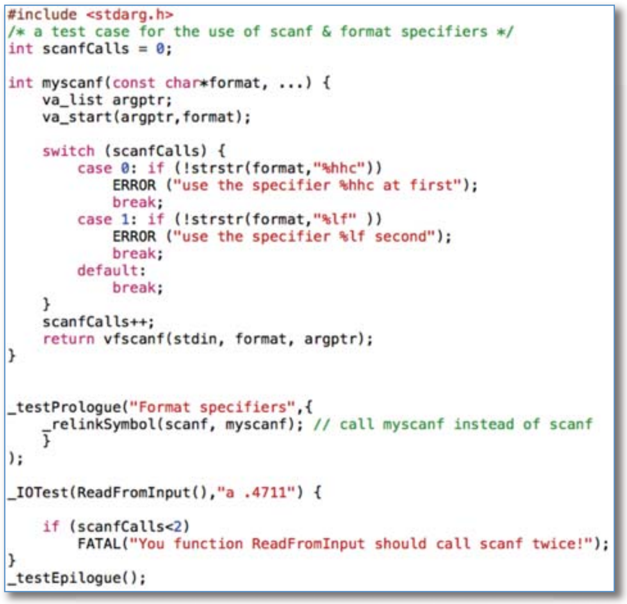
3.4.4 I/O tests

A console C program typically reads data from *stdin* and prints results to *stdout*. I/O tests can be performed on functions or whole programs. The MOPVM extensions library allows simple redirection of *stdin* and *stdout*. The `_IOTest` macro requires a string literal as input for *stdin*. Instead of the NUL-character, EOF is passed to the application. The optional third and further arguments present *stdout*. This is a list of string literals representing a regular expression on the expected or (with the !-Operator) unexpected output plus a descriptive error message. Alternatively, the test can have a body for an explicit test definition: the pseudo variable `$return` refers to the return value of the FUT whereas `$out` can be used to check on *stdout*, compare Figure 5.



```
/* an I/O test case for Fibonacci numbers */
_testPrologue("Fibbonacci output");

/* simple positive I/O test */
_IOTest(main,                  // call main()
        "5\n8\n10\n",          // input is 5 8 10 plus enter
        "\\b5\\b",             // regular expression, output
        "\\b21\\b",            // should contain 5, 21 and
        "\\b55\\b",            // 55 in that order.
        "Wrong output for the input 5, 8 and 10" //error
);

/* simple negative I/O test */
_IOTest(main, "11\n",          // input is 11 plus enter
        !"55",                 // output should not contain 55
        "For input 11 your program should not print 55!"
        );

/* explicit checks on stdout */
_IOTest(main,                 // call main()
        "3\n14\n-1\n"         // input is 3 14 -1 plus enter
) {
    if ($return == 0)         // check return value of main
        WARN("main() should return EXIT_FAILURE for -1.");
    if(!_containsRegEx("\\b2\\b[^3]*377\\s", $out))
        FATAL("Fibonnaci of 3 and 14 expected!");
    if (!_containsRegEx("error|invalid|illegal", $out))
        FATAL("Error message expected for -1");
}
_testEpilogue();
```

Figure 5. I/O tests



```
#include <stdarg.h>
/* a test case for the use of scanf & format specifiers */
int scanfCalls = 0;

int myscanf(const char*format, ...) {
    va_list argptr;
    va_start(argptr,format);

    switch (scanfCalls) {
        case 0: if (!strstr(format,"%hhc"))
            ERROR ("use the specifier %hhc at first");
            break;
        case 1: if (!strstr(format,"%lf" ))
            ERROR ("use the specifier %lf second");
            break;
        default:
            break;
    }
    scanfCalls++;
    return vfscanf(stdin, format, argptr);
}


_testPrologue("Format specifiers",{
    _relinkSymbol(scanf, myscanf); // call myscanf instead of scanf
    }
);

_IOTest(ReadFromInput(),"a .4711") {

    if (scanfCalls<2)
        FATAL("You function ReadFromInput should call scanf twice!");
}
_testEpilogue();
```

Figure 6. Function injection

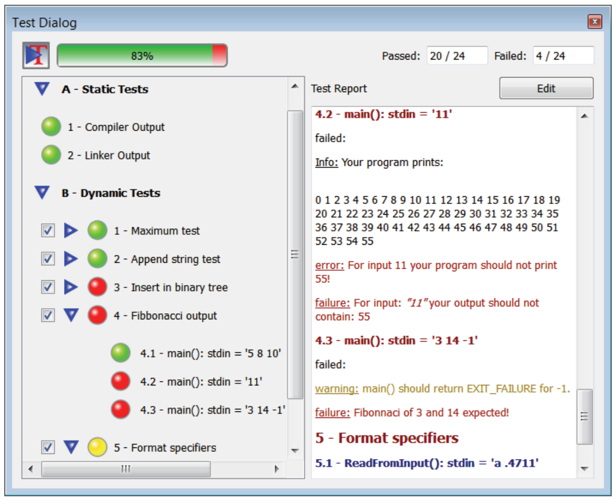
3.4.5 Function injection

The *Virtual-C IDE* uses dynamic linking for testing, i.e. the VM maintains a look-up-table for each function. A test case can modify the look-up-table with the `_relinkSymbol()` function by overwriting the original function pointer with a function pointer to a mock or a test function.

This allows replacing any function as long as the new function provides the same signature. Figure 6 shows a test case on the `scanf()` function by replacing `scanf()` with `myscanf()`. This function counts the number of calls as well as it checks the format specifiers. The function injecttion is done here in the test fixture, thus it is active throughout the test case. Function injection can also be done on a test basis, i.e. each test can provide it's own mock function. The original function linking is restored when running the next test case, thus the following test case will operate again on the original `scanf()`-function unless it is re-linked again.

4 Field of application

The testing framework has two major modes of operation. By opening test suites (file extension .tsc) the test dialog is opened and tests can be directly run in the IDE. This mode is called the exercise mode, as it is designed for self-learning. The same dialog can be indirectly opened as part of the automated assessment system (assessment mode).



Test Dialog

83% Passed: 20 / 24 Failed: 4 / 24

A - Static Tests
- ● 1 - Compiler Output
- ● 2 - Linker Output

B - Dynamic Tests
- ✓ ● 1 - Maximum test
- ✓ ● 2 - Append string test
- ✓ ● 3 - Insert in binary tree
- ✓ ● 4 - Fibbonacci output
  - ● 4.1 - main(): stdin = '5 8 10'
  - ● 4.2 - main(): stdin = '11'
  - ● 4.3 - main(): stdin = '3 14 -1'
- ✓ ● 5 - Format specifiers

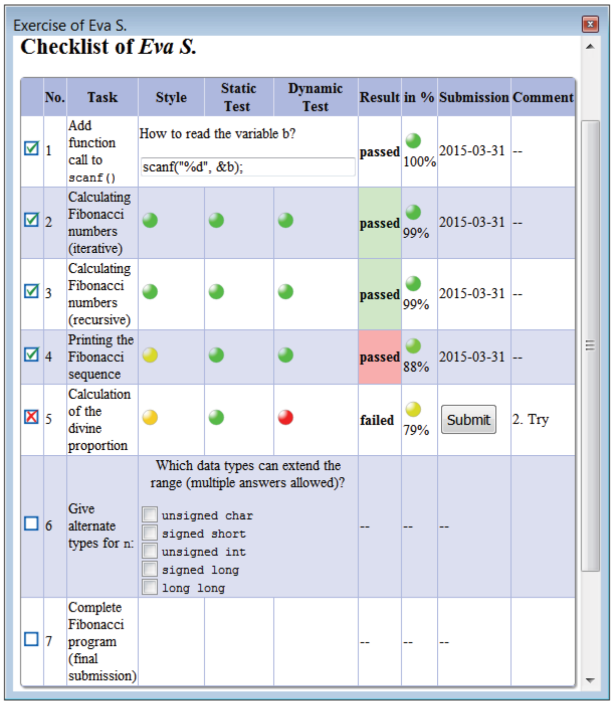
Test Report

```
4.2 - main(): stdin = '11'

failed:

Info: Your program prints:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35
36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51
52 53 54 55

error: For input 11 your program should not print
55!

failure: For input: "11"your output should not
contain: 55

4.3 - main(): stdin = '3 14 -1'

failed:

warning: main() should return EXIT_FAILURE for -1.

failure: Fibonnaci of 3 and 14 expected!

5 - Format specifiers

5.1 - ReadFromInput(): stdin = 'a .4711'
```

Figure 7. Test dialog of the exercise mode (EM)

4.1 Exercise mode

A student can open a test file and run it on her/ his C-module to receive a report on her/ his achievements. Figure 7 shows an example dialog of the exercise mode (EM). The results of the static and dynamic tests are directly visualized in a traffic-light scheme: red (fatal failures), yellow (errors), green (pass). The test report is printed to the console. In addition to *x*Unit tests, the user can directly deselect test cases or select specific test cases in the dialog to focus on single failures. The test code is initially hidden, to focus on the testing; the "edit" button allows viewing and editing the test specification, as described in Section 3.4. EM is for self-learning or lecture accompanying; the lecturer can provide an exercise description together with a test file to allow students testing their solutions. As the IDE supports direct opening from URLs, a test file can also be published on a web server.



Exercise of Eva S.

Checklist of *Eva S.*

	No.	Task	Style	Static Test	Dynamic Test	Result	in %	Submission	Comment
✓	1	Add function call to `scanf()`	How to read the variable b? scanf("%d", &b);			passed	● 100%	2015-03-31	--
✓	2	Calculating Fibonacci numbers (iterative)	●	●	●	passed	● 99%	2015-03-31	--
✓	3	Calculating Fibonacci numbers (recursive)	●	●	●	passed	● 99%	2015-03-31	--
✓	4	Printing the Fibonacci sequence	●	●	●	passed	● 88%	2015-03-31	--
✗	5	Calculation of the divine proportion	●	●	●	failed	● 79%	Submit	2. Try
☐	6	Give alternate types for n:	Which data types can extend the range (multiple answers allowed)? ☐ `unsigned char` ☐ `signed short` ☐ `unsigned int` ☐ `signed long` ☐ `long long`			--	--	--	
☐	7	Complete Fibonacci program (final submission)				--	--	--	

Figure 8. Submission dialog of the assessment mode (AM)

4.2 Assessment Mode

In assessment mode (AM) a student works on a larger programming assignment. She/ he has to submit her/ his solution from the *Virtual-C IDE* to a server. The student's code is checked and the progress is stored on the server. Unless the code has passed the tests, the student can submit corrections of his/ her code. After a successful submission, the student can continue locally in EM to enhance her/ his solution. Programming assignments can consist of multiple tests. Figure 8 shows an example of the submission dialog, which is a plug-in of the Virtual-C IDE. The dialog is implemented as a web view and controls the workflow of a programming assignment. The submission dialog is actually presented as a questionnaire with embedded test suites – for details see [3]. It is an html document presenting the tasks and gathering the results. Each submission is stored on the server. For a code submission, the test suite will be downloaded from the server and executed as if the test is executed in EM. Afterwards the results are updated in the submission dialog:

- *Style*: coding style warnings – additional style checks
- *Static Test*: test results from static tests (EM)
- *Dynamic Test*.: test results from dynamic tests (EM)
- *Result*: an overall percentage based on the criteria above

A threshold is defined for each criterion, so that too many compiler, style or linker warnings might already abort further tests. In case a test fails, the EM dialog is opened to present the detailed test results. In opposite to EM, the student cannot continuously run tests, as the number of test runs is limited to prevent try-and-error submissions. The student is not allowed to edit or view the test suite, as it may contain a reference implementation. In addition to EM, style checks and plagiarism detection are performed.

4.2.1 Coding Style

Today's software developer tools widely support auto formatting of source code. Nevertheless, the authors think, that following code styling rules – especially with respect to structuring source code by proper indentation – is still a competence students should achieve in programming courses. The IDE therefore does not provide auto format but checks the source code against a set of rules like for instance:

- Indentation; a consistent indentation is expected throughout the code: either K&R style or ANSI style. Proper indention in conditions and loop bodies.
- Identifier names are using the proper upper/ lower case conventions for variables, functions, defines, etc.
- No use of magic numbers.

The coding style test is a build-in function of the *Virtual-C IDE*. In AM the style test is mandatory, i.e. a source code submission without any indentation (as for instance received via email) won't pass the style test.

4.2.2 Plagiarism Detection

It is beyond doubt, that plagiarism detection is required in any automated assessment system with respect to source code. As plagiarism is an academic offence, it is handled completely different compared to programming faults; as an option, a plagiarizing student can be blocked from any further submissions to trigger a talk with the course instructor. After the talk, the student can continue working on his/ her assignment if applicable, as the course instructor can enable re-submission. Results from the plagiarism detection are presented with a traffic-light background color in the result report, but are not included in the overall percentage. In Figure 8 the first two program submissions have passed the plagiarism detection and are marked with a green background, whereas the last submission failed. For details on the plagiarism detection system see [11].

4.3 Offline Mode

A third mode is the offline mode (OM), which is nearly identical to AM. It allows performing a programming assignment offline, i.e. either from a local repository or from a webserver but with unidirectional access. OM can serve as a preparation for the examination or to provide additional more extensive exercises. OM is also important for the lecturers or course instructors to prepare the test suites of a programming assignment.

5 Evaluation

The automatic assessment system is used in our C programming course for three years now. Students prepare their programming assignments at home and are allowed to submit their code during two hours class time. Each programming assignment typically consists of five consecutive code submissions. During class time 2-3 instructors are present to support 4 groups of about 20 students each. Initially the system was installed to reduce the administrative work of the instructors, to reduce plagiarizing and to focus more on programming issues.

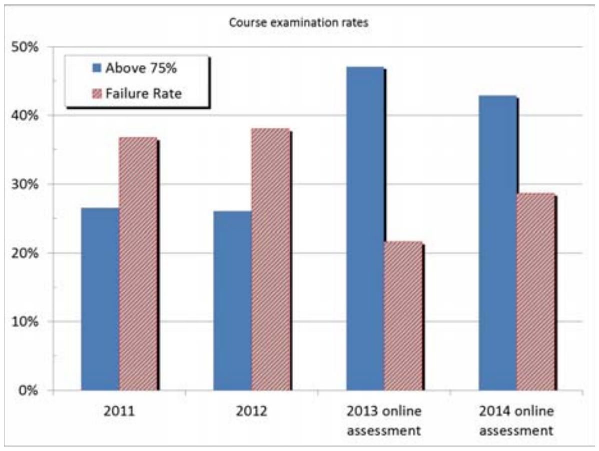


Figure 9. Course examination results compared with online assessment from 2013 and without before

5.1 Code submissions

The formal functional tests of the automated assessment system require, that students put more time into their programming assignments with respect to fixing errors compared to the years before. In addition, each submission is treated equally; instructors cannot turn a blind eye to minor programming mistakes or to collaborative work. This had a positive effect on the examination results: students were more comfortable with typical programming constructs, compare Figure 9.

5.2 Test reports

Properly explaining a failure to a student is the most difficult part of the automated assessment system. The advantage of the *x*Unit based testing is, that assertions are printed in the report in a "standardized" format, like, e.g.: *expected 55, but result is 34*. The TSC will additionally add the corresponding function call for function tests in the report, like e.g. *fibonacci(10)*. Students have the most difficulties with exceptions. Although the test report prints the line in the source code, that is responsible for the exception, students hardly find the problem on their own. Reasons for exceptions were tests, passing NULL pointers, as well as erroneous code like uninitialized pointers, unallocated or too less allocated memory and array bounds. Our expectation was that students debug their failing functions by simply copying the function call from the test report to their source code. But with respect to function testing, students seem to be overextended. They often seem to flinch from changing their code for debugging. For I/O tests on the opposite, students usually run their programs with the test input without difficulties.

5.3 But-it-works syndrome

As other studies show, students perform rarely tests on their code with a high coverage [12] [13]. So a failure in their submission is often taken as an error of the test system or harassment. Unfortunately not all students read the assignment description properly. They might for instance print the Fibonacci number to *stdout* inside the function `fibonacci()` instead of returning the calculated number as requested. The program gives the expected output to screen, but the function test self, of course, fails. Another typical fault is storing the return value in a global variable instead of using the return statement; and again the function test will fail. Although these examples can be easily explained to a good student, as they represent unstructured programming habits, other students often see the modification of a working solution just as additional work. Laborious but effective is to add a reference to the assignment description in the report, e.g. *your function does not return the expected value as described in section … on page …*

5.4 Test development

Writing tests with the new testing framework is exceptional easier compared to writing tests by directly using the function based MOPVM extension library (compare [3]). Especially beneficial is the automated report generation and the simplified random data generation. Thus an instructor must put less time in developing tests; still the test coverage is much higher as the number of actual tests rises due to the simple randomization of arguments.

6 Conclusion and outlook

The new testing framework integrated in the *Virtual-C IDE* enables students to develop, debug and tests their programs in a single programming environment. Small test suites provided by the course lecturer can serve as accompanying exercises with little effort for the lecturer. At the same time, the test framework smoothly integrates into an automated assessment system. We expanded the system towards a better reporting, an appealing visualization and higher test coverage. In opposite to secret tests for programming submissions, details on the test data is laid open to students in order to give a better feedback for fixing errors.

Although the TF supports performance tests there is still a high potential in pushing performance tests further. A lack of the assessment system is, that code fitting the requirements will mostly pass even if it is written cumbersome or less effective. So good students may miss an opportunity to discuss their solutions with the course instructors or fellow students because of a failure or an unexpected poor feedback. An additional report on the quality of the submission could trigger such a discussion for the benefit of these students. An ongoing research at our institute is detailed analyzing the dynamic structure of programs, which should result in a metrics for code quality.

7 References

[1] A. Robins, J. Rountree and N. Rountree. Learning and teaching programming: A review and discussion. Computer Science Education, Vol. 13, (2003), 137-172

[2] W. Pullan, S. Drew and S. Tucker. A Problem Based Approach to Teaching Programming. In *Proc. of the 9th Int. Conf. on Frontiers in Education: Computer Science and Computer Engineering* (Las Vegas, USA), FECS'13, 403-408

[3] D. Pawelczak and A. Baumann. Virtual-C - a programming environment for teaching C in undergraduate programming courses. In *Proc. of IEEE Global Engineering Education Conference EDUCON*, (Istanbul, Turkey, April 3-5, 2014), 1142-1148

[4] J. L. Whalley and A. Philpott. A unit testing approach to building novice programmers' skills and confidence. In *Proc. of the 13th Australasian Computing Education Conf.* (Perth, Australia), ACE 2011, 113-118

[5] K. Beck, Test Driven Development: By Example. Addison-Wesley, 2002.

[6] M. T. Helmick. Interface-based programming assignments and automatic grading of Java programs. In *Proc. of the 12th annual SIGCSE conf. on Innovation and technology in computer science education* (Dundee, Scotland, UK, June 23-27, 2007), ITiCSE'07, 63-67

[7] S. H. Edwards. Using software testing to move students from trial-and-error to reflection-in-action. In *Proc. of the 35th SIGCSE technical symp. on Comp. science education* (Norfolk, Virginia, USA, March 3-7, 2004) SIGCSE '04, 26-30.

[8] D.M. de Souza, J.C. Maldonado and E.F. Barbosa. ProgTest: An environment for the submission and evaluation of programming assignments based on testing activities. In *Proc. of 24th IEEE-CS Conf. on Software Engineering Education and Training* (Waikiki, Honolulu, HI, USA, 22-24 May, 2011), CSEE&T, 1-10

[9] D. M. Zimmerman, J. R. Kiniry and F. Fairmichael. Toward Instant Gradeification. In *Proc. of 24th IEEE-CS Conf. on Software Engineering Education and Training* (Waikiki, Honolulu, HI, USA, 22-24 May, 2011), CSEE&T, 406-410

[10] Zhanyong Wan, et al. Google C++ Testing Framework – googletest: https://code.google.com/p/googletest/.

[11] D. Pawelczak: Online Detection of Source-code Plagiarism in Undergraduate Programming Courses. In *Proc. of the 9th Int. Conf. on Frontiers in Education: Computer Science and Computer Engineering* (Las Vegas, USA), FECS'13, 57-63

[12] Gómez-Martín, M. A., and Gómez-Martín, P. P. Fighting against the 'But it works!' syndrome. In *Proc. XI Int. Symp. on Computers in Education* (Coimbra, Portugal, November 18-20, 2009). SIIE'09.

[13] S. H. Edwards and Z. Shams. Do student programmers all tend to write the same software tests? In *Proc. of the 2014 conf. on Innovation & technology in computer science education* (Uppsala, Sweden, June 21-25, 2014), ITiCSE '14, 171-176