



Formal Specification-Based Testing: Getting Started

Version	Date	Description of the changes	Author
1.0	28-Feb-2006	The first version.	Alexey Khoroshilov
1.1	7-Jul-2006	Requirements coverage report description added.	Alexey Khoroshilov

Introduction

This document is intended for getting started with the testing process based on formal specifications. The process is based on the UniTesK testing technology (for more information, see www.unitesk.com).

The testing process is used in the Linux Verification Centre [1] for creation of a test suite in the OLVER project. The goal of the project is to formalize the requirements of the *Linux Standard Base Core Specification 3.1* [2] for the functions of the application binary interface contained in the sections *III Base Libraries* and *IV Utility Libraries*. The above mentioned sections of the standard define the requirements to the presence and the functionality of 1532 functions of the Linux application binary interface.

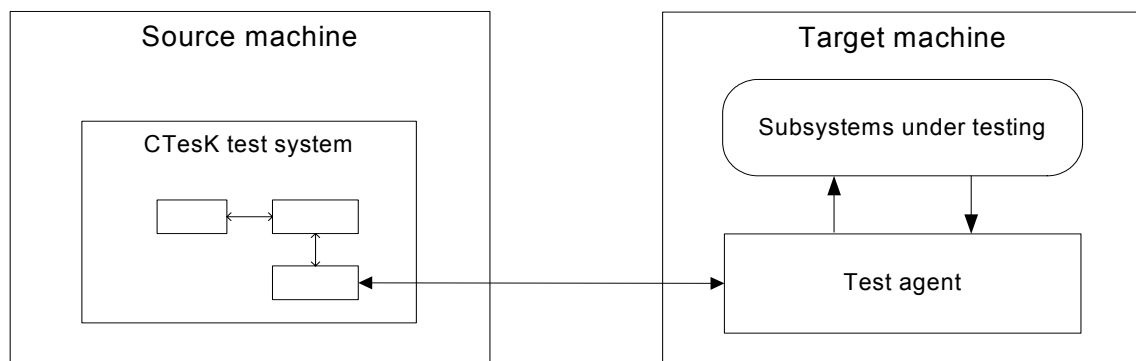
The structure of this document is as follows. The section «*Test bench architecture*» gives the general overview of the test execution process organization in the OLVER project. The subsequent sections discuss several groups of functions from the *Linux Standard Base Core Specification* standard to demonstrate the role of formal specifications in the process of functional software testing. The appendix contains the glossary of the terms and notions used in the document.

Test bench architecture

The distributed test architecture is used for creation of the test bench. According to this architecture, the system under testing runs on the *target machine*, while the test system runs on the *source machine*. Usually, these machines coincide.

To apply the test stimuli and to get information on the target system behavior, the test system uses small software components called *test agents* that run on the source machine.

All interactions between the test system and the test agents are performed through the abstract interface implemented basing on sockets in the current version. In the future, alternate implementations based on other interprocess communication mechanisms will be introduced. This will allow choosing for each test scenario the optimal implementation minimizing the impact of the test system on the operating system functionality tested by this scenario.



Testing of integer arithmetics

This section considers a small example of using formal specifications to organize the functional testing of software systems. The `abs` function calculating the absolute value of an integer number will be used as the function under testing.

Requirements of the standard

The requirements to the function `abs` are defined in the ISO POSIX 2003 [3] standard:

The Open Group Base Specifications Issue 6 IEEE Std 1003.1, 2004 Edition Copyright © 2001-2004 The IEEE and The Open Group, All Rights reserved.	
NAME	<code>abs</code> - return an integer absolute value
SYNOPSIS	<pre>#include <stdlib.h> int abs(int i);</pre>
DESCRIPTION	<p>[CX] ⓘ The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-2001 defers to the ISO C standard. ⓘ</p> <p>The <code>abs()</code> function shall compute the absolute value of its integer operand, <i>i</i>. If the result cannot be represented, the behavior is undefined.</p>
RETURN VALUE	The <code>abs()</code> function shall return the absolute value of its integer operand.
ERRORS	No errors are defined.
<i>The following sections are informative.</i>	
APPLICATION USAGE	In two's-complement representation, the absolute value of the negative integer with largest magnitude <code>{INT_MIN}</code> might not be representable.

These requirements consist of two statements:

Statement 1. The function must return the absolute value of its integer operand.

Statement 2. If the result cannot be represented as the value of the type `int`, the function behavior is undefined.

For the requirements traceability during testing, each statement gets a unique identifier. The statement 1 gets the identifier `abs.01` and corresponds to two sentences in the text of the standard. The statement 2 gets the identifier `app.abs.02` and corresponds to one sentence in the text of the standard. The `app.` prefix in the identifier of the second statement indicates that this is the requirement to the application using this function, but not to the implementation of this function. Actually, in the cases when the function behaviour is undefined, the application cannot reasonably use this function.

The Open Group Base Specifications Issue 6
IEEE Std 1003.1, 2004 Edition
Copyright © 2001-2004 The IEEE and The Open Group, All Rights reserved.

NAME



`abs` - return an integer absolute value

SYNOPSIS

```
#include <stdlib.h>

int abs(int i);
```

DESCRIPTION

[\[CX\]](#)  The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of IEEE Std 1003.1-2001 defers to the ISO C standard. 

[{abs.01}](#) The `abs()` function shall compute the absolute value of its integer operand, `i`.
[{app.abs.02}](#) If the result cannot be represented, the behavior is undefined.

RETURN VALUE

[{abs.01}](#) The `abs()` function shall return the absolute value of its integer operand.

ERRORS

No errors are defined.

The following sections are informative.

APPLICATION USAGE

In two's-complement representation, the absolute value of the negative integer with largest magnitude `{INT_MIN}` might not be representable.

Formal specifications

Formal specifications contain the machine-readable representation of the requirements to the system under testing. The specification extension of C (SeC) is used for such representation of the requirements. More information on SeC can be found in the document «CTesK 2.1: SeC Language Reference» [5].

The requirements to the function `abs` written in SeC are as follows:

```

specification
IntT abs_spec( CallContext context, IntT i )
{
    pre
    {
        /* If the result cannot be represented, the behavior is undefined. */
        /* [INFORMATIVE SECTION: APPLICATION USAGE]
        * [In two's-complement representation, the absolute value of the negative
        * integer with largest magnitude {INT_MIN} might not be representable.]
        */
        /* [Checking is implicit to avoid overflow] */
        REQ( "app.abs.02", "Absolute value shall be representable",
            (i >= 0) || (i + max_IntT) >= 0
            );
        return true;
    }
    post
    {
        /* The abs() function shall compute the absolute value of its
        * integer operand, i.
        */
        /* [Function shall return non-negative result] */
        REQ("abs.01", "Function shall return non-negative result", abs_spec >= 0);
        if( i>=0 )
        {
            /* [For non-negative i function abs() shall return i] */
            REQ( "abs.01", "For non-negative i function abs() shall return i",
                i==abs_spec
                );
        }
        else
        {
            /* [For negative i expression i+abs(i) shall equal to zero] */
            REQ( "abs.01",
                "For negative i expression i+abs(i) shall equal to zero",
                i+abs_spec == 0
                );
        }
        return true;
    }
}

```

The requirements to the behavior of the function `abs` are described with the special SeC functions called specification functions. The specification functions have the SeC keyword `specification` in their signatures.

The parameters of the specification function `abs_spec` correspond to those of the target function `abs`. The use of the type `IntT` instead of the basic type `int` is due to the distributed architecture of the test. Since the target and the source machines can have different architectures, the type `int` on the source machine can be different from the same type on the target machine. The special type `IntT` is introduced to represent on the source machine the values of the type `int` of the target machine.

The parameter `context` of the specification function `abs_spec` defines the process and its thread, in which the target function `abs` is called. This parameter is the first parameter of all specification functions developed in the OLVÉR project.

The body of the specification function `abs_spec` consists of two compound statements marked with the keywords `pre` and `post` correspondingly. Both compound statements contain the program code returning the Boolean value with the operator `return`. The compound statement marked with the keyword `pre` is called the *precondition* of the specification function. The compound statement marked with the keyword `post` is called the *postcondition* of the specification function.

The precondition defines whether the values of the input parameters of the function are valid for the target function. If the Boolean verdict returned by the precondition is true, then the values of the input parameters are considered valid, otherwise the target function cannot be called with these values of the parameters.

The function `abs` can be called with any integer value. But if the result of such call cannot be represented with the type `int`, then the function behavior is undefined. So, the application intended for correct work in any implementation conforming to the standard should not call the function `abs` with such value of the parameter. In the *Linux Standard Base Core* standard, the result cannot be represented by the type `int`, if the parameter value is negative and its absolute value is greater than the maximal value representable by the type `int`.

Let's consider the precondition of the function `abs_spec` in detail:

```
pre
{
    /* If the result cannot be represented, the behavior is undefined. */
    /* [INFORMATIVE SECTION: APPLICATION USAGE]
    * [In two's-complement representation, the absolute value of the negative
    * integer with largest magnitude {INT_MIN} might not be representable.]
    */
    /* [Checking is implicit to avoid overflow] */
    REQ( "app.abs.02", "Absolute value shall be representable",
        (i >= 0) || (i + max_IntT) >= 0
        );
    return true;
}
```

The key element of the precondition is the macro `REQ` checking the requirement with the identifier `app.abs.02`. The comment preceding this macro contains the citation from the text of the standard formulating the requirement being checked:

```
pre
{
    /* If the result cannot be represented, the behavior is undefined. */
    /* [INFORMATIVE SECTION: APPLICATION USAGE]
    * [In two's-complement representation, the absolute value of the negative
    * integer with largest magnitude {INT_MIN} might not be representable.]
    */
    /* [Checking is implicit to avoid overflow] */
    REQ( "app.abs.02", "Absolute value shall be representable",
        (i >= 0) || (i + max_IntT) >= 0
        );
    return true;
}
```

The comment in square brackets `[]` gives additional information on the requirement being checked. It can be a citation from the informative section of the standard:

```

pre
{
    /* If the result cannot be represented, the behavior is undefined. */
    /* [INFORMATIVE SECTION: APPLICATION USAGE]
    * [In two's-complement representation, the absolute value of the negative
    * integer with largest magnitude {INT_MIN} might not be representable.]
    */
    /* [Checking is implicit to avoid overflow] */
    REQ( "app.abs.02", "Absolute value shall be representable",
        (i >= 0) || (i + max_IntT) >= 0
    );
    return true;
}

```

or developer's remarks:

```

pre
{
    /* If the result cannot be represented, the behavior is undefined. */
    /* [INFORMATIVE SECTION: APPLICATION USAGE]
    * [In two's-complement representation, the absolute value of the negative
    * integer with largest magnitude {INT_MIN} might not be representable.]
    */
    /* [Checking is implicit to avoid overflow] */
    REQ( "app.abs.02", "Absolute value shall be representable",
        (i >= 0) || (i + max_IntT) >= 0
    );
    return true;
}

```

The most important part of the macro is its third parameter, which contains the Boolean expression checking the requirement of the standard. In the given example, this expression checks that either the value of the parameter `i` is non-negative, or its absolute value does not exceed the value `max_IntT` equal to the maximal value representable with the type `int` on the target machine.

```

pre
{
    /* If the result cannot be represented, the behavior is undefined. */
    /* [INFORMATIVE SECTION: APPLICATION USAGE]
    * [In two's-complement representation, the absolute value of the negative
    * integer with largest magnitude {INT_MIN} might not be representable.]
    */
    /* [Checking is implicit to avoid overflow] */
    REQ( "app.abs.02", "Absolute value shall be representable",
        (i >= 0) || (i + max_IntT) >= 0
    );
    return true;
}

```

The first parameter of the macro contains the identifier of the requirement checked by this expression, and the second parameter contains the string briefly describing the requirement. After preprocessing, the macro `REQ` is converted into the C code performing the following actions:

- tracing the fact of checking the requirement;
- checking the third parameter on being true;
- if the expression is false, then
 - tracing the fact that the requirement violation is detected;
 - return from the compound statement with the operator `return false`.

Thus, if the value of the third parameter is false, then the compound statement is exited, and the operator following the macro `REQ` is not executed at all.

The postcondition is the key element of the formal description of the requirements to the target function behavior. It should give the unambiguous answer to the question: is the behavior of the target function correct or not? To do this, it should determine whether the returned value is correct for the given values of the input parameters. If the returned value is considered correct, then the postcondition exits with the operator `return true`, otherwise it exits with the operator `return false`.

The postcondition of the function `abs_spec` is set up as follows. The sequence of the macros `REQ` checks the requirements to the returned value. If all these requirements are met, then the postcondition returns true:

```
post
{
    /* The abs() function shall compute the absolute value of its
     * integer operand, i.
     */
    /* [Function shall return non-negative result] */
    REQ("abs.01", "Function shall return non-negative result", abs_spec >= 0);
    if( i >= 0 )
    {
        /* [For non-negative i function abs() shall return i] */
        REQ( "abs.01", "For non-negative i function abs() shall return i",
            i == abs_spec
            );
    }
    else
    {
        /* [For negative i expression i+abs(i) shall equal to zero] */
        REQ( "abs.01",
            "For negative i expression i+abs(i) shall equal to zero",
            i + abs_spec == 0
            );
    }
    return true;
}
```


Binding of the formal specification with the system under test

To bind the specification function formally describing the requirements to the target function with the particular implementation of this target function, a special kind of functions called *mediator functions* is introduced in SeC. For each specification function, the corresponding mediator function is defined performing the actions necessary to call the target function.

In the distributed test architecture, the call of the target function consists of the following steps:

1. The mediator function interacts with the test agent to call the target function with the given values of parameters.
2. The test agent calls the target function.
3. The target function returns control and the returned value.
4. The test agent passes the returned value to the mediator function.
5. The mediator function returns the received value.

The mediator function for the specification function `abs_spec` looks as follows:

```
mediator abs_media for specification
IntT abs_spec( CallContext context, IntT i )
{
    call
    {
        TSCCommand command = create_TSCCommand();
        IntT res = 0;

        format_TSCCommand( &command, "abs:${int}", create_IntTObj(i) );
        executeCommandInContext( context, &command );
        if (!isBadVerdict())
        {
            timestamp = command.meta.timestamp;
            res = readInt_TSStream(&command.response);
        }

        destroy_TSCCommand(&command);

        return res;
    }
}
```

The key elements here are the forming of the command to the test agent:

```
mediator abs_media for specification
IntT abs_spec( CallContext context, IntT i )
{
    call
    {
        TSCCommand command = create_TSCCommand();
        IntT res = 0;

        format_TSCCommand( &command, "abs:$(int)", create_IntTObj(i) );
        executeCommandInContext( context, &command );
        if (!isBadVerdict())
        {
            timestamp = command.meta.timestamp;
            res = readInt_TSStream(&command.response);
        }

        destroy_TSCCommand(&command);

        return res;
    }
}
```

decoding of the returned value:

```
mediator abs_media for specification
IntT abs_spec( CallContext context, IntT i )
{
    call
    {
        TSCCommand command = create_TSCCommand();
        IntT res = 0;

        format_TSCCommand( &command, "abs:$(int)", create_IntTObj(i) );
        executeCommandInContext( context, &command );
        if (!isBadVerdict())
        {
            timestamp = command.meta.timestamp;
            res = readInt_TSStream(&command.response);
        }

        destroy_TSCCommand(&command);

        return res;
    }
}
```

and returning of the decoded value:

```

mediator abs_media for specification
IntT abs_spec( CallContext context, IntT i )
{
    call
    {
        TSCommand command = create_TSCommand();
        IntT res = 0;

        format_TSCommand( &command, "abs:$(int)", create_IntTObj(i) );
        executeCommandInContext( context, &command );
        if (!isBadVerdict())
        {
            timestamp = command.meta.timestamp;
            res = readInt_TSStream(&command.response);
        }

        destroy_TSCommand(&command);

        return res;
    }
}

```

The remaining code is auxiliary for interacting with the test agent, allocating and releasing the necessary resources.

Test agent

The test agent consists of a common part and a set of commands. The common part interacts with the test system and calls the certain commands on the test system request. Each command is a function in C with the predefined signature:

```

typedef
TACommandVerdict (*CommandProcessorRoutine)(TAThread thread,TAInputStream stream);

```

Consider the command for the function abs:

```

static TACommandVerdict abs_cmd(TAThread thread,TAInputStream stream)
{
    int i,res;

    /* Prepare */
    i = readInt(&stream);

    /* Execute */
    START_TARGET_OPERATION(thread);
    res = abs(i);
    END_TARGET_OPERATION(thread);

    /* Response */
    writeInt(thread, res);
    sendResponse(thread);

    return taDefaultVerdict;
}

```

The command reads the parameters:

```
static TACommandVerdict abs_cmd(TAThread thread,TAInputStream stream)
{
int i,res;

/* Prepare */
i = readInt(&stream);

/* Execute */
START_TARGET_OPERATION(thread);
res = abs(i);
END_TARGET_OPERATION(thread);

/* Response */
writeInt(thread, res);
sendResponse(thread);

return taDefaultVerdict;
}
```

calls the target function:

```
static TACommandVerdict abs_cmd(TAThread thread,TAInputStream stream)
{
int i,res;

/* Prepare */
i = readInt(&stream);

/* Execute */
START_TARGET_OPERATION(thread);
res = abs(i);
END_TARGET_OPERATION(thread);

/* Response */
writeInt(thread, res);
sendResponse(thread);

return taDefaultVerdict;
}
```

stores the values received from the target function and sends them to the test system:

```
static TACommandVerdict abs_cmd(TAThread thread,TAInputStream stream)
{
  int i,res;

  /* Prepare */
  i = readInt(&stream);

  /* Execute */
  START_TARGET_OPERATION(thread);
  res = abs(i);
  END_TARGET_OPERATION(thread);

  /* Response */
  writeInt(thread, res);
  sendResponse(thread);

  return taDefaultVerdict;
}
```

For the common part of the test agent to know which command to call when calling the function `abs`, we register the command `abs_cmd` with the function `ta_register_command`.

```
void register_math_integer_commands(void)
{
  ta_register_command("abs",abs_cmd);
  ...
}
```

Test scenarios

Test scenarios define the test data for which the correctness of the target function behavior should be checked during testing.

For example, we select the following values for testing the function `abs` and arrange them in an array:

```
IntT abs_seeds[] = {
  INT_MAX, INT_MIN+1, 0, 15, 1685, 0x5A755AC0, -19, -543829
};
int num_abs_seeds = sizeof(abs_seeds) / sizeof(abs_seeds[0]);
```

Then, we write the sequence of calls as a scenario function:

```
scenario
bool abs_scen()
{
  iterate(int i=0; i < num_abs_seeds; i++;)
  {
    abs_spec(getContext(), abs_seeds[i]);
  }
  return true;
}
```

The scenario function `abs_scen` defines the sequence of calls of the specification function with each element of the array `abs_seeds`. The operator `iterate` defined in SeC is used as a loop operator. In this case, use of this operator is almost equivalent to use of the C operator `for`.

The test scenario is defined in SeC as follows:

```
scenario dfsm abs_scenario =
{
    .actions = {
        abs_scen,
        NULL
    }
};
```

The test scenario `abs_scenario` contains the only scenario function `abs_scen` defining the sequence of calls of the specification function `abs_spec`. Each call of the specification function results in the call of the target function, after which the correctness of the target function behavior is automatically checked basing on the postcondition.

Function main

To run the test, we define the function `main` as follows:

```
int main(int argc, char** argv)
{
    initTestSystem();
    loadSUT();

    /* Run test scenario */
    abs_scenario(argc,argv);

    return 0;
}
```

Here, we initialize the test system:

```
int main(int argc, char** argv)
{
    initTestSystem();
    loadSUT();

    /* Run test scenario */
    abs_scenario(argc,argv);

    return 0;
}
```

establish connection with the target machine:

```
int main(int argc, char** argv)
{
    initTestSystem();
    loadSUT();

    /* Run test scenario */
    abs_scenario(argc,argv);

    return 0;
}
```

and start the test scenario:

```
int main(int argc, char** argv)
{
    initTestSystem();
    loadSUT();

    /* Run test scenario */
    abs_scenario(argc, argv);

    return 0;
}
```

Quality of testing

Several methods are used to estimate the quality of testing. First, the coverage of the basic requirements of the standard determined at the first step is checked. Details about requirements coverage report see below.

Another mechanism for estimation of the testing quality uses the partitioning of the domain of the specification functions parameters into the equivalence classes. The example of such partitioning for the function `abs_spec` is given below:

```

specification
IntT abs_spec( CallContext context, IntT i )
{
  pre
  {
    /* If the result cannot be represented, the behavior is undefined. */
    /* [INFORMATIVE SECTION: APPLICATION USAGE]
     * [In two's-complement representation, the absolute value of the negative
     * integer with largest magnitude {INT_MIN} might not be representable.]
     */
    /* [Checking is implicit to avoid overflow] */
    REQ( "app.abs.02", "Absolute value shall be representable",
        (i >= 0) || (i + max_IntT) >= 0
        );
    return true;
  }
  coverage C
  {
    if (i == 0)
      return { Zero, "Zero parameter" };
    else if (i > 0)
    {
      if (i == max_IntT)
        return { IntMax, "INT_MAX" };
      return { Positive, "Positive value" };
    }
    else /* [i<0] */
      return { Negative, "Negative value" };
  }
  post
  {
    /* The abs() function shall compute the absolute value of its
     * integer operand, i.
     */
    /* [Function shall return non-negative result] */
    REQ("abs.01", "Function shall return non-negative result", abs_spec >= 0);
    if( i>=0 )
    {
      /* [For non-negative i function abs() shall return i] */
      REQ( "abs.01", "For non-negative i function abs() shall return i",
          i==abs_spec
          );
    }
    else
    {
      /* [For negative i expression i+abs(i) shall equal to zero] */
      REQ( "abs.01",
          "For negative i expression i+abs(i) shall equal to zero",
          i+abs_spec == 0
          );
    }
    return true;
  }
}

```

The equivalence classes are defined with the special compound statement placed in the body of the specification function and marked with the keyword **coverage**. In the function `abs_spec`, this statement defines 4 equivalence classes with the identifiers `Zero`, `IntMax`, `Positive` and `Negative`.

The definition of the equivalence classes enables the automatic tracking of hitting these classes during testing and subsequent generation of the report on their coverage.

Building and running the test

The detailed description of the process of building the test is given in the file `readme.txt` located in the root directory of the archive.

To build the test, execute the command:

```
> build_test.sh
```

To run the test, in the file `run_tests.sh` comment out the lines starting with `run`, except for the line `run integer_scenario`, and execute the command:

```
> run_tests.sh
```

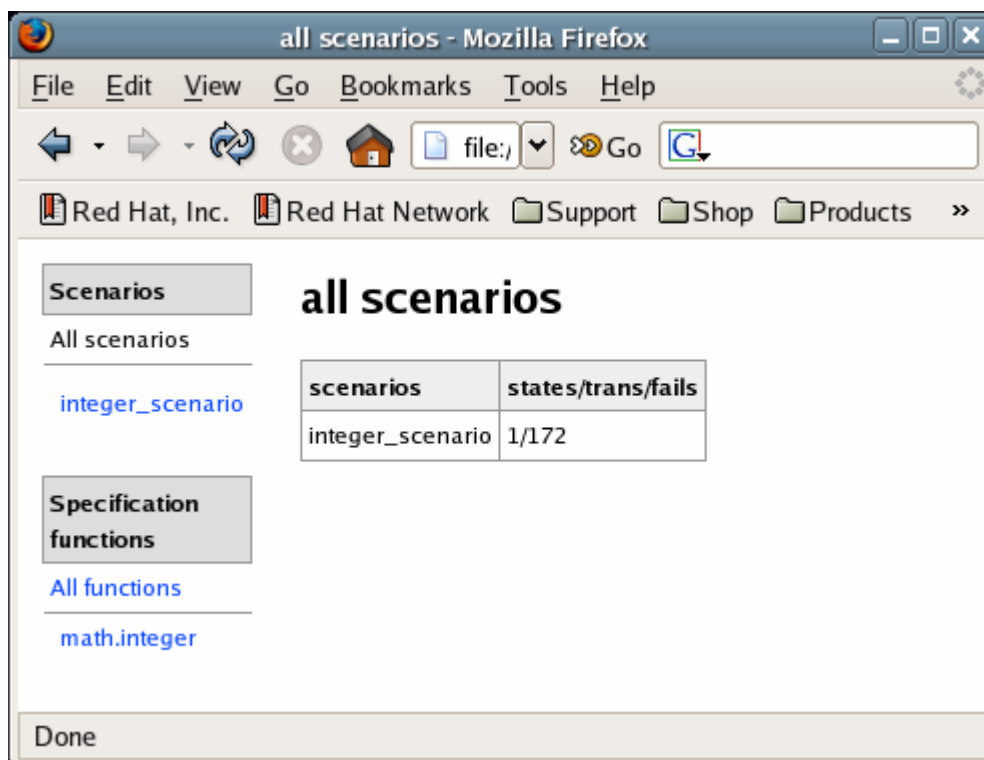
Analysis of results

The information on the testing process and the results of the test scenario execution is stored automatically in the file `integer_scenario_%Y-%m-%d_%H-%M-%S.utt` located in the directory `results/%Y-%m-%d_%H-%M-%S`, where the letters preceded with the percent sign denote the date and the time of the test start.

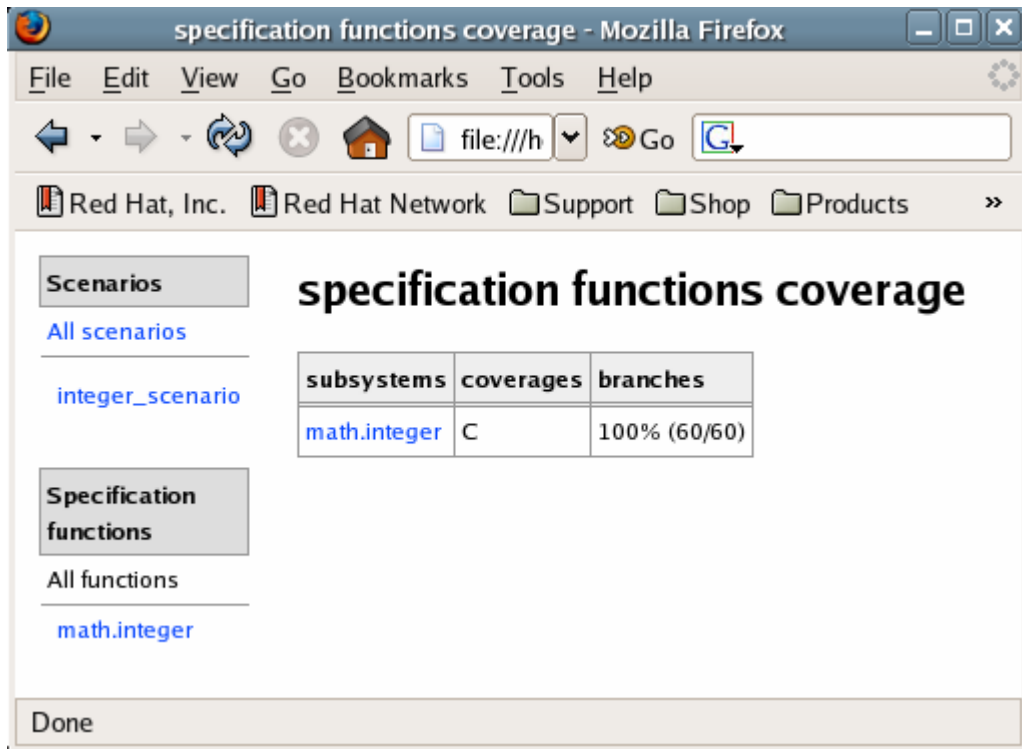
Two kinds of reports will be generated automatically after test execution finishes. The first one is Test Execution Report and the second one is Requirements Coverage Report.

Test Execution Report

Test Execution Report is placed at the directory `results/%Y-%m-%d_%H-%M-%S/report`. The enter point of the report is the file `results/%Y-%m-%d_%H-%M-%S/report/index.html`.

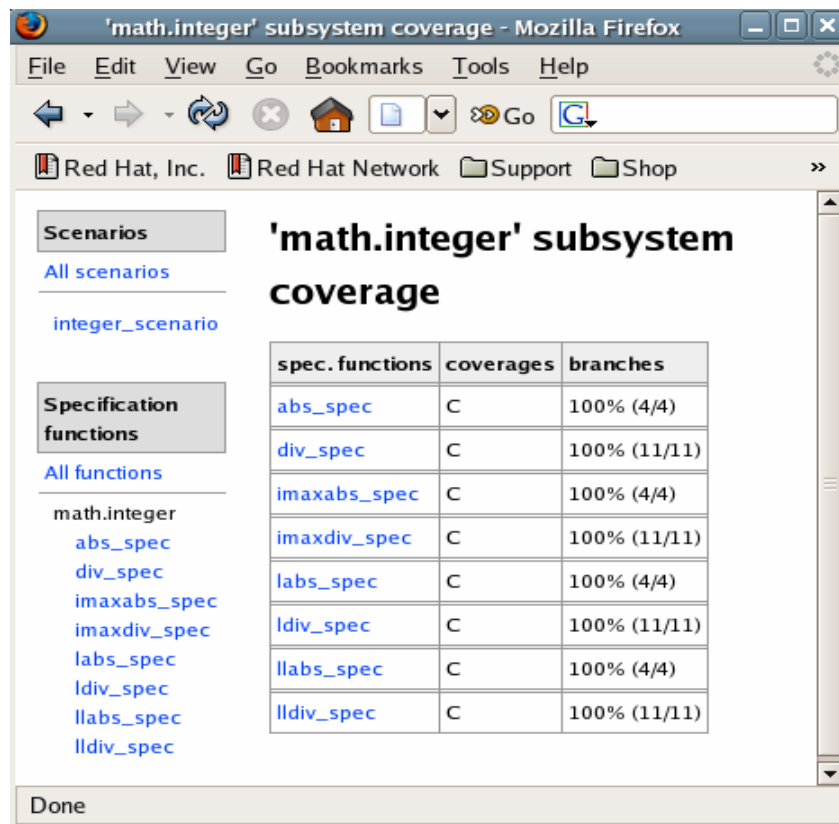


The link `All functions` points to the total coverage report:



The report shows the percentage of the achieved coverage of the functional branches for each subsystem tested in this test. In this case, during testing of the subsystem `math.integer`, all 60 equivalence classes of total 60 ones defined in this subsystem were covered.

The link `math.integer` points to the total coverage report for the subsystem `math.integer`:



The link `abs_spec` points to the total coverage report for the specification function `abs_spec`:

The screenshot shows a Mozilla Firefox browser window displaying a coverage report for the function `abs_spec()`. The browser's address bar shows a file path. The page content includes a sidebar with navigation links for 'Scenarios' and 'Specification functions'. The main content area displays the function signature and a table of coverage data for the 'C' category.

coverages	branches	hits
C	Zero parameter	1
	INT_MAX	1
	Positive value	3
	Negative value	3
	100% (4/4)	8

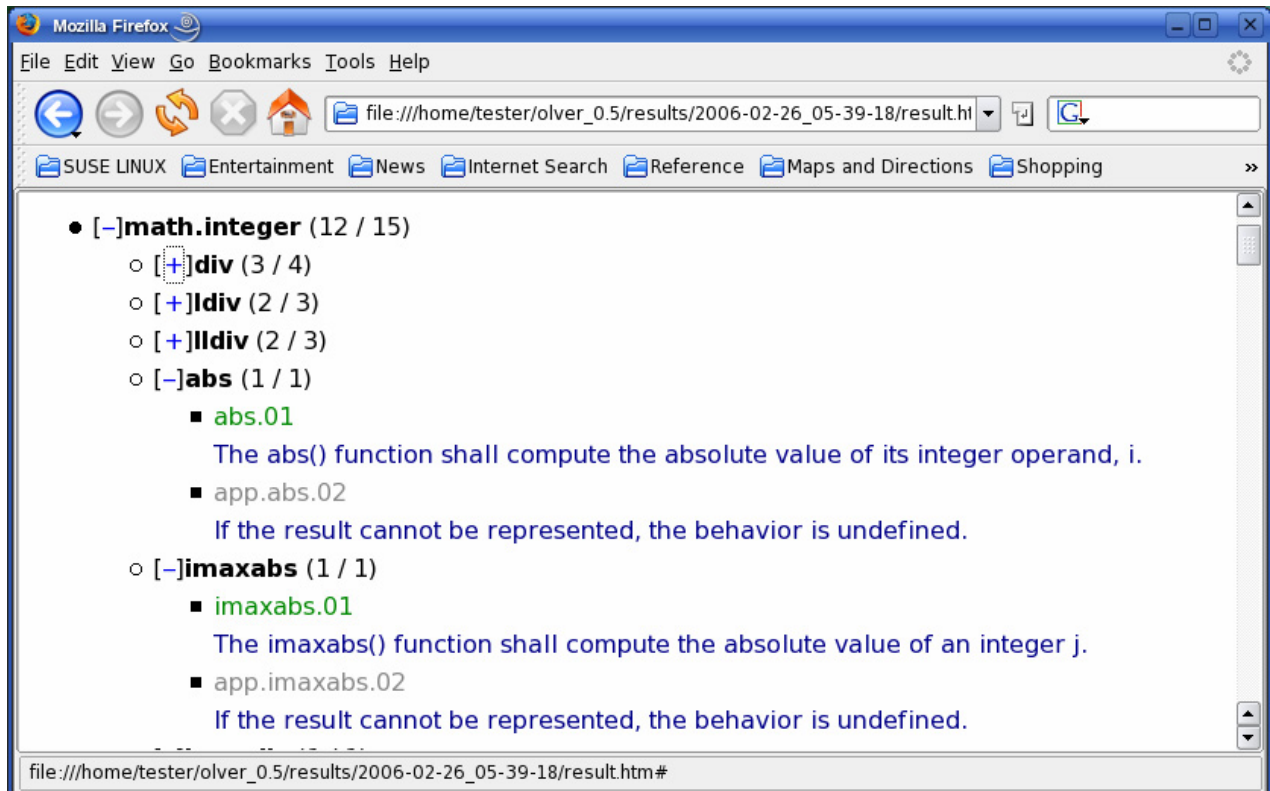
This report shows the coverage of the equivalence classes defined in the specification function `abs_spec`. The covered classes are highlighted with the green background of the table cells. In this case, all classes are covered. The right column of the table contains the number of hits during testing for each equivalence class.

Requirements Coverage Report

Requirement Coverage Report is placed at the file `results/%Y-%m-%d_%H-%M-%S/result.htm`. It contains information about coverage of atomic requirements of the standard. Each requirement presents as a leaf in the requirements tree. Nodes of the tree correspond to interface functions and subsystems.

Atomic requirements covered during the test execution are marked by green colour, noncovered atomic requirements are marked by red colour. Requirements to the application, which has "app." prefix, are marked by grey colour. At last requirements marked by yellow colour were covered during testing, but the coverage was not confirmed.

For example, the `abs.01` requirement was covered by a test execution. So it is marked by green colour as demonstrated at the following picture:



References

- [1] Linux Verification Centre (<http://www.linuxtesting.org>)
- [2] *Linux Standard Base Core Specification 3.1*
(http://refspecs.freestandards.org/LSB_3.1.0/LSB-Core-generic/LSB-Core-generic.html)
- [3] The UniTesK technology web site (<http://www.unitesk.com>)
- [4] *ISO/IEC 9945-2:2003 Information technology -- Portable Operating System Interface (POSIX) -- Part 2: System Interfaces* (<http://www.unix.org/version3/>)
- [5] CTesK 2.1: SeC Language Reference
(<http://www.unitesk.com/download/papers/ctesk/CTesK2.1LanguageReference.eng.pdf>)

Glossary

Target system – the software system to be tested. This term is often abbreviated as SUT (System Under Test) and IUT (Implementation Under Test).

Test system – the software suite intended for testing the target system.

Source machine – the platform where the main part of the test system runs.

Target machine – the platform where the target system runs.

Test agent – part of the test system running on the target machine and intended for performing test actions and getting information on the target system behavior.