Automated Formal Testing of C API Using T2C Framework

Alexey V. Khoroshilov, Vladimir V. Rubanov, and Eugene A. Shatokhin

Institute for System Programming of Russian Academy of Sciences (ISPRAS),
B. Communisticheskaya, 25, Moscow, Russia
{khoroshilov,vrub,spectre}@ispras.ru
http://ispras.ru

Abstract. A problem of automated test development for checking basic functionality of program interfaces (API) is discussed. Different technologies and corresponding tools are surveyed. And T2C technology developed in ISPRAS is presented. The technology and associated tools facilitate development of "medium quality" (and "medium cost") tests. An important feature of T2C technology is that it enforces that each check in a developed test is explicitly linked to the corresponding place in the standard. T2C tools provide convenient means to create such linkage. The results of using T2C are considered by example of a project for testing interfaces of Linux system libraries defined by the LSB standard.

Keywords: Formal testing, compliance testing, parameterized tests, medium-quality tests.

1 Introduction

Verification of a complex software system, checking its correctness in each situation is a very important but an extremely difficult task. Automated testing is often used for software verification and when considering developing such tests we have to deal with the trade-off between thoroughness of the tests and the resources needed to develop, use and maintain these tests.

The optimal solution depends on many factors specific to a particular project. In this paper we consider development of tests that check program interfaces for C language ("C API") for compliance with a standardized specification (and actually for other kinds of mature developers documentation) for program interfaces. Such problem statement suggests taking the following factors into account:

- The tests need to be maintained as the standard evolves.
- The existence of a standard means assuming that the behaviour of the system under test is described in enough detail. Nevertheless, it may not be the case for fast evolving standards.
- More often than not, the inconsistencies found by the tests will be analyzed not by the tests' developer but rather by the experts from the companies that wish to check their products for compliance with the standard. So it can be crucial to facilitate analysis of such failures.

These factors lead to several requirements for test suites and thus for an approach taken to develop test suites. But they impose no restrictions on the trade-off between available resources and the thoroughness of the tests.

As far as compliance testing is concerned, testing of basic functionality is a rather common choice. To check basic functionality means to verify behaviour of the system in its several common use cases probably including some scenarios when the system must report an error (error scenarios). This path is quite attractive for verification of industrial software because it often gives a guarantee of revealing all significant violations of the standard for reasonable cost.

When it is crucial to ensure strict compliance of a software system with a standard and there are enough resources available, more thorough (”deep”) testing is chosen. For instance, the deep tests may strive to check each class of test situations possible for each aspect of functionality of each interface function. Deep tests of this kind that still remain maintainable can be created using the tools based, for example, on UniTesK technology [1] that, among other things, suggests using special model-based testing techniques. This technology is used, for instance, in CTesK tools [2]. The following features of CTesK allow to go through all the test situations and to facilitate analysis of test coverage:

- the means for formal description of the requirements for the system under test;
- support for automatic generation of test action sequences by dynamic creation of a test system behaviour model;
- support for a wide range of test quality metrics in terms of a requirement model with automated gathering of data concerning the achieved coverage.

There are alternatives, of course. When it is necessary to cover a large amount of functions in a short period of time, the decision can be made in favour of less thorough testing. Sometimes its purpose is to ensure that each of these functions can just be called with correct arguments and does not lead to a crash. One of the approaches to development of such tests (”shallow tests”), AZOV technology is described in [15].

The purpose of T2C tools described in this paper is to facilitate development of tests that check basic functionality of a software system. These tools, while being inferior to CTesK in respect of test thoroughness, allow achieving a reasonable balance between the quality of the tests and the resources needed to develop and maintain these tests. T2C tools support basic recommendations for dealing with requirements in formal testing for compliance with a standard such as creating a catalog of elementary requirements specified in the standard, ensuring traceability of the requirements in the tests and measuring test quality in terms of covered elementary requirements. T2C technology enforces that each check in a developed test is explicitly linked to the corresponding place in the standard and T2C tools provide convenient means to create such linkage.

This paper is organized as follows. In the first part several approaches to similar problems are considered and their advantages and disadvantages with respect to testing basic software functionality are discussed. Then basic features of T2C tools are described as well as the work flow they support. After that the results of applying T2C for a real-world problem are discussed, namely the experience of using this technology to develop tests for several libraries from GTK+ stack and fontconfig for compliance with the Linux Standard Base (LSB) [3]. The future directions of T2C development as well as its integration with other tools from UniTesK family are presented in the conclusion.

2 Technologies and Tools for the Development of Basic Functionality Tests for Program Interfaces

2.1 MANUAL

Test systems that ensure the functionality is checked thoroughly usually need a range of services of underlying operating system. That is why when the very operating system is under test, it is required to be relatively stable for these tests to operate properly.

To mitigate this problem and to minimize unintentional influence of test system on the target system, distributed architecture is often used in test development. This implies that most of the tasks are performed on an auxiliary (”instrumental”) machine, while only a small test agent is working on the target machine. But even in this case interaction between the test agent and the instrumental machine requires some components of the system under test to be operational.

That is why it is important to make sure that the key components of the target system are operational before proceeding to thorough testing of its program interfaces.

An approach for creating tests of basic operation system’s functionality named MANUAL was developed in software engineering department of ISPRAS for the project that involved testing of a POSIX-compliant real time operating system for embedded devices. These tests verified that the key components of the operating system are operational before the beginning of deeper testing by the system developed using CTesk tools.

A test in MANUAL is in fact code in C programming language using macros and functions of MANUAL support library. Each test is a separate function beginning with `TC_START("name of the test")` macro and ending with `TC_END()` macro. The body of the test consists of three parts:

- preparation of data and of the environment;
- test action itself and checking its results;
- deallocation of resources.

Checking the system under test for correctness is done with the function `tc_assert(expression_to_check, "text describing the failure")`. If the expression to check is false, it is assumed that an failure has been detected and a message is output that describes this failure. Besides, the system automatically catches exceptional situations that appear during the execution of the test and treats them like failures.

MANUAL system supports hierarchical composition of tests into the packages. There are two modes for running the tests: automatic and interactive. In the automatic mode the system executes the specified set of tests or packages and stores execution log. In the interactive mode the user can navigate the package tree down to an individual test and execute the chosen test or package.

The main drawback of MANUAL system is its low scalability. The scalability problems result from the fact that each test is a function in C language which, as the test suite grows, requires either multiple duplication of the source code or a significant amount of a rather tedious manual work to structure the code.

Lack of test parametrization, while reasonable when implementing simple operability checks for basic functionality of a target system, is a significant obstacle for applying this approach to development of more thorough test suites.

2.2 Check

Check system [4] is designed in the first place for unit testing of software during the development process. Nevertheless, Check can also be used for testing program interfaces for compliance with the standards.

Check provides the test developer with a set of macros and functions to perform checks in the tests, to combine the tests in suites, to manage output of the results, etc.

A test is code in C programming language enclosed between `START_TEST` and `END_TEST` macros. The requirements are checked in the tests using the following functions: `fail_unless(expression_to_check, "text describing the failure")` and `fail_if(expression_to_check, "text describing the failure")`.

Functions performing initialization and clean-up of used resources can be specified both for each particular test and for each test suite (so called *checked and unchecked fixtures*).

Advantages of Check system:

- support for running each test in a separate process, i.e. a kind of isolation of the tests from one another and from Check environment;
- automatic handling of exceptional situations in the tests;
- support for specifying maximum execution time for a test;
- special facilities for checking situations where execution of the function under test should result in sending a signal;
- integration of test building and test execution system with autoconf and automake the tools commonly used for automation of software building and installation [5].

Check, however, has several drawbacks that prevent using it in some cases:

- It is difficult to develop parameterized tests with Check. It is often the case that some function needs to be tested with different sets of its arguments' values while the code of the test remains almost the same. It could be reasonable to pass these sets of values to the test as parameters. However only

the number of the test can be explicitly passed to this test as a parameter which is not convenient.
- There is no linkage of the checks performed in a test to the places in the documentation (standard) where the corresponding requirements are stated.
- To add a new test in a suite, it is necessary to recompile the source of all the other tests of this suite too, which is not always reasonable.
- Common test results codes (”verdicts”) listed in the standard for testing compliance to POSIX [6] are not supported which may make it more difficult to analyze test results.

2.3 CUnit

CUnit system [7] can be used in the same cases as Check [4], but is generally less powerful.

One of the most important drawbacks of CUnit compared to Check is the fact that all the tests as well as the harness that executes them and collects their results run in the same process. This means that a failure in a test may, for example, lead to corruption of memory used by CUnit harness or by some other test.

Also, unlike Check, there is no protection from test ”hang-up”: maximum execution time can not be specified for a test.

Still there are some advantages CUnit has over Check:

- Support for so called *fatal and non-fatal assertions*. In the first case if a check reveals that a requirement was violated, test execution stops and thus further checks are not performed in this test (this approach is always used in Check). One the other hand, if violation of a requirement has been detected in a non-fatal assertion, test execution continues. Further checks in this test can probably provide the developer with more detailed information about what was really happening in the system under test. This may help to discover the cause of the detected failure.
- A set of special functions and macros that facilitate commonly used checks such as equality and inequality of integers, floating-point numbers, strings, etc.
- Support for reporting the test results in several formats including those that can be displayed in a web browser (xml+xslt).

Nevertheless, the drawbacks pointed out for Check in the previous section apply to CUnit too. Test Environment Toolkit (TET) described below is free from some of these.

2.4 TET (Test Environment Toolkit)

TETware system (TET, TestEnvironmentToolkit) is quite widely used for testing various program interfaces. TET tools provide a common way to run different tests and to obtain a report of the test results in a common format [8]. Data concerning test execution including its result (test verdict) and the messages it outputs is accumulated in so called *TET journal*.

TET consists of the following basic components:

- test case controller (tcc) this component manages test execution and gathering information the tests output;
- application program interface (TET API) that should be used in the tests to be able to run them within TET harness. TET API is available for several programming languages including C and C++.

The most important advantages of TET are probably the following:

- a common environment for running the tests;
- handling exceptional situations in the tests (segmentation faults, for example) by the means of the test case controller;
- common test result codes (verdicts) that comply with the standard [6]: PASS, FAIL, UNRESOLVED, etc., along with the ability to define additional test result codes;
- an ability to add new tests to the suite without recompiling the remaining tests (using so called *TET scenarios*)
- a common format for a report of test execution (*TET journal*).

These TET's features make the analysis of test execution results easier. Particularly, the program tools processing TET journal may not take the specifics of the tests into account while collecting the statistics of test results, for example.

On the other hand, TET tools have something to do mostly with automation of test execution and collecting of test results. TET provides neither means to somehow automate test development nor the API for performing checks in the tests. Consequently, there are several reasons that make using "pure" TET (without any enhancements) rather inconvenient:

- Lack of means to link the checks performed in the tests to the corresponding parts of the standard.
- It is often necessary to create tests with almost the same source code and the difference is, for example, only in the parameters passed to the functions called in this test or, say, in element types of used arrays, etc. It seems reasonable to automate development of such tests so as to reuse common parts of the source code. Unfortunately, TET provides no special means for this.
- The test developer needs to manually add definitions of special functions, data structures, etc., required to run the tests within TET harness. This could be done automatically as well.
- The tests being executed by the test case controller are not always easy to debug. It could be helpful both for searching for errors in the test itself and for investigating the behaviour of the system under test to be able to avoid TET's influence on test execution. This could facilitate the use of debugger programs such as gdb and others.

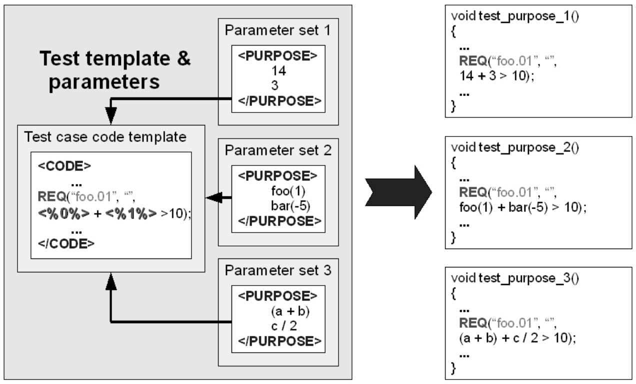
Described below are two systems that have TET as their basis: GTKVTS and T2C. These systems manage to overcome TET's drawbacks to some extent.

2.5 Automation of TET-Compliant Test Development in GTK+-2.0 Verification Test Suite (GTKVTS)

An approach used in GTK+-2.0 Verification Test Suite (GTKVTS) for development of TET-compliant tests allows avoiding some of the TET's drawbacks [9].

First of all, GTKVTS uses so called parameterized tests. That is, the developer writes a template for the test source code in plain C language just marking in some special way the places where to put the values of test's parameters. Several sets of parameter's values can be specified for each test template of this kind. Almost anything can be a test parameter, not only parameters of tested functions or their expected returns values. Sometimes its is reasonable to consider types of used data as test parameters (like C++ templates) or even to make a statement calling the target function a parameter and so on.

The GTKVTS C code generator creates a single function in C language for each set of parameters' values based on the test source template (see Fig. 1).



```
<CODE>
  ...
REQ("foo.01", "",
<%0%> + <%1%> >10);
  ...
</CODE>
```

```
<PURPOSE>
    14
    3
</PURPOSE>
```

```
<PURPOSE>
    foo(1)
    bar(-5)
</PURPOSE>
```

```
<PURPOSE>
    (a + b)
    c / 2
</PURPOSE>
```

```
void test_purpose_1()
{
  ...
  REQ("foo.01", "",
  14 + 3 > 10);
  ...
}
```

```
void test_purpose_2()
{
  ...
  REQ("foo.01", "",
  foo(1) + bar(-5) > 10);
  ...
}
```

```
void test_purpose_3()
{
  ...
  REQ("foo.01", "",
  (a + b) + c / 2 > 10);
  ...
}
```

Fig. 1. Generating C code based on a template. "<%0%>" and "<%1%>" mark the places in the template where actual values of the parameters are to be inserted.

Second, GTKVTS tools automatically insert in the generated C source code definitions of special data structures and functions required to be able to run the test within TET harness, so the developer does not have to worry about this. Besides that, makefiles for building the tests and TET scenario files are also generated automatically which can be convenient.

The authors of GTKVTS also tried to encourage linking the checks in the tests with the relevant fragments of a standard: the test developer should specify the text of the requirements checked in this test in the comments before it. Unfortunately, this text is not used in the test itself and it is usually difficult to find out from the trace of the test, which requirements have been checked and which of them have been violated.

There are also some less significant drawbacks of GTKVTS tools, such as lack of support for debugging the test outside of TET harness and the fact that the tools are specialized for developing tests for the libraries from GTK+ stack only.

2.6 Comparison of Existing Approaches

Considered above are five approaches (and corresponding tools) for test development for program interfaces in C language. The summary of their advantages and disadvantages is given in Table 1.

Table 1. Comparison of existing approaches

	MANUAL	Check	CUnit	TET	GTKVTS
Test parametrization	-	-	-	-	+
Traceability of requirements	-	-	-	-	-
Execution of tests in separate processes	-	+	-	-	-
Automatic handling of exceptional situations	+	+	-	+	+
Restriction of test execution time	-	+	-	+	+
Hierarchical package organization	+	-	-	-	-
Convenience of debugging	+	+	+	-	-
Portability of the tests	-	+	+	+	-
Using standard test verdicts [6]	-	-	-	+	+

Each of the approaches considered above has some advantages. However all of them have one significant drawback from the point of view of testing program interfaces for compliance with standards, namely lack of support for linkage of the checks in the tests to the requirements of the standard ("traceability of requirements"). In addition, none of these approaches has all the advantages described above while there seems to be no contradiction between them.

When it was decided to develop T2C tools, a requirement was stated that these tools support (and enforce to some extent rather than just encourage) traceability of requirements while still keeping the advantages of existing approaches shown in table 1 except hierarchical organization of test packages. This exception is due to the fact that the possibility to hierarchically organize the test packages does not significantly affect the development and usage of compliance tests.

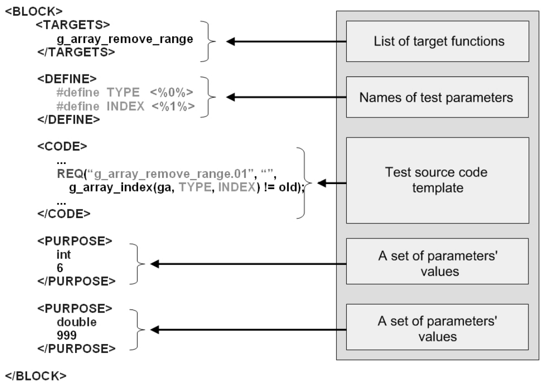
3 T2C ("Template-to-Code") System

3.1 General Information

T2C ("Template-to-Code") system facilitates the development of parameterized tests that can be executed both within and outside of TET harness.

The source code of the tests in C programming language is created based on a T2C-file that contains test templates along with the sets of parameter's values for these tests (the idea is the same as in GTKVTS - see Fig. 2). A fragment of a T2C-file is shown below.

The tests to be created based on a template presented in Fig. 2 have two parameters: `TYPE` and `INDEX`. `int` will be used as `TYPE` and `6` as `INDEX` in the first of the tests, `double` and `999`, respectively, in the second one.



```
<BLOCK>
    <TARGETS>
        g_array_remove_range
    </TARGETS>

    <DEFINE>
        #define TYPE  <%0%>
        #define INDEX <%1%>
    </DEFINE>

    <CODE>
        ...
        REQ("g_array_remove_range.01", "",
          g_array_index(ga, TYPE, INDEX) != old);
        ...
    </CODE>

    <PURPOSE>
        int
        6
    </PURPOSE>

    <PURPOSE>
        double
        999
    </PURPOSE>

</BLOCK>
```

List of target functions
Names of test parameters
Test source code template
A set of parameters' values
A set of parameters' values

Fig. 2. A fragment of a T2C-file

Like in GTKVTS, definitions of data structures and functions required to run the tests within TET harness will be added to the source of the tests automatically. Necessary makefiles and TET scenario files will be created as well.

So T2C tools retain main advantages of GTKVTS system while supporting the recommendations for development of compliance tests stated, for instance, in [10]:

- creating a catalog of elementary requirements for program interfaces to be tested;
- linkage of the checks performed in the tests to the relevant places in the corresponding standard;
- testing quality measurement in terms of covered elementary requirements.

The following enhancements have been made in T2C compared to GTKVTS:

- The test developer is provided with a set of high level program interfaces (T2C API) to perform the checks in the tests. Now if a check in the test reveals violation of some requirement, the text of this requirement is output to the test's trace (TET journal) along with other useful information.
- It is possible to create a standalone version of a test in pure C/C++ without using any of TET's features. This can be rather convenient both for debugging the test and for thorough investigation of what happens in the system under test in case of a failure.
- Templates of T2C-files (do not confuse them with the test templates) are created automatically for a text of the standard with the requirements marked up in it in a special way.
- T2C-file can also contain the code needed for initialization and cleanup of the resources used by any the tests to be generated from this file as well as for deallocation of resources allocated in each particular test.
- Execution of each test in a separate process is supported as well as in a single process.
- Maximum execution time for a test can be specified. This is useful if some of the tests may hang.

3.2 Test Development with T2C Tools: The Workflow

Main stages of test development process with T2C tools are described in this section (see also Fig. 3).

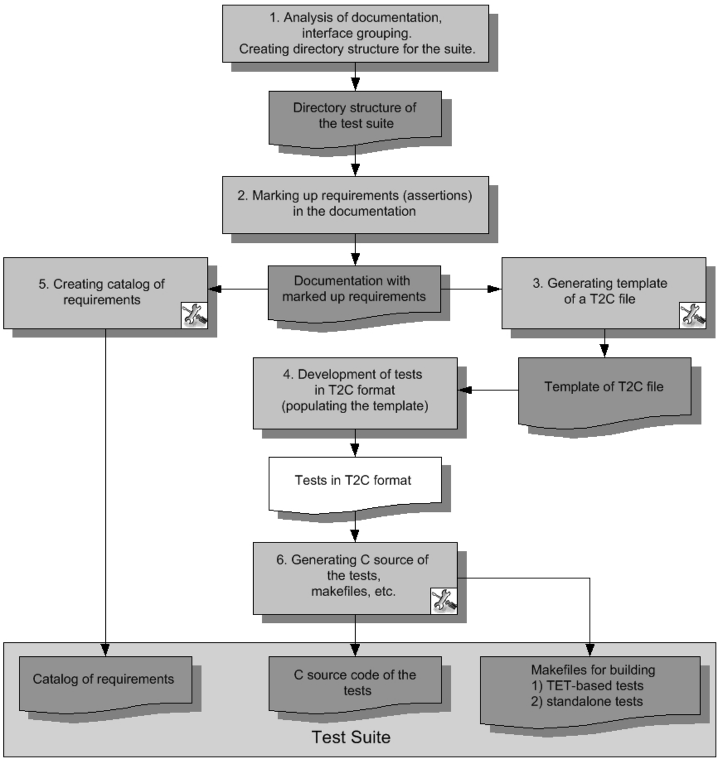

1. Analysis of documentation, interface grouping. Creating directory structure for the suite.
Directory structure of the test suite
2. Marking up requirements (assertions) in the documentation
Documentation with marked up requirements
5. Creating catalog of requirements
3. Generating template of a T2C file
Template of T2C file
4. Development of tests in T2C format (populating the template)
Tests in T2C format
6. Generating C source of the tests, makefiles, etc.
Catalog of requirements
C source code of the tests
Makefiles for building 1) TET-based tests 2) standalone tests
Test Suite

Fig. 3. Test development with T2C

Analysis of Documentation and Interface Grouping. First of all, before trying to write conformance tests for some interfaces one should examine the documentation of these interfaces to find out what is to be tested. During this analysis one should also split the interfaces to be tested into groups, each of which implements a coherent part of the system's functionality ("functional groups"). One should avoid the situations when some interfaces from group A are needed to check the interfaces from group B ("A depends on B") and in the same time interfaces from A are needed to test those from B (cyclic dependency).

Sometimes the grouping is already done in the documentation. For instance, a reference manual for Glib2 library [11] is divided in sections such as "Memory Allocation", "String Utility Functions", "Key-value file parser", etc. Interfaces described in each section usually form a single functional group.

During the test development one or more T2C-files are created for each functional group. It is often reasonable to create appropriate directory structure for the test suite at this stage too.

From now on it is assumed that the standard (documentation) for the interfaces to be tested is a set of HTML documents.

Requirement (Assertion) Markup in the Documentation. Elementary requirements for each of the interfaces to be tested are marked up in the documentation in a special way at this stage. Each elementary requirement is given a unique identifier [10]. The text of a requirement can be assembled from several parts if necessary or it can be reformulated to improve readability.

Markup of requirements is performed in HTML editor KompoZer (`www.kompozer.net`) enhanced with ReqMarkup tool that was developed during OLVER project [12] and then remodeled and integrated into T2C system.

Creating a Template of T2C-file. Once the requirements for a particular functional group of interfaces have been marked up, the ReqMarkup tool automatically creates a template for the corresponding T2C-file.

Populating the T2C File Template. This stage is the most important in the development of the tests. Now the developer should populate the template of a T2C-file adding the templates of test case source code along with the parameter's values for the tests. In addition, the code for initialization and cleanup of resources used by the tests should be specified in special sections of the file.

The T2C Editor tool a plugin for Eclipse IDE - can be helpful for visual editing of T2C files providing advanced navigation among the file's sections, convenient means for dealing with the parameters of the tests, etc.

Preparing Catalog of Elementary Requirements. Based on the documentation with the requirements marked up, ReqMarkup tool also creates a catalog of requirements for the corresponding group of interfaces. This catalog is used during the execution of the test: if it is detected that a requirement is violated, the text of the requirement with the particular identifier is loaded from the catalog and output to the test's trace for future analysis.

Generating the Source Code of the Tests, Makefiles and TET Scenarios. When the tests in T2C format are prepared and so is the catalog of elementary requirements, the developer should invoke T2C Code Generator that will create the files with source code of the tests (in C or C++ language), makefiles for building the tests from these sources, TET scenario files, etc.

Building, Executing and Debugging the Tests. At this stage the developer should build the test suite using the makefiles generated at the previous step. After that the test suite is ready. One may run the tests within TET harness or debug some of them outside of TET and so forth.

4 Applying T2C to Test Development for LSB Desktop

T2C system was used (and is used now) in development of tests for interface operations (*"interfaces"*) of Linux libraries, defined in the Linux Standard Base (LSB). For example, the tests for the following libraries were prepared using T2C tools:

- Glib (libglib-2.0);
- GModule (libgmodule-2.0);
- GThread (libgthread-2.0);
- GObject (libgobject-2.0);
- ATK (libatk-1.0);
- Fontconfig (libfontconfig).

Table 2 shows the results of testing these libraries. The descriptions of inconsistencies found by the tests are published in `http://linuxtesting.ru/results/impl_reports`.

Table 2. Results of testing several Linux libraries for compliance with LSB by the tests developed using T2C tools

Library	Version	Total interfaces	Tested interfaces	Problems found
libatk-1.0	1.19.6	222	222 (100%)	11
libglib-2.0	2.14.0	847	832 (98%)	13
libgthread-2.0	2.14.0	2	2 (100%)	0
libgobject-2.0	2.16.0	314	313 (99%)	2
libgmodule-2.0	2.14.0	8	8 (100%)	2
libfontconfig	2.4.2	160	160 (100%)	11
Total		**1553**	**1537(99%)**	**39**

Remark 1. The *"Version"* column shows the latest version of the corresponding library at the moment when the test suite was published. The number of errors found by the tests is shown for this very version of the library. There is an ongoing work on these errors in collaborations with the developers of respective libraries, so it is possible that some or even all of these errors are (or will be) fixed in newer versions.

Remark 2. The *"Total interfaces"* column shows total number of interface operations (*"interfaces"*) defined in the LSB for the particular library including undocumented ones. Almost all documented interfaces were tested.

The average costs for a full cycle of test development (from the analysis and markup of requirements to the debugged code of the tests) for a single interface are about 0.5 - 1 man-day.

It should also be mentioned that the interfaces from these libraries are not always described in detail in the documentation. In average, 2 - 3 elementary requirements were found for each interface.

Table 3. Coverage of requirements for LSB libraries

Library	Requirements	Checked requirements	Requirement coverage (%)
libatk-1.0	515	497	96%
libglib-2.0	2461	2290	93%
libgthread-2.0	2	2	100%
libgobject-2.0	1205	1014	84%
libgmodule-2.0	21	17	80%
libfontconfig	272	213	78%
Total	**4476**	**4033**	**90%**

Table 4. Code coverage data

Library	Lines of code (total)	Executed lines	Code coverage (%)
libglib-2.0	16263	12203	75.0%
libgthread-2.0	211	149	70.6%
libgobject-2.0	7000	5605	80.1%
libgmodule-2.0	270	211	78.1%
Total	**23744**	**18168**	**76.5%**

The information concerning the number of elementary requirements for tested interfaces is given in Table 3 as well as requirement coverage data.

It can be interesting to find out what portion the source code of the libraries under test the tests act upon. The code code coverage data for four libraries of glib2 group is shown in Table 4. The data was collected for glib2 package version 2.16.3 using gcov tool. The parts of these libraries not defined in the LSB were not taken into account.

5 Conclusion

The problem of testing program interfaces for compliance with their documentation (including standards) is very important for providing quality and interoperable software systems. Various technologies as well as corresponding tools are developed for this purpose that allow to somehow automate the work and make it more systematic. These approaches always have to deal with a trade-off between the quality of the tests and the cost of developing these tests. The choice is often made here based on some quite subjective factors. Meanwhile, the choice of target testing quality governs the choice of the optimal technology and tools as well, because different levels of cost and quality require different approaches. For instance, as far as deep (thorough) testing is concerned, UniTesK technology proved very useful [13], although the cost of learning the technology and of the actual test development is rather high.

This paper describes T2C technology which is oriented to efficient development of "medium level" tests checking basic functionality of program interfaces. The term "medium level" corresponds in this case to the common notion of industrial testing quality achieved in the most of the test suites analyzed by the authors (e.g., Open Group certification tests, LSB certification tests, OpenPosix tests and Linux Test Project). T2C allows raising the efficiency of development of such tests by providing the following basic features that reduce manual work for preparing the environment and duplicating the code that is not specific for a particular test:

- automatic generation of test templates based on the catalog of requirements;
- usage of named parameters in the source code of the tests with automated generation of a separate test instance for each set of parameters' values;
- a high-level API that can be used in the code of the test to check the requirements and output trace messages;
- generation of standalone tests, i.e. self-sufficient programs in C or C++ language which significantly simplifies debugging the tests as well as the tested system compared to debugging them within TET test execution environment or the like.

The execution environment for the tests created using T2C technology is based on widely used TETware tools, which facilitates integration of the tests into existing test suites and the environments that manage test execution and analysis of the results. Besides that, one of important features of T2C is systematic work with catalogs of elementary requirements and enforcing the explicit linkage of requirement checks in the tests to the relevant places in the standard and output of corresponding messages to the test execution report.

T2C technology was successfully used at the Institute for System Programming of Russian Academy of Sciences in the project [14] for development of certification tests for checking compliance of Linux libraries with the LSB standard. Presented in this paper are the statistical data concerning the developed tests, found errors in the libraries and the costs of development. The data allow concluding that the technology is efficient for development of tests of the particular quality level for various modules and libraries. The tools support C and C++ programming languages for the present, but there are no principal obstacles that prevent applying the technology to other general purpose programming languages such as C# and Java. It should be mentioned however that the availability of quite stable text of the requirements is essential for successful use of T2C because the stage of documentation analysis and preparing of the requirement catalog may require a lot more work when the quality of the documentation is low and/or when it is changed too actively.

It is planned to enhance integration of T2C tools with Eclipse IDE as well as provide means for using these tools from other popular development environments. The possibility of integration of T2C and CTesK systems will be also investigated.

References

1. Kuliamin, V.V., Petrenko, A.K., Bourdonov, I.B., Kossatchev, A.S.: UniTesK Test Suite Architecture. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, pp. 77–88. Springer, Heidelberg (2002)
2. CTESK web page, `http://www.UniTesK.com/products/ctesk`
3. The Linux Standard Base, `http://www.linux-foundation.org/en/LSB`
4. Check web page, `http://check.sourceforge.net/doc/check.html/index.html`
5. Autoconf and Automake web page, `http://www.gnu.org/software/automake/`
6. IEEE.2003.1-1992 IEEE Standard for Information Technology – Test Methods for Measuring Conformance to POSIX – Part 1: System Interfaces. IEEE, New York, NY, USA (1992) ISBN 1-55937-275-3
7. CUnit web page, `http://cunit.sourceforge.net/`
8. TETware User Guide, `http://tetworks.opengroup.org/documents/3.7/uguide.pdf`
9. GTKVTS Readme, `http://svn.gnome.org/viewvc/gtkvts/trunk/README`
10. Kuliamin, V.V., Pakulin, N.V., Petrenko, O.L., Sortov, A.A., Khoroshilov, A.V.: Formalization of requirements in practice, ISPRAS, Moscow (preprint, 2006) (in Russian)
11. Glib Reference Manual, `http://www.gtk.org/api/2.6/glib/`
12. Linux Verification Center, `http://linuxtesting.ru/`
13. UniTesK web site, `http://UniTesK.com/`
14. LSB Infrastructure project web page, `http://ispras.linux-foundation.org/`
15. AZOV Framework web page, `http://ispras.linux-foundation.org/index.php/AZOV_Framework`