Тестирование на основе формальных спецификаций: быстрое знакомство

Версия	Дата изменения	Описание изменения	Автор
1.0	28.02.2006	Первая версия документа.	Хорошилов А.В.

Введение

Настоящий документ предназначен для быстрого знакомства с процессом тестирования на основе формальных спецификаций. Этот процесс базируется на технологии тестирования UniTesK, дополнительную информацию о которой можно найти на сайте www.unitesk.com.

Рассматриваемый процесс используется в Центре Верификации ОС Linux [1] для построения тестового набора, в рамках проекта OLVER. Целью этого проекта является формализация требований стандарта *Linux Standard Base Core Specification 3.1* [2] относительно функций прикладного бинарного интерфейса, описанных в разделах *III Base Libraries* и *IV Utility Libraries*. Вышеозначенные разделы стандарта определяют требования к наличию и функциональности 1532 функций прикладного бинарного интерфейса операционной системы Linux.

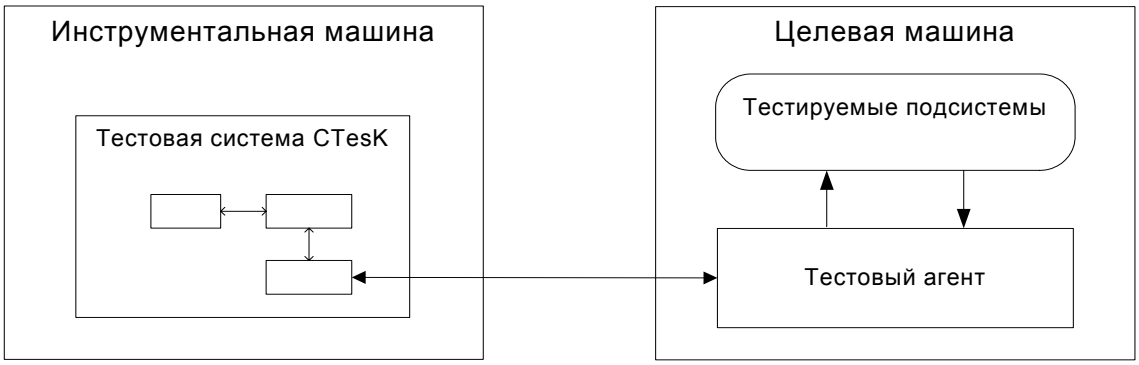
Документ построен по следующей схеме. Раздел «*Архитектура тестового стенда*» дает общее представление об организации процесса выполнения тестов, принятого в проекте OLVER. В последующих разделах рассматривается несколько групп функций из стандарта *Linux Standard Base Core Specification*, на примере которых демонстрируется роль формальных спецификаций в процессе функционального тестирования программных систем. В приложении к документу представлен глоссарий, содержащий определения терминов, используемых в данном документе.

Архитектура тестового стенда

Для построения тестового стенда используется распределенная архитектура теста. Согласно этой архитектуре тестируемая система работает на *целевой машине*, а тестовая система выполняется на *инструментальной машине*. Обычно эти машины совпадают.

Для осуществления тестовых воздействий и для получения информации о поведении тестируемой системы тестовая система использует небольшие программные компоненты, работающие на целевой машине и получившие название *тестовых агентов*.

Все взаимодействия между тестовой системой и тестовыми агентами выполняются через абстрактный интерфейс, который в текущей версии реализован на основе сокетов. В дальнейшем предполагается разработать альтернативные реализации на базе других механизмов межпроцессного взаимодействия. Это позволит выбирать для каждого тестового сценария оптимальную реализацию, минимизирующую влияние тестовой системы на функциональность операционной системы, тестируемую данным сценарием.



Тестирование целочисленной арифметики

В настоящем разделе мы рассмотрим на простейшем примере, как используются формальные спецификации для организации функционального тестирования программных систем. В качестве тестируемой функции будет выступать функция `abs`, вычисляющая абсолютное значение целого числа.

Требования стандарта

Требования к функции `abs` представлены в стандарте ISO POSIX 2003 [3]:

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.

The *abs*() function shall compute the absolute value of its integer operand, *i*. If the result cannot be represented, the behavior is undefined.

RETURN VALUE

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.

Настоящие требования состоят из двух утверждений:

Утверждение 1. Функция должна вернуть абсолютное значение своего целочисленного операнда.

Утверждение 2. Если результат не может быть представлен значением типа `int`, то поведение функции не определено.

С целью обеспечения прослеживаемости требований стандарта в процессе тестирования каждое утверждение получает уникальный идентификатор. Утверждению 1, которое получило идентификатор `abs.01`, соответствует два предложения в тексте стандарта. Утверждению 2, получившему идентификатор `app.abs.02`, соответствует одно предложение. Префикс `app.` в идентификаторе второго утверждения сигнализирует о том, что это требование рассматривается как требование к приложению, использующему данную функцию, а не как требование к реализации этой функции. Действительно, в тех случаях, когда поведение функции не определено, приложение не может использовать функцию разумным образом.

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.

Формальные спецификации

Формальные спецификации содержат машиночитаемое представление требований к тестируемой системе. Для записи этих требований используется Спецификационное расширение языка C (SeC), детальная информация о котором представлена в документе «*CTesK 2.1. Справочник языка спецификации SeC*»[5].

Требования к функции `abs`, записанные формальным образом на SeC, представлены ниже:

```
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;
    }
}
```

Требования к поведению функции `abs` описаны при помощи специального вида функций SeC, называемых спецификационными функциями. Спецификационные функции характеризуются присутствием в сигнатуре функции ключевого слова SeC **`specification`**.

Параметры спецификационной функции `abs_spec` соответствуют параметрам целевой функции `abs`. Использование типа `IntT` вместо базового типа `int` обусловлено распределенной архитектурой теста. Так как целевая и инструментальная машины могут иметь различные архитектуры, то тип `int` на инструментальной машине может не совпадать с типом `int` на целевой машине. Поэтому вводится специальный тип `IntT`, предназначенный для представления на инструментальной машине значений типа `int` целевой машины.

Параметр `context` спецификационной функции `abs_spec` определяет, в каком потоке управления какого процесса целевой машины вызывается целевая функция `abs`. Этот параметр

является первым параметром всех спецификационных функций, разрабатываемых в рамках проекта OLVER.

Тело спецификационной функции `abs_spec` состоит из двух составных операторов, помеченных ключевыми словами **`pre`** и **`post`** соответственно. В обоих составных операторах содержится код, который возвращает булевское значение при помощи оператора **`return`**. Составной оператор, помеченный ключевым словом **`pre`**, называется *предусловием* спецификационной функции. Составной оператор, помеченный ключевым словом **`post`**, называется *постусловием* спецификационной функции.

Предусловие определяет, являются ли значения входных параметров функции допустимыми для передачи целевой функции. Если булевский вердикт, возвращаемый из предусловия, является истинным, то считается, что значения входных параметров являются допустимыми, иначе считается, что с такими значениями к целевой функции обращаться нельзя.

К функции `abs` можно обращаться с любым целочисленным значением, но если результат такого обращения не представим в типе `int`, то поведение функции не определено. Поэтому приложение, которое предназначено для корректной работы на любой реализации, удовлетворяющей стандарту, не должно вызывать функцию `abs` с таким значением параметра. В рамках стандарта *Linux Standard Base Core* результат не может быть представлен в типе `int`, если параметр отрицателен и его абсолютное значение превосходит максимальное значение представимое типом `int`.

Рассмотрим подробнее предусловие функции `abs_spec`:

```
    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;
    }
```

Ключевым элементом предусловия является макрос `REQ`, который проверяет требование с идентификатором `app.abs.02`. Комментарии, предваряющие этот макрос, содержат цитату из текста стандарта, формулирующую проверяемое требование:

```
    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;
    }
```

Комментарии, содержащиеся в квадратных скобках [], представляют дополнительную информацию, касающуюся проверяемого требования. Это может быть цитата из информативной части стандарта:

```
    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;
    }
```

или авторские заметки разработчика:

```
    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;
    }
```

Наиболее важной частью макроса является его третий параметр, который содержит булевское выражение, осуществляющее проверку требования стандарта. В данном примере это выражение проверяет, что либо значение параметра `i` неотрицательно, либо оно по модулю не превосходит значение `max_IntT`, которое равняется максимальному значению, представимому типом `int` на целевой машине.

```
    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;
    }
```

Первый параметр макроса содержит идентификатор требования, проверяемого данным выражением, а второй параметр – строку, кратко описывающую осуществляемую проверку. После работы препроцессора макрос `REQ` преобразуется в код на языке C, выполняющий следующие действия:

- трассировку того факта, что данное требование проверялось;
- проверку третьего параметра на истинность;
- если выражение ложно, то
  - трассировку того факта, что обнаружено нарушение требования;

- выход из составного оператора при помощи оператора **`return false`**.

Таким образом, если значение третьего параметра оказывается ложным, то выполняется выход из предусловия, и следующий оператор за макросом `REQ` уже не выполняется.

Постусловие является ключевым элементом в формальном определении требований к поведению целевой функции. Постусловие должно дать однозначный ответ на вопрос: является ли поведение целевой функции корректным или нет? Для этого оно должно определить, является ли возвращаемое значение корректным для данных входных значений параметров. Если возвращаемое значение считается корректным, то постусловие завершается оператором **`return true`**, если нет – то оператором **`return false`**.

Постусловие функции `abs_spec` устроено следующим образом. Последовательность макросов `REQ` проверяет требования к возвращаемому значению, и если все эти требования выполнены, то возвращается истина:

```
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;
}
```

Связывание формальной спецификации с тестируемой системой

Для связывания спецификационных функций, в которых формальным образом описываются требования к целевой функции, с конкретной реализацией целевой функции в SeC предназначен специальный вид функций, называемых медиаторными функциями. Для каждой спецификационной функции определяется соответствующая медиаторная функция, которая осуществляет необходимые действия по вызову целевой функции.

В рамках распределенной архитектуры теста вызов целевой функции состоит из следующих шагов:

1. Медиаторная функция обращается к тестовому агенту с указанием вызвать целевую функцию с данными значениями параметров.
2. Тестовый агент вызывает целевую функцию.
3. Целевая функция возвращает управление и возвращаемое значение.
4. Тестовый агент передает возвращаемое значение медиаторной функции.
5. Медиаторная функция возвращает полученное значение.

Медиаторная функция для спецификационной функции `abs_spec` выглядит следующим образом:

```
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;
    }
}
```

Ключевыми элементами здесь являются формирование команды для тестового агента:

```
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;
    }
}
```

декодирование возвращаемого значения:

```
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;
    }
}
```

и возврат декодированного значения:

```
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;
    }
}
```

Остальной код играет вспомогательную роль по осуществлению взаимодействия с тестовым агентом, получению и освобождению необходимых ресурсов.

Тестовый агент

Тестовый агент состоит из общей части и набора команд. Общая часть осуществляет взаимодействие с тестовой системой и вызывает по ее указанию определенные команды. Каждая команда представляет собой функцию на языке C с заданной сигнатурой:

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

Рассмотрим команду для функции `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;
}
```

Команда читает параметры:

```
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;
}
```

вызывает целевую функцию:

```
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;
}
```

записывает полученные от целевой функции значения и посылает их тестовой системе:

```
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;
}
```

Чтобы общая часть тестового агента знала, какую команду необходимо выполнить для вызова функции `abs`, мы зарегистрируем команду `abs_cmd` при помощи функции `ta_register_command`.

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

Тестовые сценарии

Тестовые сценарии определяют тестовые данные, на которых необходимо проверить корректность работы целевой функции в процессе тестирования.

Например, мы выберем следующие значения для тестирования функции `abs` и оформим их в виде массива:

```
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]);
```

Далее мы оформим последовательность вызовов в виде сценарной функции:

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

Сценарная функция `abs_scen` определяет последовательность вызовов спецификационной функции с каждым элементом массива `abs_seeds`. В качестве оператора цикла используется

оператор SeC **iterate**. В данном случае использование этого оператора практически эквивалентно использованию оператору **for** языка C.

Тестовый сценарий определяется при помощи следующей конструкции SeC:

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

Тестовый сценарий `abs_scenario` содержит одну единственную сценарную функцию `abs_scen`, которая определяет последовательность вызовов спецификационной функции `abs_spec`. Каждый вызов спецификационной функции приводит к вызову целевой функции, после которого автоматически проверяется корректность поведения целевой функции на основании постусловия.

Функция main

Для запуска теста мы определим функцию `main` следующим образом:

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

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

    return 0;
}
```

Здесь мы инициализируем тестовую систему:

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

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

    return 0;
}
```

устанавливаем соединение с целевой машиной:

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

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

    return 0;
}
```

и запускаем тестовый сценарий:

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

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

    return 0;
}
```

Качество тестирования

Для оценки качества тестирования используется несколько методов. В первую очередь оценивается покрытие атомарных требований стандарта, выделенных на первом шаге. В текущей версии тестового набора автоматическая генерация отчета о покрытии требований не доступна. Появление этой возможности планируется на весну 2006 года.

Другой механизм для оценки качества тестирования работает на основе разбиения пространства входных значений спецификационных функций на классы эквивалентности. Пример такого разбиения для функции `abs_spec` представлен ниже:

```
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;
    }
}
```

Определение классов эквивалентности выполнено посредством специального составного оператора, находящегося в теле спецификационной функции и помеченного ключевым словом **coverage**. В функции `abs_spec` такой оператор задает 4 класса эквивалентности с идентификаторами `Zero`, `IntMax`, `Positive` и `Negative` соответственно.

Основной целью определения классов эквивалентности является автоматическое отслеживание попаданий в эти классы в процессе тестирования и последующая генерация отчета об их покрытии.

Сборка и запуск теста

Подробное описание процесса сборки теста представлено в файле `readme.txt`, находящемся в корневом каталоге архива.

Для сборки теста выполните команду:

```
> build_test.sh
```

Для запуска теста закомментируйте в файле `run_tests.sh` все сроки, начинающиеся с `run`, кроме строки `run integer_scenario`, и выполните команду:

```
> run_tests.sh
```

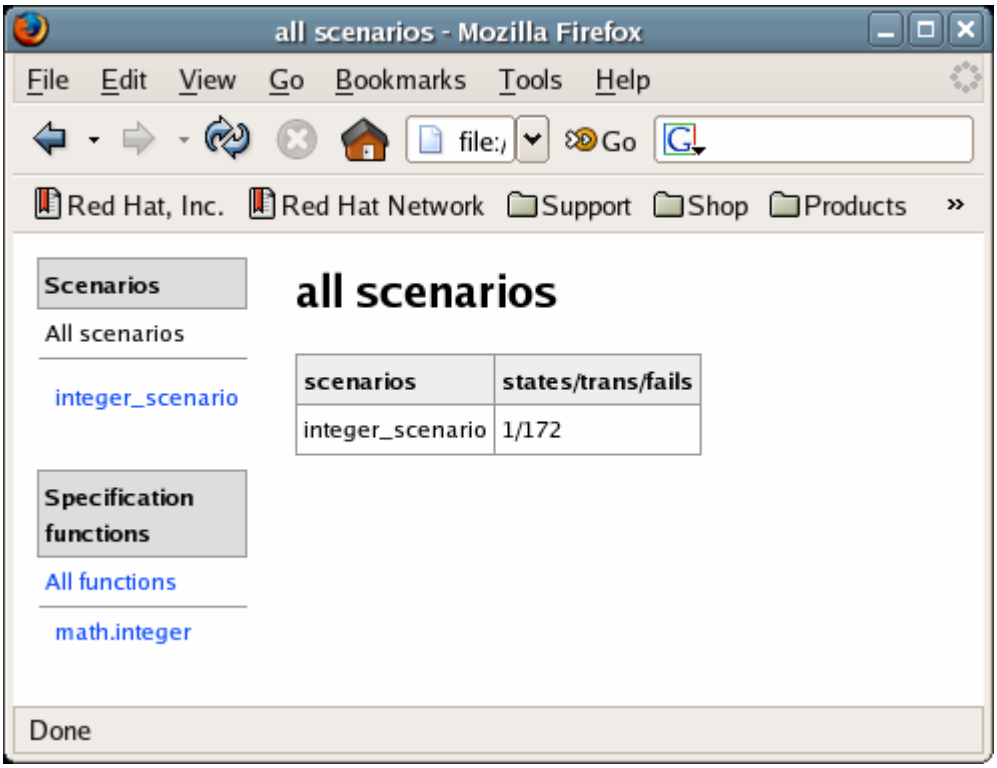
Анализ результатов

Информация о процессе тестирования и результатах работы тестового сценария автоматически сохраняется в файле `integer_scenario_%Y-%m-%d_%H-%M-%S.utt`, находящемся в каталоге `results/%Y-%m-%d_%H-%M-%S`, где вместо букв, предваренных знаком процента, находится дата и время запуска теста.

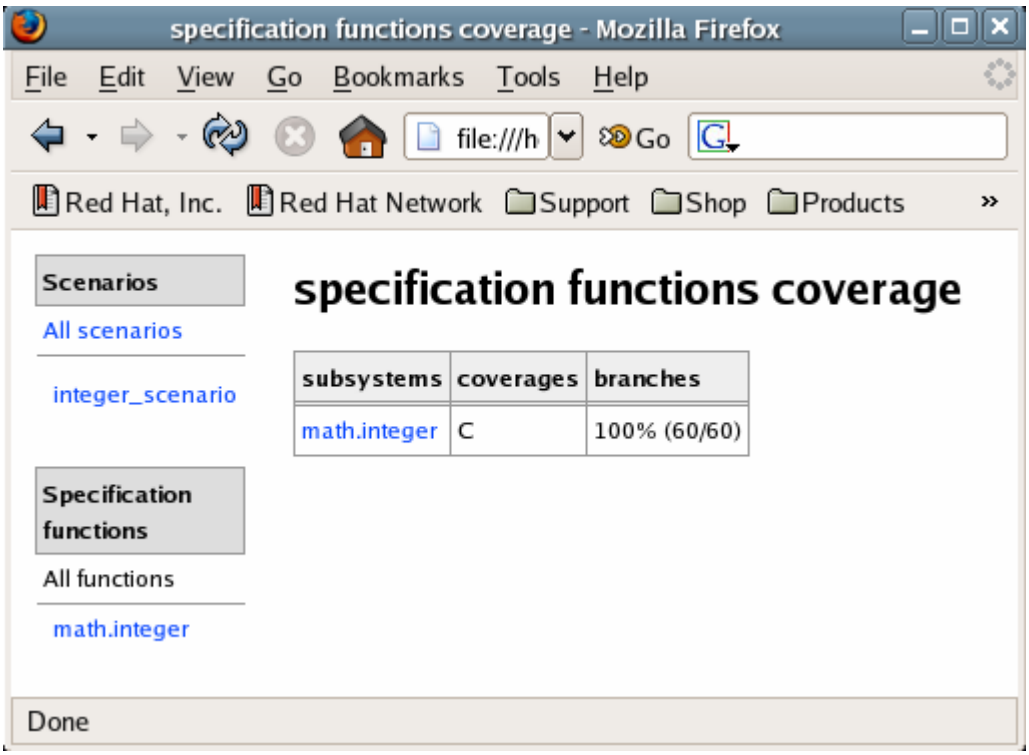
Для генерации отчета о покрытии в терминах классов эквивалентности, определенных в спецификационных функциях, предназначен инструмент `stargen`, входящий в состав системы тестирования CTesK. Чтобы воспользоваться эти инструментом, выполните команду:

```
> create-report.sh
```

После завершения работы генератора в каталоге `results/%Y-%m-%d_%H-%M-%S/report` появится отчет о результатах тестирования в формате HTML. Начальной точкой отчета является файл `results/%Y-%m-%d_%H-%M-%S/report/index.html`.

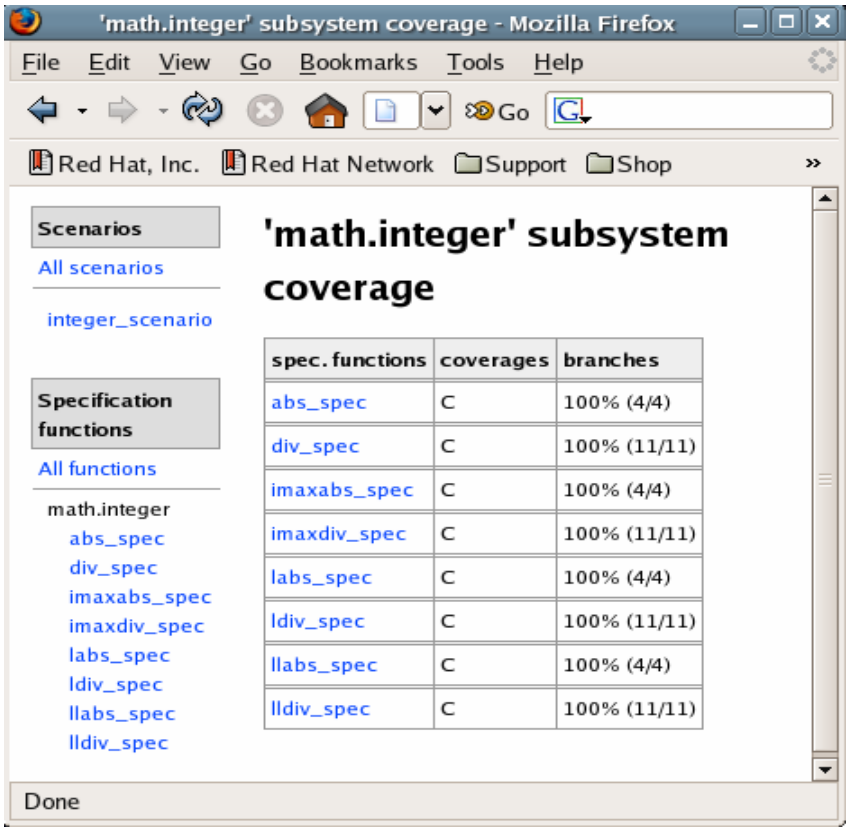


По ссылке `All functions` доступен итоговый отчет по покрытию:

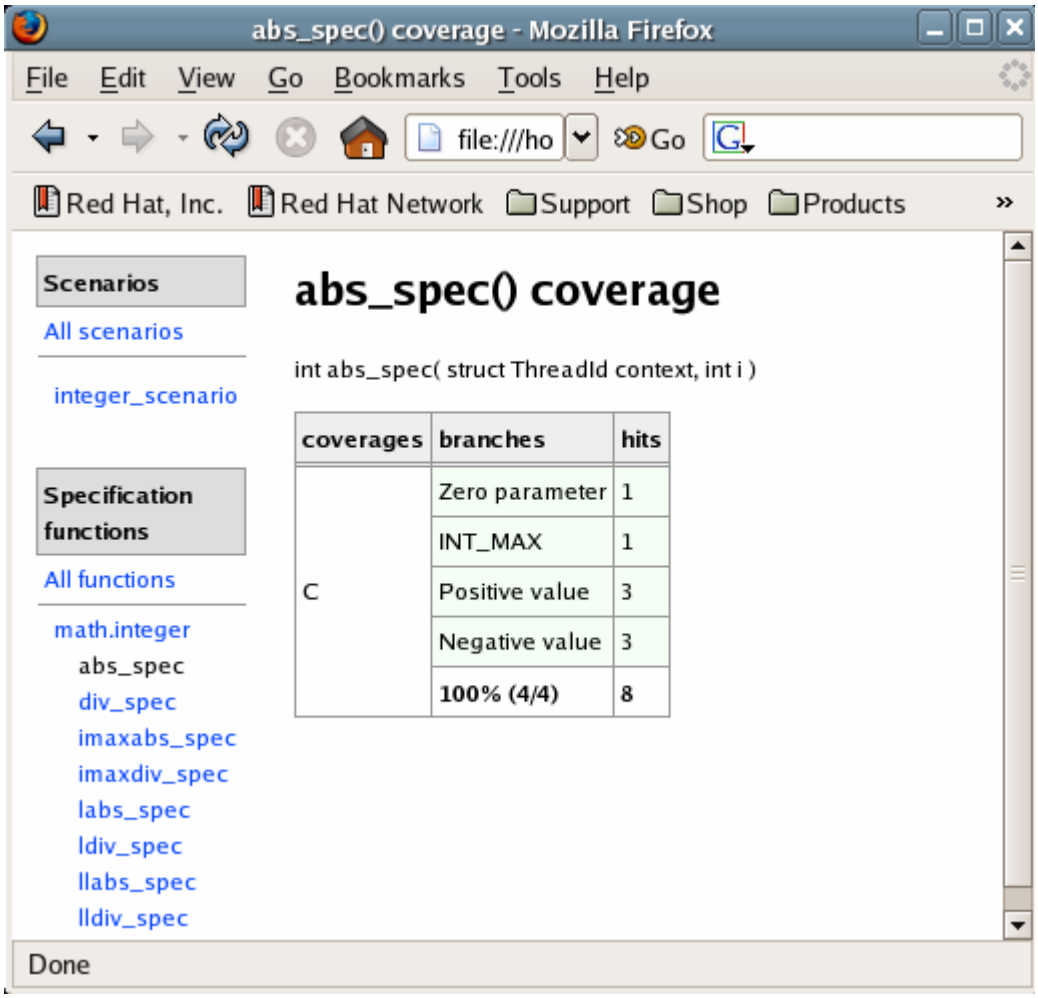


Отчет показывает процент достигнутого покрытия функциональных ветвей для каждой подсистемы, которая тестировалась в тесте. В данном случае, при тестировании подсистемы `math.integer` покрыты все 60 классов эквивалентности из 60 определенных в данной подсистеме.

По ссылке `math.integer` доступен итоговый отчет по покрытию функций подсистемы `math.integer`:



По ссылке `abs_spec` доступен итоговый отчет по покрытию классов эквивалентности, определенных в спецификационной функции `abs_spec`:



В нем отражено покрытие классов эквивалентности, определенных в спецификационной функции `abs_spec`. Покрытые классы выделяются зеленым фоном ячеек таблицы. В данном случае все классы покрыты. В правой колонке таблицы отображается число попаданий в каждый класс эквивалентности в процессе тестирования.

Дополнительная информация

[1] Центр Верификации ОС Linux (http://www.linuxtesting.ru)
[2] *Linux Standard Base Core Specification 3.1* (http://refspecs.freestandards.org/LSB_3.1.0/LSB-Core-generic/LSB-Core-generic.html)
[3] Сайт, посвященный технологии UniTesK (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. Справочник языка спецификации SeC (http://unitesk.com/papers/ctesk/CTesK2.1LanguageReference.rus.pdf)

Глоссарий

Целевая система – программная система, которую необходимо протестировать. В качестве синонима целевой системы также будем использовать словосочетание «тестируемая система». В англоязычной литературе этим терминам соответствуют сокращения SUT (System Under Test) и IUT (Implementation Under Test).

Тестовая система – комплекс программ, предназначенный для тестирования целевой системы.

Инструментальная машина – система, на которой работает основная часть тестовой системы.

Целевая машина – система, на которой работает целевая система.

Тестовый агент – часть тестовой системы, работающая на целевой машине и предназначенная для осуществления тестовых воздействий и получения информации о поведении тестируемой системы.