CTesK 2.2 Community Edition: Руководство пользователя

Copyright © 2006 Институт Системного Программирования Российской Академии Наук
Москва, 2006

Содержание

Введение ... 1

Назначение CTesK ... 1

Технология UniTesK ... 1

Реализация UniTesK в CTesK ... 4

Что содержится в документе ... 5

Другие документы ... 6

Условные обозначения ... 6

Язык SeC ... 7

Общие сведения ... 7

Типы данных ... 8

Допустимые типы SeC ... 8

Спецификационные типы ... 9

Создание новых спецификационных типов ... 12

Инварианты ... 22

Инвариант типа ... 23

Инвариант переменной ... 24

Спецификации ... 25

Спецификационные функции ... 25

Отложенные реакции ... 26

Ограничения доступа ... 27

Предусловие ... 28

Критерий покрытия ... 29

Постусловие ... 31

Медиаторы ... 32

Медиаторная функция ... 33

Блок воздействия ... 34

Блок синхронизации ... 34

Сборщик реакций ... 35

Сценарии ... 36

Сценарная функция ... 36

Оператор итерации ... 37

Переменные сценарного состояния ... 39

Функция построения сценарного состояния ... 39

Функция определения стационарности состояния ... 39

Функция сохранения модельного состояния ... 40

Функция восстановления модельного состояния ... 40

Функция инициализации ... 41
Функция завершения ... 41
Тестовый сценарий ... 42
Трансляция и сборка тестов ... 45
Трансляция SEC файлов ... 45
Стандартные макроопределения ... 46
Библиотеки CTesK ... 46
Анализ результатов тестирования и отладка тестов ... 47
Трасса теста ... 47
Сообщения ... 48
Управление трассировкой ... 49
Статические отчеты ... 51
Сценарии ... 51
Спецификационные функции ... 52
Ошибки ... 53
Анализ результатов ... 54
Поиск ошибок ... 54
Нарушение постусловия ... 55
Нарушение инварианта или ограничения доступа ... 56
Недетерминированность графа состояний ... 58
Нарушение сильной связности графа состояний ... 59
Ошибка при инициализации ... 61
Внутренние и пользовательские ошибки ... 62
Анализ полноты покрытия ... 63
Примеры использования CTesK ... 65
Системы с прикладным программным интерфейсом ... 65
Описание интерфейса целевой системы ... 65
Разработка спецификаций ... 66
Спецификационная модель данных ... 66
Спецификация поведения ... 70
Функция уничтожения очереди ... 71
Разработка медиаторных функций ... 76
Разработка тестового сценария ... 79
Главная функция теста ... 82
Сборка и запуск теста ... 83
Генерация отчета и анализ результатов ... 83
Приложение A: Код теста очереди ... 90
Реализация ... 90
queue.h ... 90
queue.c ... 91
Спецификации ... 92
queue_spec.seh ... 92

queue_spec.sec 92
Медиаторы 96
queue_media.seh 96
queue_media.sec 96
Сценарий 98
queue_scen.seh 98
queue_scen.sec 98
Функция main 100
queue_main.sec 100

Введение

Назначение CTesK

Набор инструментов CTesK предназначен для автоматизированной разработки тестов для систем, предоставляющих программный интерфейс на языке C. Тестирование ПО с помощью инструмента CTesK основано на технологии UniTesK.

Технология UniTesK

Контроль качества является важной проблемой, стоящей перед разработчиками программного обеспечения. Тестирование является наиболее известным и широко используемым способом оценки и повышения качества. Функциональность и сложность современного программного обеспечения растёт очень быстро, а его жизненный цикл увеличивается. В этих условиях трудоёмкость применения традиционных подходов к тестированию растёт, а эффективность падает. Использование технологии UniTesK помогает преодолеть эти проблемы.

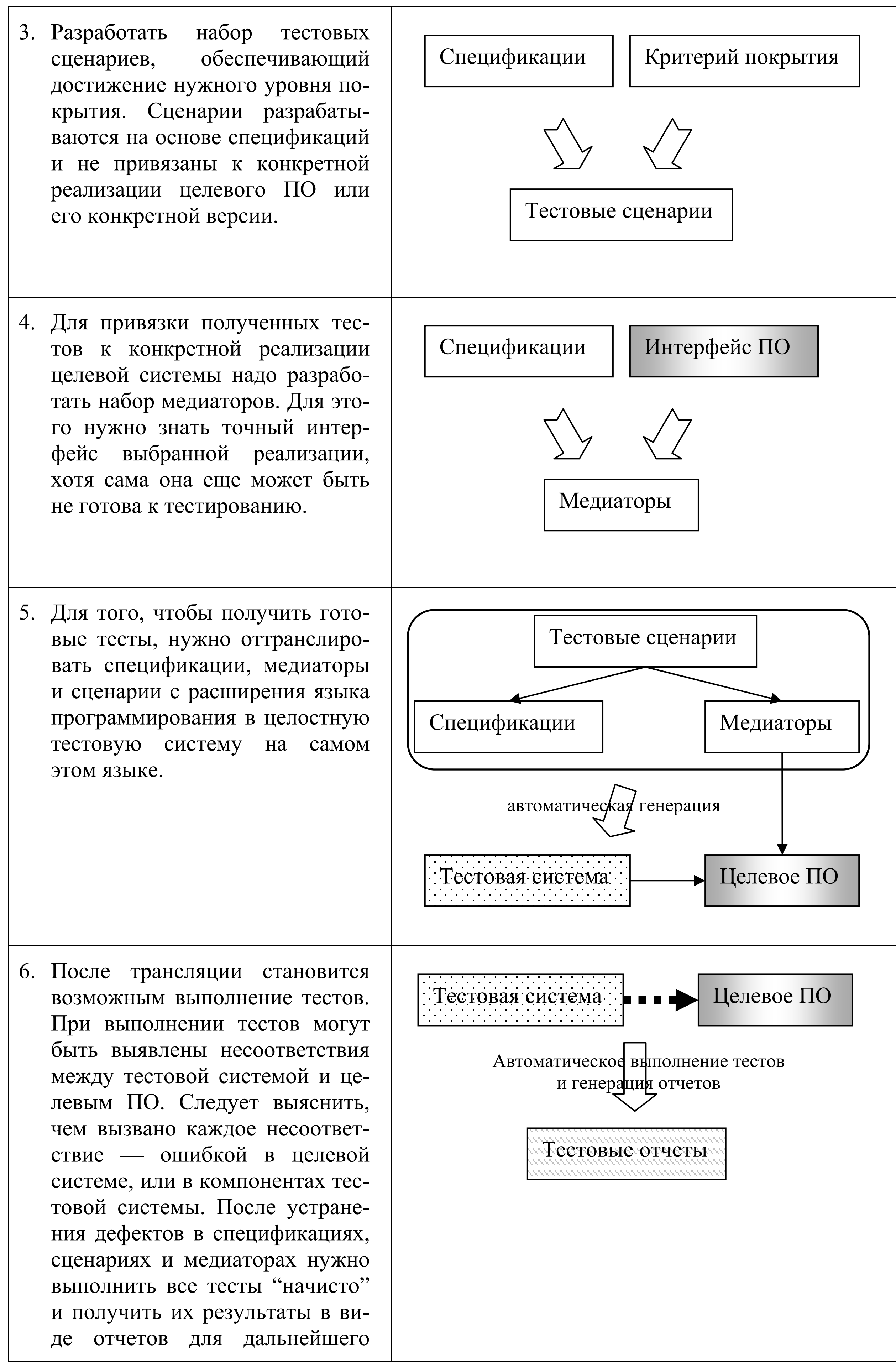
Основные характеристики технологии UniTesK таковы:

- Для чёткого определения функциональности ПО создаются *формальные спецификации*. Это может быть сделано как для вновь разрабатываемого ПО, даже до завершения реализации, так и для уже существующего. Таким образом, технология может применяться в задачах прямой и обратной инженерии ПО.
- Тесты разрабатываются на основе формальных спецификаций, а не реализации. Это позволяет проверять соответствие поведения ПО предъявляемым требованиям. Такое тестирование называют методом “чёрного ящика”. Оно позволяет создать набор тестов, не учитывающий особенности конкретной реализации.

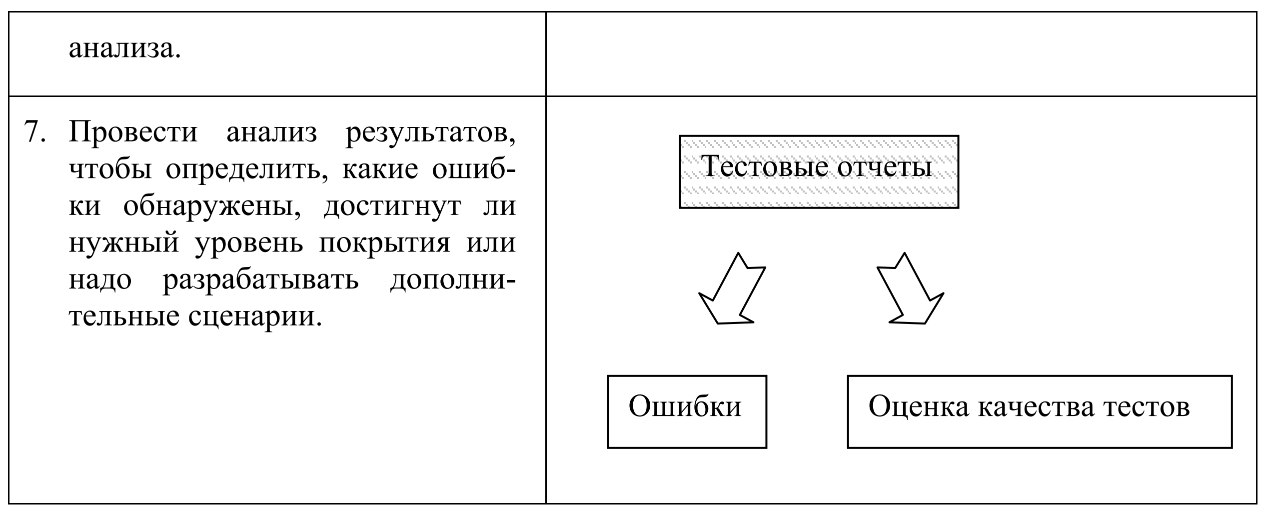
- *Критерии тестового покрытия* также строятся на основе формальных спецификаций. Эти критерии позволяют оценить, насколько полно проверено соответствие поведения ПО заданным требованиям.
- Для выбранного критерия тестового покрытия строится *сценарий тестирования*, который нацелен на достижение максимального покрытия по выбранному критерию. Аналогом сценариев тестирования являются широко используемые тестовые скрипты. Однако сценарии тестирования UniTesK позволяют значительно улучшить качество тестирования при тех же усилиях на разработку.
- Формальные спецификации и сценарии тестирования могут быть использованы в неизменном виде для тестирования различных реализаций, даже если их интерфейсы отличаются. Сопряжение реализации и тестов осуществляется с помощью специальных компонентов тестовой системы — *медиаторов*. Такой подход позволяет увеличить степень переиспользования компонентов тестовой системы, облегчая разработку и сопровождение набора тестов.
- Для записи всех компонентов тестовой системы (формальных спецификаций, медиаторов и тестовых сценариев) используется спецификационное расширение языка программирования, который используется для разработки тестируемого ПО. Это значительно облегчает освоение технологии и понимание связи теста с тестируемой системой.

Действия, которые необходимо выполнить для разработки теста по технологии UniTesK, описаны в следующей таблице:

1. Проанализировав функциональные требования к целевому ПО на основе имеющихся документов или знаний участников проекта, надо записать их в виде формальных спецификаций.	Требования → Спецификации
2. На основе полученных спецификаций сформулировать требования к качеству тестирования — какой уровень покрытия по какому критерию будет считаться достаточным, чтобы прекратить тестирование.	Спецификации → Критерий покрытия



3. Разработать набор тестовых сценариев, обеспечивающий достижение нужного уровня покрытия. Сценарии разрабатываются на основе спецификаций и не привязаны к конкретной реализации целевого ПО или его конкретной версии.	Спецификации Критерий покрытия Тестовые сценарии
4. Для привязки полученных тестов к конкретной реализации целевой системы надо разработать набор медиаторов. Для этого нужно знать точный интерфейс выбранной реализации, хотя сама она еще может быть не готова к тестированию.	Спецификации Интерфейс ПО Медиаторы
5. Для того, чтобы получить готовые тесты, нужно оттранслировать спецификации, медиаторы и сценарии с расширения языка программирования в целостную тестовую систему на самом этом языке.	Тестовые сценарии Спецификации Медиаторы автоматическая генерация Тестовая система Целевое ПО
6. После трансляции становится возможным выполнение тестов. При выполнении тестов могут быть выявлены несоответствия между тестовой системой и целевым ПО. Следует выяснить, чем вызвано каждое несоответствие — ошибкой в целевой системе, или в компонентах тестовой системы. После устранения дефектов в спецификациях, сценариях и медиаторах нужно выполнить все тесты “начисто” и получить их результаты в виде отчетов для дальнейшего	Тестовая система Целевое ПО Автоматическое выполнение тестов и генерация отчетов Тестовые отчеты

анализа.	
7. Провести анализ результатов, чтобы определить, какие ошибки обнаружены, достигнут ли нужный уровень покрытия или надо разрабатывать дополнительные сценарии.	

Этапы разработки тестовых сценариев и медиаторов не зависят друг от друга, поэтому шаги 3 и 4 могут выполняться в любом порядке или даже одновременно.

Реализация UniTesK в CTesK

CTesK реализует UniTesK на платформе языка программирования C.

В CTesK для разработки тестов используется язык *SeC* — специально разработанное спецификационное расширение языка программирования C (Specification Extension of C). SeC расширяет язык C специальными конструкциями, которые позволяют компактным и удобным образом описывать требования к тестируемой системе и другие компоненты тестовой системы. Это делает разработку тестов максимально удобной, а также позволяет сократить затраты на обучение специалистов, уже знакомых с языком программирования C. Так же SeC позволяет определять полностью независимые от реализации спецификации и сценарии, что дает возможность их повторного использования.

Набор инструментов CTesK включает транслятор из SeC в C, библиотеку поддержки тестовой системы, библиотеку спецификационных типов и генераторы тестовых отчетов.

Транслятор из SeC в C позволяет генерировать компоненты тестов из спецификаций, медиаторов и тестовых сценариев.

Библиотека поддержки тестовой системы предоставляет *обходчик* — реализацию на языке C алгоритмов построения тестовой последовательности — и обеспечивает трассировку выполнения тестов.

Библиотека спецификационных типов поддерживает типы, интегрированные со стандартными функциями создания, копирования, сравнения и уничтожения данных этих типов, и содержит набор уже определенных спецификационных типов.

Генераторы текстовых и графических тестовых отчетов дают возможность генерации удобных для анализа представлений трассы выполнения теста.

Что содержится в документе

В главе "*Язык SeC*" подробно рассматривается спецификационное расширение языка C.

- В разделе "*Общие сведения*" приводятся отличия SeC от C, описываются расширяющие C понятия и конструкции.
- В разделе "*Типы данных*" вводится концепция *допустимых* и *спецификационных типов*, в деталях рассматривается работа с существующими спецификационными типами и правила создания новых типов данных. Описывается механизм *инвариантов*, накладываемых на типы данных и переменные.
- В разделе "*Спецификации*" рассматривается способ формального описания требований к тестируемой системе в виде *предусловий, постусловий* и *ограничений доступа* в *спецификационных функциях* и *отложенных реакциях*. Описывается способ задания *критерия покрытия* с помощью описания *ветвей функциональности*.
- В разделе "*Медиаторы*" объясняется механизм связывания спецификации с реализацией тестируемой системы в виде *медиаторных функций*, осуществляющих *воздействие* и *синхронизацию состояний*.
- В разделе "*Сценарии*" описано построение *тестового сценария*, объединяющего набор *сценарных функций* для итерации параметров, механизм построения теста, функцию вычисления *сценарного состояния*, способ инициализации и завершения работы тестовой и тестируемой систем.

В главе "*Анализ результатов тестирования и отладка тестов*" рассматриваются вопросы, связанные с анализом информации, полученной при запуске теста.

- В разделе "*Трасса теста*" описывается формат *трассы*, генерируемой при выполнении теста.
- В разделе "*Статические отчеты*" рассматриваются html-отчеты, содержащие информацию о найденных ошибках, о покрытии ветвей функциональности спецификационных функций, о структуре пройденного тестом конечного автомата.
- В разделе "*Графические представления трассы*" рассматриваются различные представления трассы как последовательности событий, происходивших во время выполнения теста.
- В разделе "*Анализ результатов*" описываются различные сообщения, которые могут появиться в отчете, и способы поиска ошибок по полученной информации. Объясняется способ оценки достигнутого в ходе тестирования покрытия.

В главе "*Примеры использования CTesK*" собраны полностью описанные примеры тестирования различных систем.

- В разделе "*Системы с прикладным программным интерфейсом*" рассматривается тестирование систем, предоставляющих прикладной программный интерфейс, на примере системы, обеспечивающей работу с очередью.

Другие документы

Дополнительную информацию по CTesK и поддерживаемой технологии разработки тестов можно найти в других документах, включенных в набор документации по CTesK 2.2 Community Edition: “*CTesK 2.2 Community Edition: Инструкция по установке и удалению*”, “*CTesK 2.2 Community Edition: Быстрое знакомство*”, “*CTesK 2.2 Community Edition: Описание языка SeC*”.

Сайт www.unitesk.com содержит информацию по UniTesK, CTesK и другим инструментам, поддерживающим UniTesK.

Также с любыми вопросами по технологии UniTesK и использованию CTesK можно обращаться по электронному адресу support@unitesk.com.

Условные обозначения

Курсивом выделяются термины основных понятий и части текста, содержащие важную информацию.

“*Курсивом в кавычках*” выделяются ссылки на разделы этого документа и другие документы по CTesK.

```
В абзацах с отступом представлены примеры кода на SeC.
```

`Шрифтом с фиксшированной шириной` выделяются фрагменты кода, появляющиеся в основном тексте. **`Полужирным шрифтом с фиксированной шириной`** — ключевые слова SeC.

Полужирный шрифт используется для выделения элементов меню, команд и имен файлов и каталогов.

Язык SeC

Общие сведения

SeC полностью поддерживает ANSI C. Дополнительно вводятся *спецификационные типы*, *типы* и *переменные с инвариантами*, и четыре вида функций: *спецификационные*, *реакции*, *медиаторные* и *сценарные*. Эти типы, инварианты и функции определяются в спецификационных файлах с расширением **sec**. Спецификационные заголовочные файлы, содержащие декларации спецификационных типов и функций должны находится в файлах с расширением **seh**.

Спецификационные заголовочные файлы включаются в спецификационные файлы при помощи директивы препроцессора C `#include`. Спецификационные файлы могут содержать и обычные функции C, требуемые для различных вспомогательных целей. При необходимости использования типов данных, констант, переменных или функций включаются обычные заголовочные файлы языка C.

Для удобства записи и чтения логических выражений в язык SeC дополнительно введен оператор импликации **=>**, являющийся бинарным инфиксным оператором, приоритет которого меньше приоритета оператора дизъюнкции `||`, но больше приоритета условного оператора `?:`. Выражение `x => y` эквивалентно выражению `!x || y`, и при его вычислении, также как при вычислении других логических операторов, действуют правила короткой логики. Оператор импликации ассоциативен слева направо, то есть выражение `x => y => z` эквивалентно выражению `(x => y)` **=>** `z`.

Типы данных

Язык SeC полностью поддерживает типы данных языка C. Кроме того, в язык SeC введены дополнительные типы и их виды:

- *Булевский тип* `bool` для представления логических значений и константы `true` и `false`.
- *Спецификационные типы*, объединяющие типы языка C с базовыми операциями работы с данными этих типов: создание, копирование, сравнение, уничтожение (см. подраздел "*Спецификационные типы*").
- *Типы с инвариантами* или *подтипы* — типы данных, множества значений которых являются подмножествами значений других типов данных, которые являются для них надтипами. Подмножество значений подтипа задается при помощи ограничений, описанных в инварианте типа (см. подраздел "*Инварианты типов*").

Допустимые типы SeC

Для аргументов и возвращаемых значений спецификационных функций, отложенных реакций и медиаторных функций, для итерационных переменных и переменных сценарного состояния сценарных функций, а так же для глобальных переменных, используемых в вышеперечисленных функциях, допускаются только следующие типы:

- *Арифметические типы* (`int`, `char`, `double`, ...).
- *Перечислимые типы* (`enum`), множество значений которых, в отличие от C, ограничено их константами и не может содержать произвольное целочисленное значение.
- *Типизированный указатель*. Указатель на любой допустимый тип. При этом предполагается, что указатель либо нулевой, либо указывает на одно единственное значение соответствующего типа. Структура указателей должна быть древовидной, т. е. не должно быть неориентированных циклов по указателям.
- *Нетипизированный указатель* (`void*`). Значения такого типа интерпретируются просто как адрес некоторой ячейки памяти.
- *Функциональный указатель*. Значения такого типа интерпретируются просто как адрес некоторой функции.
- *Структурный тип*. Структура должна иметь завершенный тип, то есть описание ее тела должно быть видно в месте ее использования. Поля структуры должны иметь допустимые типы.
- *Массив фиксированной длины*. Тип элементов массива должен быть допустим.

Те же ограничения накладываются на типы, которые являются базовыми в определениях типов с инвариантами, и на типы переменных с инвариантами. Эти ограничения позволяют тестовой системе автоматически управлять данными таких типов: размещать и освобождать память, копировать и сравнивать значения.

Далее типы, допустимые в вышеперечисленных случаях, будут называться *допустимыми типами SeC*.

Спецификационные типы

Часто ограничения, накладываемые *допустимыми типами*, оказываются слишком строгими. Например, указатель в качестве ссылки на массив или рекурсивные структуры данных с циклами по указателям не являются *допустимыми типами*. Для преодоления этих ограничений и служат *спецификационные типы*.

Также *спецификационные типы* необходимы при использовании библиотеки поддержки тестовой системы CTesK (см. главу "*Библиотека поддержки тестовой системы CTesK*" документа "*CTesK 2.2 Community Edition. Описание языка SeC*").

Спецификационный тип объединяет тип языка C с базовыми операциями работы с данными этого типа: создание, копирование, сравнение, уничтожение.

Значения спецификационных типов всегда размещаются в динамической памяти и доступны только через спецификационные ссылки — указатели соответствующих типов, которые должны быть *либо нулевыми*, *либо указывать на выделенную и инициализированную память*. То есть *не допускаются объявления переменных и параметров самих спецификационных типов (не ссылок на них), а также непосредственное использование спецификационных типов при определении структур, объединений и массивов*.

Если спецификационная ссылка при объявлении не инициализируется явно, то ей автоматически присваивается нулевое значение.

Спецификационные ссылки разыменования так же как указатели в C — при помощи операторов разыменования `*` и `->`. Результатом разыменования ссылки является l-value, тип которого совпадает с типом, на основе которого определен спецификационный тип (то есть с базовым типом в определении спецификационного типа), или типом поля спецификационной структуры. (см. подраздел "*Создание новых спецификационных типов*").

Не допускается разыменование нулевых ссылок (приводит к ошибке во время исполнения).

Не допускается разыменование спецификационных ссылок, возвращенных вызовом функций, без присваивания возвращенных ссылок в переменные (приводит к утечке памяти или ошибке во время выполнения).

```
List* l = create(type_List, type_Integer)/* List — библиотечный
                                             спецификационный тип
                                           */;
Integer* spec_i/* Integer — библиотечный спецификационный тип */;
int i;

add_List(create(type_Integer, 1));
i = *get_List(0); /* не допустимо */
spec_i = get_List(0);
i = *spec_i; /* i равно 1 */

typedef specification struct {int a; int b;} IntPair ={};

IntPair* ip = create(type_IntPair, 1, 1);
int ai = create(type_IntPair, 1, 1)->a; /* не допустимо */

ai = ip->a; /* ai равно 1 */
```

Не допускается адресная арифметика над ссылками и их индексирование.

Допускается сравнение адресов в ссылках при помощи операторов C `==` и `!=`.

Не допускается сравнение адресов в ссылках, возвращенных вызовом функций, без присваивания возвращенного результата в переменную (приводит к утечке памяти) .

```
List* l = create(type_List, type_Integer)/* List — библиотечный
                                              спецификационный тип
                                            */;
Integer* spec_i/* Integer — библиотечный спецификационный тип */;
int i;

add_List(create(type_Integer, 1));
spec_i = get_List(0);
if(get_List(0)!= NULL) /* не допустимо */
if(spec_i != NULL)
  i = *spec_i; /* i равно 1 */
```

Для спецификационных ссылок допускается приведение их типов только к указателям на типы `void` и `Object` (см. ниже), а также к спецификационным ссылкам *совместимых спецификационных типов*. *Совместимыми* являются спецификационные типы — подтипы одного и того же спецификационного типа.

Управление памятью, на которую ссылаются спецификационные ссылки, автоматизировано при помощи механизма подсчета ссылок с отслеживанием циклических зависимостей. При использовании спецификационных ссылок в операторах присваивания, передаче ссылок в качестве аргументов функций, возвращении ссылки из функции, выходе ссылки из области видимости, счетчики ссылок изменяются автоматически. Память, выделенная под значение спецификационного типа, автоматически освобождается при обнулении счетчика ссылок на это значение.

Использование указателей на спецификационные ссылки и составных типов языка C, содержащих спецификационные ссылки, не рекомендуется, так как в таких случаях не поддерживается автоматическое изменение счетчиков ссылок.

В SeC определен встроенный спецификационный тип `Object`, который является неполным спецификационным типом (incomplete specification type). Он является базовым для всех спецификационных типов, но значений этого типа быть не может. Тип ссылки `Object*` используется по аналогии с `void*`. Ссылка на любой спецификационный тип может быть преобразована к ссылке на `Object` и наоборот. Если при обратном преобразовании тип значения по ссылке не совместим с типом, к которому осуществляется преобразование, то поведение системы не определено.

Функции, реализующие базовые операции над значениями спецификационных типов, находятся в библиотеке спецификационных типов CTesK (см. раздел “*Библиотека спецификационных типов”*).

Функция создания ссылок

`Object*` **`create(const Type *type, ...)`**

В качестве первого параметра функция получает указатель на *дескриптор спецификационного типа*. Константа-дескриптор спецификационного типа всегда имеет имя, состоящее из имени типа с префиксом `type_`:

```
const Type type_имя_спецификационного_типа;
```

Остальные параметры являются параметрами инициализации типа. Они отличаются для разных типов и передаются в списке типа `va_list*` в функцию инициализации спецификационного типа (см. подраздел “*Создание новых спецификационных типов”*).

Функция выделяет память для значения спецификационного типа, обнуляет ее, вызывает функцию инициализации типа, передавая ей в списке типа `va_list*` полученные значения параметров инициализации типа, и возвращает указатель на выделенную и инициализированную память.

```
Integer* i   = create(&type_Integer, 10);
String*  str = create(&type_String, "a string");
```

В приведенном выше коде создаются и инициализируются ссылки библиотечных спецификационных типов `Integer` и `String` — спецификационного представления встроенного типа языка C `int` и строкового спецификационного типа соответственно (см. раздел “*Библиотека спецификационных типов”*). При создании ссылки типа `Integer*` функции `create()` должно передаваться целочисленное значение, которое будет храниться по ссылке. При создании ссылки типа `String*` должна быть передана обычная строка языка C.

Функции копирования значений по ссылкам

`void `**`copy(`**`Object* `**`src`**`, Object* `**`dst)`**

Функция копирует данные, находящиеся по ссылке `src`, на место данных, находящихся по ссылке `dst`. Ссылки должны быть ненулевыми и одного типа, то есть они должны иметь одинаковые дескрипторы типов. Если эти условия не выполняются, то во время выполнения происходит завершение программы с сообщением об ошибке. Перед вызовом пользовательской функции копирования функция `copy()` обнуляет память по ссылке `dst`.

```
SpecificationType* ref1 = create(...);
SpecificationType* ref2 = create(...);
...
copy(ref1,ref2);
```

В приведенном выше примере ссылки `ref1` и `ref2` после инициализации ссылаются на разные значения спецификационного типа `SpecificationType`. После вызова функции `copy()` значение по ссылке `ref2` становится эквивалентным значению по ссылке `ref1`.

`Object* `**`clone(`**`Object* `**`ref)`**

Функция выделяет память для значения типа, на который ссылается `ref`, инициализирует выделенную память значением, эквивалентным значению по ссылке `ref`, и возвращает указатель на выделенную и инициализированную память.

```
SpecificationType* ref1 = create(...);
SpecificationType* ref2 = clone(ref1);
```

Значения по ссылкам `ref1` и `ref2` становятся эквивалентными после вызова `clone()`.

Функции сравнения значений по ссылкам

`int `**`compare(`**`Object* `**`left`**`, Object* `**`right)`**

В случае эквивалентности значений по переданным ссылкам функция возвращает нулевое значение. Если значения не эквивалентны, функция возвращает ненулевое значение, которое может интерпретироваться в зависимости от типа сравниваемых значений. Например, для библиотечного типа `String` результат будет таким же, как у функции `strcmp()` для типа языка C `char*`. Если параметры имеют несравнимые типы, то есть типы ссылок неодинаковые, не являются подтипами одного типа, и тип одной ссылки не является подтипом другой (см. подраздел “*Инварианты типов*“), то функция возвращает ненулевое значение. Если одна из ссылок нулевая, а другая нет, то возвращается ненулевое значение. Если обе ссылки нулевые, то возвращается ноль.

```
/* создание двух ссылок типа SpecificationType* */
SpecificationType* ref1 = create(&type_SpecificationType);
SpecificationType* ref2 = create(&type_SpecificationType);
...
/* сравнение значений */
if (!compare(ref1,ref2)) {/* значения эквивалентны */
  ...
}
else {/* значения не эквивалентны */
  ...
}
```

`bool` **`equals(`**`Object*` **`self`**`, Object*` **`ref)`**

Функция возвращает значение `true`, если значения по переданным ссылкам эквивалентны, и `false` в противном случае. Если параметры имеют различный тип, то функция возвращает `false`. Если одна из ссылок нулевая, а другая нет, то возвращается `false`. Если обе ссылки нулевые, то возвращается `true`.

```
/* создание ссылок типа SpecificationType* */
SpecificationType* ref1 = create(&type_SpecificationType);
SpecificationType* ref2 = create(&type_SpecificationType);
...
if (equals(ref1,ref2)) {/* значения эквивалентны */
  ...
}
else {/* значения не эквивалентны */
  ...
}
```

Функция построения строкового представления значения по ссылке

`String* toString(``Object*` **`ref)`**

Функция возвращает ссылку на значение типа `String` — спецификационное представление строкового типа.

```
/* создание ссылки типа SpecificationType* */
SpecificationType* ref = create(&type_SpecificationType);
/* ссылка на значение типа String */
String* str;
...
/* преобразование *ref в строковое представление */
str = toString(ref);
/* вывод его на печать */
printf("*ref == %s\n", toCharArray_String(str);
...
```

В приведенном выше фрагменте кода используется библиотечная функция `toCharArray_String()`, возвращающая содержимое строки в виде массива типа `char`, заканчивающегося нулевым значением — `'\0'`. Эта функция возвращает указатель на внутренние данные, доступные через переданную ссылку типа `String*`. Поэтому, во-первых, для возвращаемого указателя нельзя вызывать `free()`, и, во-вторых, этим указателем нельзя пользоваться после уничтожения значения по переданной ссылке.

Создание новых спецификационных типов

Спецификационные типы вводятся с помощью обычной конструкции языка C `typedef`, помеченной ключевым словом SeC **`specification`**.

Различаются декларация спецификационного типа

`specification` `typedef` *`базовый_тип новый_тип`*`;`

и его определение, в котором должен присутствовать инициализатор

```
specification typedef базовый_тип новый_тип = {
  .init = указатель_на_функцию_инициализации
, .copy = указатель_на_функцию_копирования
, .compare = указатель_на_функцию_сравнения
, .to_string = указатель_на_функцию_преобразования_в_строку
, .enumerate = указатель_на_функцию_перечисления_ссылок
, .destroy = указатель_на_функцию_освобождающую_ресурсы
};
```

В каждой единице трансляции до использования спецификационного типа он должен быть декларирован или определен.

Определение спецификационного типа должно встречаться ровно один раз только в одной из единиц трансляции, которые собираются в единую систему.

В *определении спецификационного типа* в качестве базового типа *запрещается* использовать

- спецификационные типы, недоопределенные структуры и массивы неопределенной длины;
- структуры, объединения и массивы определенной длины, содержащие элементы типов, указанных выше;
- указатели на все вышеперечисленные типы, кроме спецификационных ссылок.

Допускается использование *недоопределенных структур* в *декларациях спецификационных типов*.

Инициализатор в определении спецификационного типа определяет, какие функции будут использоваться для базовых операций над данными спецификационного типа.

Поле `init` имеет тип `Init`:

```
typedef void (*Init)(void*, va_list*);
```

По этому указателю из библиотечной функции `create()` при создании ссылки (см. подраздел "*Функция создания ссылок*") вызывается функция инициализации данного спецификационного типа. В первом параметре ей передается указатель на выделенную область памяти, которая должна быть инициализирована. Во втором аргументе передается список параметров, на основе которых инициализируются данные спецификационного типа. Список параметров строится из параметров функции `create()`, следующих за первым параметром — дескриптором типа. Поэтому параметры функции `create()` должны по типам и порядку соответствовать типам и порядку ожидаемым в функции инициализации спецификационного типа. Дескриптор типа — глобальная константа типа `Type` — неявно определяется или декларируется при определении или декларации спецификационного типа соответственно, и имеет имя, состоящее из имени типа и префикса `type_`: `type_`*`имя_типа`*.

Поле `copy` имеет тип `Copy`:

```
typedef void (*Copy)(void*,void*);
```

По этому указателю из библиотечных функций `copy()` и `clone()` (см. подраздел "*Функции копирования значений по ссылкам*") вызывается функция копирования значений данного спецификационного типа. Функция копирования спецификационного типа вызывается, если ссылки, переданные в функцию `copy()` или `clone()`, ненулевые. В первом параметре ей передается ссылка, значение по которой должно быть скопировано в область памяти по ссылке, переданной ей во втором параметре.

Поле `compare` имеет тип `Compare`:

```
typedef int (*Compare)(void*,void*);
```

По этому указателю из библиотечных функций `compare()` и `equals()` (см. подраздел "*Функции сравнения значений по ссылкам*") вызывается функция сравнения значений данного спецификационного типа. Функция сравнения спецификационного типа вызывается, если ссылки, переданные в функцию `compare()` или `equals()`, ненулевые и типы ссылок либо одинаковые, либо являются *подтипами* одного типа, либо тип одной ссылки является *подтипом* другой (см. подраздел "*Инварианты типов*"). В качестве параметров ей передаются ссылки в том же порядке, в котором они были переданы в функцию `compare()` или `equals()`.

Поле `to_string` имеет тип `ToString`:

```
typedef String* (*ToString)(void*);
```

По этому указателю из библиотечной функции `toString()` (см. подраздел "*Функция построения строкового представления значения по ссылке*") вызывается функция построения строкового представления данного спецификационного типа. Функция построения строкового представления спецификационного типа вызывается, если ссылка, переданная в `toString()`, ненулевая.

Поле `enumerate` имеет тип `Enumerate`:

```
typedef void (*Enumerate) (void*,void(*callback)(void*,void*),void*);
```

По этому указателю вызывается функция перечисления ссылок спецификационных типов, содержащихся в значении данного спецификационного типа. Данная функция используется для разрешения циклов по спецификационным ссылкам при автоматическом управлении динамической памятью.

Поле `destroy` имеет тип `Destroy`:

```
typedef void (*Destroy)(void*);
```

По этому указателю автоматически вызывается функция освобождения ресурсов при обнулении счетчика ссылок на значение данного спецификационного типа.

Если в определении спецификационного типа базовый тип является допустимым типом языка SeC (см. подраздел "*Допустимые типы SeC*"), то инициализация любого поля может быть опущена. При этом для соответствующей базовой операции над данными спецификационного типа используется *функция по умолчанию*.

Функция инициализации по умолчанию

Функция инициализации по умолчанию для всех спецификационных типов, определенных на основе простых типов (не составных), имеет единственный дополнительный параметр базового типа. Она инициализирует значение спецификационного типа посредством глубокого копирования этого параметра, с учетом возможных циклов по указателям и спецификационным ссылкам. Функция инициализации структурных спецификационных типов имеет дополнительные параметры, типы и порядок которых совпадают с типами и порядком полей базовой структуры. Поля по переданной ссылке инициализируются посредством глубокого копирования переданных параметров. Функция инициализации спецификационных типов, определенных на базе массива фиксированной длины, имеет один дополнительный параметр, являющийся указателем на набор значений типа элементов массива в количестве, совпадающем с размерностью массива. Массив по переданной ссылке инициализируется посредством глубокого копирования каждого значения по полученному указателю в соответствующий ему элемент массива. Механизм копирования, используемый в функции инициализации, совпадает с механизмом копирования, используемым в функции копирования по умолчанию.

Функция копирования по умолчанию

Функция копирования по умолчанию обеспечивает глубокое копирование с учетом возможных циклов по указателям и спецификационным ссылкам, содержащихся по копируемой ссылке:

- значения допустимых простых типов языка C, за исключением типизированных указателей, копируются побайтно;
- спецификационные ссылки копируются с помощью функции копирования соответствующего спецификационного типа;
- типизированные указатели рассматриваются как указатели на одно единственное значение, не зависящее от месторасположения в памяти, поэтому копируется единственное значение по ненулевому указателю согласно правилам, перечисленным в данном списке;
- значения составных типов копируются посредством применения перечисленных правил к каждому составляющему элементу.

Функция сравнения по умолчанию

Функция сравнения по умолчанию сравнивает значения базового типа следующим образом:

- арифметические типы, функциональные указатели и нетипизированные указатели сравниваются побайтно;
- типизированные указатели рассматриваются как указатели на одно единственное значение, не зависящее от месторасположения в памяти, то есть нулевые указатели всегда считаются равными, нулевой указатель и ненулевой указатель всегда не равны, а ненулевые указатели равны тогда и только тогда, когда равны значения, на которые они указывают, причем значения по указателям сравниваются согласно правилам данного списка;
- спецификационные ссылки сравниваются с помощью библиотечной функции сравнения `compare()`;
- составные типы сравниваются посредством применения данных правил сравнения к каждому составляющему элементу.

Функция построения строкового представления по умолчанию

Функция построения строкового представления по умолчанию возвращает строковое представление значения базового типа:

- для арифметических типов — числовое представление;
- для нетипизированных и функциональных указателей — адрес;
- для типизированных указателей — либо `NULL`, либо строковое представление значения, лежащего по ненулевому указателю, помеченное его адресом;
- для спецификационных ссылок — результат вызова библиотечной функции преобразования в строку `toString()`;
- для структурных типов — конкатенация строковых представлений полей структуры, разделенных запятыми, обрамленная фигурными скобками и предваренная словом struct;
- для массива фиксированной длины — конкатенация строковых представлений элементов массива, разделенных запятыми, обрамленная квадратными скобками.

Функция перечисления внутренних спецификационных ссылок по умолчанию

Функция перечисления внутренних спецификационных ссылок по умолчанию ничего не делает, если базовый тип является простым неспецификационным типом. Если базовый тип является спецификационной ссылкой, вызывается функция по переданному указателю `callback`, которой в качестве параметров передаются спецификационная ссылка и вспомогательный параметр `par`, переданные в функцию перечисления спецификационных ссылок. Если базовый тип является составным, данные правила применяются для каждого составляющего элемента.

Функция освобождения ресурсов по умолчанию

Функция освобождения ресурсов по умолчанию

- ничего не делает, если базовый тип является арифметическим, функциональным или нетипизированным указателем;
- если базовый тип является спецификационной ссылкой, счетчик ссылок на значение по ней уменьшается на единицу;
- если базовый тип является типизированным указателем, то значение по ненулевому указателю обрабатывается согласно перечисляемым правилам, после чего вызывается функция `free()` для самого указателя;
- если базовый тип является составным, то перечисленные правила применяются для каждого составляющего элемента.

Если в определении спецификационного типа базовый тип является допустимым типом языка SeC (см. подраздел “*Допустимые типы SeC*”), и функции по умолчанию всех базовых операций реализуют необходимую функциональность, то в определении спецификационного типа используется пустой инициализатор.

```
specification typedef struct {int x; int y;} Point = {};

Point *pt2, *pt1 = create(&type_Point, 1, 2); /* pt1->x == 1,
                                                 pt1->y == 2*/
String* s1;
...
pt2 = clone(pt1); /* pt2->x == 1, pt2->y == 2*/
pt1->x = 10; /* pt1->x == 10, pt1->y == 2*/
s1 = toString(pt1); /* toCharArray_String(s1)=="struct { 10, 2 }" */
...
if(equals(pt1, pt2)) /* if(pt1->x == pt2->x && pt1->y == pt2->y) */
  ...
```

В приведенном выше примере спецификационный тип `Point` создается на основе структуры, содержащей два поля типа `int`. В определении типа используется пустой инициализатор. Поэтому для реализации базовых операций над данными этого типа используются функции по умолчанию. При создании ссылки типа `Point*` в функцию `create()` надо передавать значения для инициализации полей базовой структуры (см. подраздел “*Функция инициализации по умолчанию*”). После создания ссылки типа `Point*` с ней можно работать как с указателем на базовую структуру.

Для создания структур данных со сложной топологией, например при определении рекурсивных структур, рекомендуется использовать только спецификационные ссылки.

```
struct link;

specification typedef struct link Link;
struct link
{
  Link* next;
  int item;
};

specification typedef struct link Link = {};
...
Link*   l1 = create(&type_Link, NULL, 1)
      , l2 = create(&type_Link, NULL, 2);

l1->next = l2;
```

В представленном выше фрагменте кода определяется спецификационный тип `Link`, реализующий односвязный список. При присваивании полю `next` ссылки `l1` значения ссылки `l2` происходит автоматическое увеличение счетчика ссылок на значение по ссылке `l2`. Поэтому после уничтожении ссылки `l2` значение, на которое она ссылается, не уничтожается.

При использовании в определении типа `Link` рекурсивной структуры, содержащий неспецификационный указатель на саму себя, правильное управление динамической памятью более трудоемко.

```
specification typedef struct link {
  struct link* next;
  int item
  } Link = {};
...
struct link*   s = malloc(sizeof(struct link));
Link*   l = create(&type_Link, s, 1);
...
free(s);
```

В приведенном выше фрагменте кода после вызова `free()` для указателя `s`, поле `next` ссылки `l` будет указывать на освобожденную память. В этом случае, чтобы избежать таких проблем, необходимо определить специальные функции инициализации и освобождения ресурсов для типа `Link`.

Если функция по умолчанию некоторой базовой операции не реализует функциональность, необходимую для определяемого спецификационного типа, то для этой операции определяется специальная функция, указателем на которую инициализируется соответствующее поле в инициализаторе определения типа. Чаще всего такая необходимость возникает, когда базовый тип в определении спецификационного типа является указателем на первый элемент динамического массива, объединением, указателем на один из таких типов, структурой или массивом фиксированный длины, содержащих элементы перечисленных типов.

Функция инициализации спецификационного типа

`void` ***`имя_функции_инициализации`***`(void*` **`p, va_list* arg_list)`**

Функция не имеет возвращаемого значения. В первом аргументе функция получает указатель типа `void*` на *обнуленную* область памяти, выделенную для хранения данных спецификационного типа и инициализирует эту область значениями, переданными во втором аргументе в списке типа `va_list*`. При необходимости в функции выделяется дополнительная память для храненения данных.

```
struct integer_seq {
  int length;
  Integer* *items;
};

void init_IntegerSeq(void* ref, va_list *arg_list) {
  struct integer_seq *is = (struct integer_seq*)ref;

  is->length = va_arg(*arg_list, int);
  is->items  = calloc(is->length, sizeof(Integer*));
}

specification typedef struct integer_seq IntegerSeq = {
  .init = init_IntegerSeq,
  ...
};
```

В приведенном выше примере создается спецификационный тип `IntegerSeq`, предназначенный для хранения последовательности заранее не известной длины, содержащей ссылки типа

Integer* — ссылки на значения библиотечного спецификационного типа Integer, который является спецификационным представлением встроенного типа языка С int. Тип IntegerSeq определяется на основе структуры struct integer_seq с двумя полями: длина последовательности — length и указатель на массив, содержащий саму последовательность,— items.

Библиотечная функция create() выделяет память только для хранения значения самой структуры struct integer_seq. Функция инициализации по умолчанию (см. подраздел "*Функция инициализации по умолчанию*") для такого структурного спецификационного типа имеет два параметра инициализации: первый параметр типа int и второй типа Integer**. Причем второй параметр в функции инициализации по умолчанию интерпретируется как указатель на единственное значение. Поэтому в функции по умолчанию поле items инициализируется указателем на ссылку, которая содержит копию значения по ссылке, переданной через указатель. В рассматриваемом случае такая функциональность неприемлема. Поэтому необходимо использовать специальную функцию инициализации init_IntegerSeq, реализующую необходимую функциональность.

Функция init_IntegerSeq ожидает, что в списке типа va_list* передается единственный параметр инициализации типа IntegerSeq — длина последовательности. Его значением инициализируется поле length по инициализируемой ссылке. Второе поле items инициализируется указателем на динамически выделенную и инициализированную нулями область памяти, достаточную для хранения последовательности ссылок нужной длины.

Указателем на функцию инициализации init_IntegerSeq инициализируется поле init в определении типа IntegerSeq.

Функция освобождения ресурсов спецификационного типа

void ***имя_функции_освобождения_ресурсов***(void* **p)**

Функция без возвращаемого результата имеет один параметр типа void* — указатель на область памяти, хранящую данные спецификационного типа. Функция должна освободить *только дополнительную память*, выделенную в функции инициализации данного спецификационного типа.

```
void destroy_IntegerSeq (void *ref) {
  struct integer_seq* is = (struct integer_seq*)ref;
  int i;

  for(i = 0; i < is->length; i++)
    is->items[i] = NULL;
  free (is->items);
}
specification typedef struct integer_seq IntegerSeq = {
  .init    = init_IntegerSeq,
  ...
  .destroy = destroy_IntegerSeq
};
```

В приведенном выше примере определяется функция освобождения ресурсов destroy_IntegerSeq для спецификационного типа IntegerSeq. Определение этой функции необходимо, так как функция освобождения ресурсов по умолчанию (см. подраздел "*Функция освобождения ресурсов по умолчанию*") для составных типов интерпретирует поле items как указатель на единственное значение типа Integer*, поэтому счетчик ссылок уменьшается только для первой ссылки последовательности, после чего вызывается free() для указателя items.

Функция destroy_IntegerSeq уменьшает счетчик ссылок на единицу для каждого элемента последовательности по переданной ссылке, после чего освобождает память по указателю items. Счетчики ссылок уменьшаются присваиванием ссылкам NULL.

Указателем на функцию освобождения ресурсов `destroy_IntegerSeq` инициализируется поле `destroy` в определении типа `IntegerSeq`.

Функция копирования спецификационного типа

`void` ***`имя_функции_копирования`***`(void*` **`src`**`, void*` **`dst`**`)`

Функция без возвращаемого значения имеет два параметра типа `void*`. Функция должна копировать *на необходимую глубину* значения данных по переданному в первом параметре указателю `src` в *обнуленную* область памяти по переданному во втором параметре указателю `dst`.

```
void copy_IntegerSeq (void *src, void *dst) {
  struct integer_seq   *is_src = (struct integer_seq *)src
                     , *is_dst = (struct integer_seq *)dst;
  int i;

  is_dst->length = is_src->length;
  is_dst->items  = calloc(is_src->length, sizeof(Integer*));
  for(i = 0; i < is_src->length; i++)
    is_dst->items[i] = clone(is_src->items[i]);
}

specification typedef struct integer_seq IntegerSeq = {
  .init    = init_IntegerSeq,
  .copy    = copy_IntegerSeq,
  ...
  .destroy = destroy_IntegerSeq
};
```

В приведенном выше примере определяется функция копирования значений типа `IntegerSeq`. Определение этой функции необходимо, так как функция копирования по умолчанию (см. подраздел “*Функция копирования по умолчанию*”) интерпретирует поле `items` как указатель на единственное значение типа `Integer*`, то есть копируется только первое значение по ссылке первого элемента последовательности `src`. Аункция `copy_IntegerSeq` обеспечивает глубокое копирование всей последовательности, при помощи библиотечной функции копирования `clone()`(см. подраздел “*Функции копирования значений по ссылкам*”). Инициализация нулями памяти, выделяемой для `is_dst->items` необходима для того, чтобы при присваивании в цикле `is_dst->items[i] = clone(is_src->items[i])` не происходило попытки уменьшения счетчика ссылок на несуществующее значение по ссылке `is_dst->items[i]`.

Указателем на функцию копирования инициализируется поле `copy` в определении типа `IntegerSeq`.

Функция сравнения спецификационного типа

`int` ***`имя_функции_сравнения`***`(void*` **`left`**`, void*` **`right`**`)`

Функция имеет возвращаемое значение типа `int` и два параметра типа `void*`. Функция сравнивает значения по переданным указателям и возвращает ноль, если значения эквивалентны, и отличное от нуля значение в обратном случае. Ненулевое значение может зависеть от отношения значений по переданным ссылкам.

```
int compare_IntegerSeq(void* left, void* right) {
  struct integer_seq   *isl = (struct integer_seq *)left
                     , *isr = (struct integer_seq *)right;
  if (isl->length != isr->length) return isl->length - isr->length;
  else {
    int i, res;
    for(i = 0; i < isl->length; i++) {
      res = compare(isl->items[i], isr->items[i]);
      if(res) return res;
    }
  }
  return 0;
}

specification typedef struct integer_seq IntegerSeq = {
  .init    = init_IntegerSeq,
  .copy    = copy_IntegerSeq,
  .compare = compare_IntegerSeq,
  ...
  .destroy = destroy_IntegerSeq
};
```

В приведенном выше примере определяется функция сравнения значений типа `IntegerSeq`. Определение этой функции необходимо, так как функция сравнения по умолчанию (см. подраздел “*Функция сравнения по умолчанию*”) интерпретирует поле `items` как указатель на единственное значение типа `Integer*`, то есть сравниваются только значения по ссылкам первых элементов последовательностей `left` и `right`.

Функция сравнения `compare_IntegerSeq` обеспечивает сравнение последовательностей поэлементно. Если последовательности имеют разную длину, то возвращается разность длин последовательностей по переданным ссылкам `left` и `right`. Если последовательности одинаковой длины, то они сравниваются поэлементно при помощи библиотечной функции `compare()`. В этом случае, если все элементы последовательностей совпадают, результатом является ноль, в противном случае возвращается результат сравнения первых несовпадающих элементов.

Указателем на функцию сравнения инициализируется поле `compare` в определении типа `IntegerSeq`.

```
struct one_dim_simpl {double x1; double y1; double x2; double y2;};
int compare_OneDimSimplex(void* left, void* right) {
  struct one_dim_simpl   *lv = (struct one_dim_simpl*)left
                       , *rv = (struct one_dim_simpl*)right;
  double   lx = lv->x2 - lv->x1
         , ly = lv->y2 - lv->y1
         , rx = rv->x2 - rv->x1
         , ry = rv->y2 - rv->y1;

  double res = sqrt(lx * lx + ly * ly) - sqrt(rx * rx + ry * ry);
  return res > 0.0 ? 1 : (res < 0.0 ? -1 : 0);
}

specification typedef struct one_dim_simpl OneDimSimplex = {
  .compare = compare_OneDimSimplex;
}
```

В приведенном выше примере создается спецификационный тип `OneDimSimplex`, представляющий отрезок на плоскости. Отрезок задается координатами двух точек на плоскости: две координаты первой точки `x1` и `y1` и две координаты второй точки `x2` и `y2`. Отрезки равны, если совпадают их длины. Функция сравнения по умолчанию реализует поэлементное сравнение составных спецификационных типов. В данном случае функция сравнения по умолча-

нию будет возвращать ноль, если значение каждого поля по первой ссылке совпадает со значением соответствующего поля по второй ссылке. Поэтому для типа `OneDimSimplex` необходимо реализовать специальную функцию сравнения `compare_OneDimSimplex`, и инициализировать поле `compare` в инициализаторе определения типа указателем на эту функцию. Функция `compare_OneDimSimplex` вычисляет длины отрезков по переданным ссылкам и возвращает `0`, если они равны, `1`, если первый отрезок длиннее второго, и `-1`, если первый отрезок короче второго.

Функция построения строкового представления спецификационного типа

String* ***имя_функции_построения_строкового_представления*****(**`void*` **p)**

Функция имеет возвращаемое значение типа `String*` и параметр типа `void*`. Функция возвращает ссылку на спецификационный тип `String` (см. раздел “*Библиотека спецификационных типов*”), по которой должно содержаться строковое представление значения спецификационного типа, соответствующее значению по ссылке, переданной в единственном параметре функции.

```
String* to_string_IntegerSeq(void *ref) {
  struct integer_seq *is = (struct integer_seq *)ref;

  String *start = create_String ("<");
  String *end   = create_String (">");
  String *sep   = create_String (", ");

  String *res = start;

  if (is->length > 0) {
    int i;
    for (i = 0; i < is->length; i++) {
      if (i > 0) res = concat_String(res, sep);
      res = concat_String(res, toString(is->items[i]));
    }
  }
  return concat_String (res, end);
}

specification typedef struct integer_seq IntegerSeq = {
  .init      = init_IntegerSeq,
  .copy      = copy_IntegerSeq,
  .compare   = compare_IntegerSeq,
  .to_string = to_string_IntSeq,
  ...
  .destroy   = destroy_IntegerSeq
}
```

В приведенном выше примере определяется функция построения строкового представления `to_string_IntegerSeq` для типа. Определение этой функции необходимо, так как функция построения строкового представления по умолчанию (см. подраздел “*Функция построения строкового представления по умолчанию*”) интерпретирует поле `items` как указатель на единственное значение типа `Integer*` и создает строку, содержащую в фигурных скобках через запятую значение поля `length` и строковое представление значения по ссылке первого элемента последовательности.

Функция `to_string_IntSeq` возвращает ссылку типа `String*`, содержащую строку, в которой в угловых скобках ('<' и '>') перечисляются через запятую строковые представления значений по ссылкам всех элементов последовательности с сохранением их порядка. В функции `to_string_IntSeq` используются функции `create_String()` и `concat_String()`, подробное описание которых дано в разделе “*Библиотека спецификационных типов*”.

Указателем на `to_string_IntSeq` функцию инициализируется поле `to_string` в определении типа `IntegerSeq`.

Функция перечисления внутренних спецификационных ссылок спецификационного типа

```
void имя_функции_перечисления_спецификационных_ссылок(*Enumerate)
  (void* p, void (*callback)(void*,void*),
   void* par
  )
```

Функция перечисления ссылок должна для каждой ссылки спецификационного типа, доступной через переданную в первом аргументе ссылку, вызвать функцию, указатель на которую передается ей во втором аргументе. Во всех вызовах этой функции перечисляемые ссылки передаются в первом аргументе, во втором аргументе передаются параметры через указатель `par`, переданный в третьем аргументе функции перечисления.

```
void enumerate_IntegerSeq(  void* p
                          , void (*callback)(void*,void*)
                          , void* par
                         ) {
  struct integer_seq *is = (struct integer_seq*)p;
  int i;

  for(i = 0; i < is->length; i++)
    callback(is->items[i], par);
}

specification typedef struct integer_seq IntegerSeq = {
  .init      = init_IntegerSeq,
  .copy      = copy_IntegerSeq,
  .compare   = compare_IntegerSeq,
  .enumerate = enumerate_IntegerSeq,
  .destroy   = destroy_IntegerSeq
};
```

В приведенном выше примере определяется функция перечисления ссылок `enumerate_IntegerSeq()` для типа `IntegerSeq`. Определение специальной функции перечисления ссылок необходимо, так как функция перечисления ссылок по умолчанию (см. подраздел “*Функция перечисления внутренних спецификационных ссылок по умолчанию*”) не обеспечивает перечисление ссылок, доступных через указатель на массив элементов.

В функции `enumerate_IntegerSeq()` для всех ссылок последовательности, доступных через поле `items` типа `Integer**`, вызывается функция через указатель, переданный в функцию перечисления `enumerate_IntegerSeq()` во втором параметре. В первом параметре при этих вызовах передаются перечисляемые спецификационные ссылки, во втором параметре передается указатель, переданный в третьем параметре в функцию перечисления `enumerate_IntegerSeq()`.

Инварианты

В *спецификации* формулируются требования к тестируемой системе, в том числе требования к данным. Такие требования заключаются в ограничении диапазона допустимых значений и могут накладываться как на некоторый тип данных в целом (с помощью *инвариантов типов*), так и на значения отдельных глобальных переменных (с помощью *инвариантов переменных*).

Инварианты могут рассматриваться и как общая часть *предусловий* и *постусловий спецификационных функций*, которые используют данные соответствующих типов.

Инварианты автоматически проверяются в *спецификационных функциях* перед проверкой *предусловия*:

- для параметров функции
- для выражений, описанных в ограничения доступа **reads** и **updates**

и перед проверкой *постусловия*:

- для параметров функции
- для выражений, описанных в *ограничениях доступа* **writes** и **updates**
- для возвращаемого значения

Также предусмотрен способ явной проверки инварианта.

При проверке инвариантов составных типов автоматически происходит проверка инвариантов и для всех его составляющих элементов.

Инвариант типа

Инвариант типа вводит ограничения на диапазон значений некоторого типа. Получающийся новый тип, множество значений которого является подмножеством значений *базового типа*, называется *подтипом*. На базовый тип наложено ограничение: он должен являться *допустимым типом*.

Тип с инвариантом определяется с помощью конструкции `typedef`, помеченной ключевым словом **invariant**:

```
invariant typedef int Nat;
```

В данном случае `int` является *базовым типом*, а `Nat` — определяемым *подтипом*. При этом в отличие от обычной конструкции `typedef` языка C, которая лишь вводит новое имя для старого типа, в данном случае определяется новый тип с собственным множеством значений.

Ограничения на множество значений подтипа определяются в конструкции **invariant**, схожей с определением функции с одним параметром определяемого *подтипа*. Функция возвращает логическое значение: `true`, если переданное значение удовлетворяет ограничениям (инвариант выполнен), или `false`, если не удовлетворяет (инвариант нарушен). Поскольку тип возвращаемого значения фиксирован, он не указывается явно:

```
invariant(Nat n) {
  return n > 0;
}
```

Если в качестве базового типа используется *спецификационный тип*, то параметр функции-инварианта будет иметь тип соответствующей *спецификационной ссылки*, поскольку значения *спецификационных типов* доступны только через указатель:

```
invariant typedef Integer Natural;
invariant(Natural* n) {
  return value_Integer(n) > 0;
}
```

При этом функции работы с *подтипом* будут взяты из определения *базового типа*.

Функция-инвариант не должна иметь побочных эффектов: не должны изменяться видимые снаружи данные; динамическая память, выделенная внутри функции, должна освобождаться в ней же.

Допускаются определения новых спецификационных типов с указанием инварианта:

```
invariant specification typedef int Natural = {};
invariant(Natural* n) {
  return *n > 0;
}
```

В этом случае *подтип* и *базовый тип* будут совпадать.

Отметим, что нельзя определять *спецификационный тип* на основе другого *спецификационного типа*, однако можно создавать *подтипы спецификационных типов*. Более того, можно определять *подтипы подтипов*, создавая таким образом иерархию типов. В таком случае для значения *подтипа* будут проверяться и инварианты всех "родительских" *подтипов* на пути к вершине иерархии.

Инвариант переменной соответствующего типа можно явно проверить с помощью функции **`invariant`**:

```
invariant typedef int Nat;
Nat n;
...
if ( invariant(n) ) ...
```

Подтип можно приводить к базовому типу, более того, такое преобразование осуществляется неявно. Базовый тип также можно привести к подтипу. Однако, поскольку значения подтипа являются подмножеством значений базового типа, такое преобразование нужно записывать явно:

```
invariant typedef int Nat;
int i;
Nat n;
...
i = n;
n = (Nat)i;
```

При этом возможна ситуация, когда инвариант типа окажется нарушенным. При необходимости его можно проверить явно после присваивания.

Инвариант переменной

Инвариант переменной позволяет ввести ограничения на диапазон значений отдельной глобальной переменной, имеющей *допустимый тип*.

Переменная с инвариантом определяется с помощью обычной декларации или определения, с ключевым словом **`invariant`**:

```
invariant int Qty;
```

Ограничения на множество значений переменной определяются в конструкции **`invariant`**, схожей с определением функции без параметров. Функция возвращает логическое значение: `true`, если значение глобальной переменной удовлетворяет ограничениям (инвариант выполнен), или `false`, если не удовлетворяет (инвариант нарушен). Поскольку тип возвращаемого значения фиксирован, он не указывается явно. В скобках указывается имя переменной, для которой определяется инвариант:

```
invariant(Qty) {
  return Qty >= 0;
}
```

Однако переменная не является параметром функции-инварианта. Функция осуществляет доступ непосредственно к значению глобальной переменной. При этом функция не должна иметь побочных эффектов: не должны изменяться видимые снаружи данные; динамическая память, выделенная внутри функции, должна освобождаться в ней же.

Инвариант переменной можно явно проверить с помощью функции **`invariant`**:

```
invariant int Qty;
...
if ( invariant(Qty) ) ...
```

Если переменная с инвариантом имеет тип, для которого определен *инвариант типа*, то сначала будет проверен *инвариант типа*, а затем инвариант переменной.

Спецификации

Спецификация представляет собой формальное описание требований к тестируемой системе. В тестируемой системе выделяются интерфейсные функции и данные, определяющие состояние этой системы. В спецификации поведение интерфейсных функций описывается *спецификационными функциями*, а состояние системы моделируется глобальными *переменными состояния*. Требования к тестируемой системе формулируются как ограничения на поведение интерфейсных функций (в виде *предусловий* и *постусловий* в *спецификационных функциях*) и на значения данных (в виде *инвариантов типов и переменных* состояния).

Привязка *спецификационных функций* и модельных данных к функциям и данным реализации тестируемой системы осуществляется при помощи *медиаторов*.

Кроме того, из спецификации извлекаются *критерии покрытия* (описанные в *спецификационных функциях*), позволяющие оценить полноту тестирования.

В случае тестирования систем с отложенными реакциями, их поведение при ответе на внешние воздействия состоит из непосредственных и отложенных реакций. Первые описываются обычными *спецификационными функциями*, а для описания последних в спецификацию добавляются *отложенные реакции*. Привязка *отложенных реакций* к тестируемой системе осуществляется с помощью *медиаторов* и *сборщиков реакций*. В отличие от *спецификационных функций*, для *отложенных реакций* не задаются *критерии покрытия*.

Спецификационные функции

Спецификационные функции служат для описания поведения интерфейсных функций тестируемой системы. В общем случае спецификационная функция определяет поведение тестируемой системы при определенном воздействии на нее через некоторую часть интерфейса.

Спецификационные функции описывают поведение в форме *ограничений доступа* к данным, *предусловий*, *критериев покрытия* и *постусловий*.

Декларация спецификационной функции состоит из ключевого слова **`specification`**, сигнатуры функции (в обычном смысле языка C) и, возможно, *ограничений доступа*.

```
specification double sqrt_spec(double x);
```

Тело спецификационной функции состоит из трех частей: *предусловия* (может быть опущено), *критериев покрытия* (любое количество, может не быть ни одного) и *постусловия* (обязательно одно).

Предусловие проверяет применимость функции к данному набору значений параметров и переменных состояния. *Критерии покрытия* разбивают поведение системы на *ветви функциональности*. И *предусловие*, и *критерии покрытия* выполняются в пре-состоянии, т. е. до

взаимодействия с тестируемой системой. Значения выражений в этот момент называются *пре-значениями*.

Перед вычислением *постусловия* осуществляется взаимодействие с тестируемой системой с помощью вызова *медиатора*. *Постусловие* проверяет соответствие полученного результата ожидаемому. Оно выполняется в пост-состоянии, т. е. после взаимодействия, и имеет дело с *пост-значениями* выражений.

```
specification double sqrt_spec(double x) {
  pre { ... }
  coverage C { ... }
  post { ... }
}
```

Ограничения доступа определяют способ использования параметров и глобальных переменных в спецификационной функции до и после взаимодействия с тестируемой системой.

В общем случае в теле спецификационной функции между описанными блоками может использоваться дополнительный код. Если внутри фигурных скобок, набранных полужирным шрифтом, нет дополнительного кода, они могут опускаться:

```
specification сигнатура ограничения_доступа {
  дополнительный_код_1_1
  pre { ... }
  {
    дополнительный_код_2_1
    coverage имя_1 { ... }
    ...
    coverage имя_n { ... }
    {
      дополнительный_код_3_1
      post { ... }
      дополнительный_код_3_2
    }
    дополнительный_код_2_2
  }
  дополнительный_код_1_2
}
```

Дополнительный код не должен иметь побочных эффектов: не должны изменяться видимые снаружи данные; динамическая память, выделенная внутри блока кода, должна освобождаться либо в нем же, либо в парном ему блоке.

Спецификационные функции вызываются, как правило, в *сценарных функциях*. Вызов спецификационной функции состоит в проверке *инвариантов* в соответствии с *ограничениями доступа*, проверке *предусловия*, вычислении покрытых ветвей в соответствии с *критериями покрытия*, осуществлении тестового воздействия и синхронизации модельного и реализационного состояний в *медиаторе*, вторичной проверке *инвариантов* и проверке *постусловия*. Спецификационная функция возвращает значение, вычисленное в *медиаторе* (если она не объявлена как `void`).

Отложенные реакции

Отложенные реакции служат для описания поведения тестируемой системы при отложенном реагировании на внешние воздействия. Отложенные реакции описывают поведение в форме *ограничений доступа* к данным, *предусловий* и *постусловий*. В отличие от *спецификационных функций*, в отложенных реакциях не используются *критерии покрытия*.

Декларация отложенной реакции состоит из ключевого слова **`reaction`**, сигнатуры функции (в обычном смысле языка C) и, возможно, *ограничений доступа*.

```
reaction String* recv_spec(void);
```

Отложенная реакция:

- никогда не имеет параметров,
- должна возвращать спецификационную ссылку.

Тело отложенной реакции состоит из двух частей: *предусловия* (может быть опущено) и *постусловия* (обязательно одно).

Предусловие проверяет возможность возникновения реакции в данном состоянии. *Предусловие* выполняется в пре-состоянии и имеет доступ только к *пре-значениям* выражений.

Постусловие проверяет соответствие результата, полученного при возникновении реакции, ожидаемому. Оно выполняется в пост-состоянии после возникновения реакции и имеет дело с *пост-значениями* выражений.

```
reaction String* recv_spec(void) {
  pre { ... }
  post { ... }
}
```

Ограничения доступа определяют способ использования глобальных переменных до и после возникновения реакции.

В общем случае в теле отложенной реакции между описанными блоками может использоваться дополнительный код. Если внутри фигурных скобок, набранных полужирным шрифтом, нет дополнительного кода, они могут опускаться:

```
reaction сигнатура ограничения_доступа {
  дополнительный_код_1_1
  pre { ... }
  {
    дополнительный_код_2_1
    post { ... }
    дополнительный_код_2_2
  }
  дополнительный_код_1_2
}
```

Дополнительный код не должен иметь побочных эффектов: не должны изменяться видимые снаружи данные; динамическая память, выделенная внутри блока кода, должна освобождаться либо в нем же, либо в парном ему блоке.

Отложенная реакция никогда не вызывается явно, поскольку инициируется тестируемой системой.

Ограничения доступа

Ограничения доступа указывают способ использования в спецификационных функциях и отложенных реакциях глобальных переменных и параметров, а так же выражений с ними. Поддерживаются три вида ограничений доступа: чтение (**`reads`**), запись (**`writes`**) и изменение (**`updates`**).

Ограничения записываются после сигнатуры спецификационной функции или отложенной реакции в виде списка выражений через запятую, помеченного одним из ключевых слов **`reads`**, **`writes`** или **`updates`**:

```
specification void root_spec(
  double a, double b, double c,
  double *x1, double *x2)
  reads a, b, c
  writes *x1, *x2;
```

Ограничение доступа **reads** для некоторого выражения означает, что значение этого выражения не изменяется в результате взаимодействия, т. е. *пре-значение* этого выражения совпадает с *пост-значением*.

Для таких выражений автоматически проверяются инварианты перед проверкой *предусловия*, а перед проверкой *постусловия* проверяется, что значение выражения не изменилось.

Ограничение доступа **writes** для некоторого выражения означает, что *пре-значение* не используется в спецификационной функции и может быть не определено, а *пост-значение* вырабатывается в результате взаимодействия с тестируемой системой. Выражения с доступом **writes** нельзя использовать в операторе взятия пре-значения `@` (см. раздел “Постусловие”).

Для таких выражений автоматически проверяются инварианты перед проверкой *постусловия*.

Ограничение доступа **updates** для некоторого выражения означает, что пре-значение этого выражения является входным параметром, от которого может зависеть поведение системы, а *пост-значение* получается в результате взаимодействия и может не совпадать с *пре-значением*.

Для таких выражений автоматически проверяются инварианты перед проверкой как *предусловия*, так и *постусловия*.

Выражениям в ограничениях доступа можно присваивать идентификатор, называемый *псевдонимом*. В дальнейшем *псевдоним* можно использовать для доступа к значению выражения, в том числе к *псевдониму* можно применять оператор `@`:

```
specification void deposit_spec(AccountModel *acct, int sum)
  reads   sum
  updates balance = acct->balance
{
  ...
  post {
    return balance == @balance + sum;
  }
}
```

Предусловие

При взаимодействии, описываемом спецификационной функцией, поведение тестируемой системы в некоторых ситуациях может быть не определено. Для выделения таких ситуаций и служит предусловие. Во время тестирования предусловие проверяется каждый раз при обращении к спецификационной функции. Нарушение предусловия означает, что тест составлен некорректно.

В случае отложенной реакции, предусловие определяет, возможно ли возникновение такой реакции в данном состоянии. Во время тестирования предусловие проверяется каждый раз при возникновении реакции. При нарушении предусловия фиксируется несоответствие между поведением целевой системы и ее спецификацией.

Предусловие записывается в виде инструкций, заключенных в фигурные скобки и помеченных ключевым словом **pre**. Эти инструкции представляют собой тело функции, имеющей те

же параметры, что и спецификационная функция или отложенная реакция, и возвращающей результат типа `bool`, показывающий, выполнено ли предусловие.

```
specification double sqrt_spec(double x) {
  pre {
    return x >= 0.0;
  }
  ...
}
```

Если поведение системы определено при любых значениях входных параметров и в любом модельном состоянии (или появление реакции допустимо в любом состоянии), предусловие можно опустить.

Предусловие вычисляется до взаимодействия с тестируемой системой. Значения выражений в этот момент называются *пре-значениями*.

Перед проверкой предусловия автоматически проверяются *инварианты* параметров спецификационной функции и выражений, описанных в *ограничениях доступа* как **reads** и **updates**.

Предусловие не должно иметь побочных эффектов: не должны изменяться видимые снаружи данные; динамическая память, выделенная внутри предусловия, должна освобождаться в нем же.

Предусловие спецификационной функции может быть явно вычислено с помощью конструкции **pre** (используется, как правило, в *сценарных функциях* для того, чтобы не вызывать спецификационную функцию с некорректными параметрами):

```
if (pre sqrt_spec(-1.0)) ...
```

Критерий покрытия

Каждый критерий покрытия разбивает поведение тестируемой системы на *ветви функциональности*. Критерии покрытия используются для оценки полноты тестирования: при тестировании ставится задача хотя бы один раз пройти по каждой ветви функциональности.

Критерий покрытия записывается в виде инструкций, заключенных в фигурные скобки и помеченных ключевым словом **coverage** и идентификатором. Эти инструкции представляют собой тело функции, имеющей те же параметры, что и спецификационная функция, и возвращающей специальную конструкцию — заключенные в фигурные скобки идентификатор ветви и строковый литерал с ее кратким описанием.

```
specification void root_spec(
  double a, double b, double c,
  double *x1, double *x2)
{
  double d = b*b - 4*a*c;
  ...
  coverage C {
    if (d < 0.0)
      return { N, "Negative discriminant" };
    else if (d == 0.0)
      return { Z, "Zero discriminant" };
    else
      return { P, "Positive discriminant" };
  }
  ...
}
```

Для любого набора пре-значений параметров и глобальных переменных, удовлетворяющих предусловию, критерий покрытия должен возвращать конструкцию, идентифицирующую ветвь функциональности. В противном случае при выполнении теста будет зафиксирована ошибка.

Если в спецификационной функции не определен ни один критерий покрытия, считается, что имеет место один псевдокритерий с одной псевдоветвью.

Так же, как и *предусловие*, критерий покрытия вычисляется до взаимодействия с тестируемой системой и имеет доступ к *пре-значениям* выражений. Критерий покрытия не должен иметь побочных эффектов, т. е. не должен изменять видимые снаружи данные и должен освобождать динамическую память, выделенную внутри него.

Один из критериев покрытия может быть объявлен критерием по умолчанию, для чего используется ключевое слово **`default`**:

```
specification int f_spec(int a) {
  ...
  default coverage C1 { ... }
  coverage C2 { ... }
  ...
}
```

Если ни один критерий покрытия спецификационной функции не объявлен критерием по умолчанию, им становится текстуально последний критерий. Критерий по умолчанию можно установить и во время выполнения теста с помощью функции `set_coverage_`*`имя_спецификационной_функции`*`()`. Эта функция принимает строковое представление идентификатора критерия и возвращает `true`, если функция отработала успешно, или `false`, если критерий с таким именем не существует:

```
set_coverage_f_spec("C2");
```

Количество ветвей функциональности в критерии покрытия по умолчанию может быть вычислено с помощью функции `get_coverage_size_`*`имя_спецификационной_функции`*`()`, не имеющей параметров:

```
int num_of_branches = get_coverage_size_f_spec();
```

Номер достигнутой на заданных параметрах ветви функциональности в критерии покрытия по умолчанию может быть вычислен с помощью конструкции **`coverage`**:

```
int branch_num = coverage f_spec(10);
```

Пример использования этих функций см. в разделе “Итерация по критерию покрытия”.

При необходимости повторных вычислений выражений, задающих разбиение на ветви функциональности, в постусловии или в критериях покрытия, определяемых после данного критерия покрытия, может использоваться конструкция **`coverage`**`(`*`идентификатор_ветви`*`)`, значением которой является идентификатор покрываемой ветви функциональности указанного критерия покрытия. Эта конструкция может использоваться в условных операторах if-else и в операторах switch языка C:

```
specification int f_spec(int a) {
  ...
  default coverage C1 {
    if (...)
      return { B1, "branch 1" };
    else
      return { B2, "branch 2" };
  }
  coverage C2 {
    if (coverage(C1) == B1) {
      ...
    } else {
      ...
    }
  }
  ...
}
```

Постусловие

Постусловие спецификационной функции служит для описания ограничений, которым должны удовлетворять результаты работы тестируемой системы при взаимодействии, описываемом спецификационной функцией. Во время тестирования постусловие проверяется каждый раз после осуществления взаимодействия. Если постусловие оказывается нарушенным, фиксируется несоответствие поведения целевой системы ее спецификации.

Постусловие отложенной реакции описывает ограничения, которым должны удовлетворять результаты взаимодействия после возникновения реакции. Если постусловие оказывается нарушенным, фиксируется несоответствие поведения целевой системы ее спецификации.

Постусловие записывается в виде инструкций, заключенных в фигурные скобки и помеченных ключевым словом **post**. Эти инструкции представляют собой тело функции, имеющей те же параметры, что и спецификационная функция, и возвращающей результат типа `bool`. Значение `true` показывает, что поведение тестируемой системы соответствует ожидаемому (постусловие выполнено), а значение `false` показывает, что поведение отличается от ожидаемого.

В спецификационной функции и отложенной реакции всегда должно быть ровно одно постусловие.

Для доступа к значению, возвращенному *медиатором* спецификационной функции, используется идентификатор спецификационной функции (если спецификационная функция не объявлена как `void`). Аналогично, для доступа к значению реакции, зафиксированному при регистрации, используется идентификатор отложенной реакции.

```
specification double sqrt_spec(double x) {
  ...
  post {
    if (x == 0.0)
      return (sqrt_spec == 0.0);
    return ( sqrt_spec >= 0.0 &&
             fabs( (sqrt_spec*sqrt_spec - x) / x ) < EPS );
  }
}
```

Постусловие вычисляется после взаимодействия с тестируемой системой. Фактически, ключевое слово **post** трактуется как тестовое воздействие. Значения выражений после воздействия называются *пост-значениями*.

Перед проверкой постусловия автоматически проверяются *инварианты* параметров спецификационной функции и выражений, описанных в *ограничениях доступа* как **writes** и **updates**, а также *инвариант* возвращаемого значения.

Для доступа из постусловия к *пре-значениям* используется унарный оператор `@`. Выражение под этим оператором должно иметь *допустимый тип* и должно быть вычислимо непосредственно перед ключевым словом **post** (поскольку значения таких выражений автоматически сохраняются непосредственно перед осуществлением взаимодействия с тестируемой системой). Запрещено использовать оператор `@` для выражений, имеющих доступ **writes**.

```
specification void f(List* l/*List — библиотечный спецификационный тип*/) {
  int j;
  post {
    int i;
    Object* pre_item;
    for(i = 0, j = 0; i < @size_List(l)/* допустимо */; i++, j++) {
      pre_item = @get_List(l, i); /* недопустимо: i не определена
                                     вне постусловия
                                   */
      pre_item = @get_List(l, j); /* недопустимо: j имеет единственное
                                     неизвестное значение вне постусловия
                                   */
      pre_item = get_List(@l, j); /* допустимо*/
      if(equals(get_List(l, i), pre_item))
        return false;
    }
    return true;
  }
}
```

Если требуется обеспечить доступ к *пре-значению* выражения типа, который не является *допустимым*, следует вручную сохранить значение этого выражения в локальной переменной до блока **post** (возможно, лучшее решение состоит в использовании подходящего или определении нового спецификационного типа):

```
{
  char *s = "...", *pre_s;
  ...
  pre_s = strdup(s);
  post {
    return !strcmp(s, pre_s);
  }
  free(pre_s);
}
```

Постусловие не должно иметь побочных эффектов: в нем не должны изменяться видимые снаружи данные; динамическая память, выделенная внутри постусловия, должна освобождаться там же.

Медиаторы

Медиаторы предназначены для связывания *спецификации* с реализацией тестируемой системы.

Медиаторы выполняют три задачи. Во-первых, они осуществляют тестовое воздействие, преобразуя при этом параметры воздействия из модельного представления в реализацион-

ное. Во-вторых, получают результат и преобразуют его из реализационного представления в модельное. В-третьих, медиаторы синхронизируют модельное состояние с состоянием реализации тестируемой системы.

Медиаторы реализуются как *медиаторные функции* и *сборщики реакций*.

В следующей таблице показано, какие задачи выполняются какой частью медиатора:

Тестирование	**Осуществление воздействия**	**Получение результата**	**Синхронизация состояния**
без отложенных реакций	блок воздействия медиаторной функции	блок воздействия медиаторной функции	блок синхронизации медиаторной функции
с отложенными реакциями	—	сборщик реакций	блок синхронизации медиаторной функции

Медиаторная функция

Медиаторная функция реализует медиатор, связывая *спецификационную функцию* или *отложенную реакцию* с реализацией.

Декларация медиаторной функции состоит из ключевых слов **`mediator for`**, между которыми указывается идентификатор медиаторной функции, и из полной сигнатуры соответствующей спецификационной функции или отложенной реакции, включая ограничения доступа:

```
specification bool push_spec(Integer* i) reads i updates stack_model;

mediator push_media for
  specification bool push_spec(Integer* i) reads i updates stack_model;
```

Тело медиаторной функции состоит из двух частей: *блока воздействия* и *блока синхронизации*.

В *блоке воздействия* осуществляется тестовое воздействие с преобразованием данных из модельного представления в реализационное и обратно. Блок воздействия обязателен для медиаторов *спецификационных функций* и отсутствует в медиаторах *отложенных реакций* (в случае отложенных реакций воздействие инициируется тестируемой системой).

Блок синхронизации приводит модельное состояние в соответствие с состоянием реализации тестируемой системы. Блок синхронизации может отсутствовать в медиаторах *спецификационных функций* и обязателен для медиаторов *отложенных реакций*.

```
mediator push_media for
  specification bool push_spec(Integer* i) reads i updates stack_model
{
  call { ... }
  state { ... }
}
```

В общем случае в теле медиаторной функции перед описанными блоками и после них может использоваться дополнительный код:

```
mediator идентификатор for сигнатура_спецификационной_функции {
  дополнительный_код_1
  call { ... }
  state { ... }
  дополнительный_код_2
}
```

Дополнительный код не должен иметь побочных эффектов: не должны изменяться видимые снаружи данные; динамическая память, выделенная внутри блока кода, должна освобождаться либо в нем же, либо в парном ему блоке.

Медиаторная функция связывается со спецификационной функцией (или отложенной реакцией) с помощью функции `set_mediator_`*`имя_спецификационной_функции`* (или `set_mediator_`*`имя_отложенной_реакции`*), принимающей указатель на медиаторную функцию. Обычно связывание осуществляется в *функции инициализации* сценария:

```
set_mediator_push_spec(push_media);
```

Если в ходе работы медиаторной функции выявится ошибка (например, невозможно будет преобразовать данные из модельного представления в реализационное или наоборот), следует зафиксировать ее с помощью функции setBadVerdict, принимающей строку с описанием возникшей ситуации:

```
setBadVerdict("Error description");
```

Блок воздействия

Блок воздействия используется только в медиаторах *спецификационных функций*. В нем выполняются следующие действия:

- параметры спецификационной функции преобразуются в реализационное представление
- вызывается интерфейсная функция (или несколько функций) тестируемой системы
- результат вызова интерфейсной функции и выходные параметры преобразуются из реализационного представления в модельное
- модельное представление результата возвращается из блока воздействия

Блок воздействия записывается в виде инструкций, заключенных в фигурные скобки и помеченных ключевым словом **`call`**. Эти инструкции представляют собой тело функции, имеющей те же параметры, что и соответствующая *спецификационная функция*, и возвращающей результат того же типа, что и *спецификационная функция*.

```
stack *stack_impl;
List*  stack_model;

int push(stack*, int);
specification bool push_spec(Integer* i) reads i updates stack_model;

mediator push_media for
  specification bool push_spec(Integer* i) reads i updates stack_model
{
  call {
    return (bool)push(stack_impl, value_Integer(i));
  }
  ...
}
```

Блок воздействия исполняется тестовой системой при вызове соответствующей *спецификационной функции*, в точке перед ключевым словом **`post`**.

Блок синхронизации

Блок синхронизации приводит модельное состояние в соответствие с состоянием реализации тестируемой системы после осуществления воздействия или возникновения отложенной реакции.

Блок синхронизации записывается в виде инструкций, заключенных в фигурные скобки и помеченных ключевым словом **state**. Эти инструкции представляют собой тело функции без возвращаемого результата, имеющей те же параметры, что и соответствующая *спецификационная функция* или *отложенная реакция*.

Если соответствующая *спецификационная функция* или *отложенная реакция* имеет тип возвращаемого значения, отличный от `void`, то доступ к этому значению можно получить через идентификатор этой функции (реакции).

При тестировании систем с открытым состоянием, когда тестовая система имеет доступ к внутренним данным реализации, блок синхронизации должен привести модельное состояние в соответствие с реализационным:

```
stack *stack_impl;
List*  stack_model;

int push(stack*, int);
specification bool push_spec(Integer* i) reads i updates stack_model;

mediator push_media for
  specification bool push_spec(Integer* i) reads i updates stack_model
{
  ...
  state {
    int k;
    clear_List(stack_model);
    for (k = stack_impl->size; k > 0;
         append_List(stack_model, create_Integer(impl_stack->elems[--k]))
        );
  }
}
```

Поскольку код построения модельного состояния по внутреннему состоянию реализации будет одинаков для всех *спецификационных функций*, его удобно вынести в отдельную функцию, часто называемую mapStateUp.

При тестировании систем со скрытым состоянием, когда тестирующая система не имеет доступа к внутренним данным реализации, блок синхронизации должен действовать в предположении, что реализация отработала без ошибок в соответствии со спецификацией, и привести модельное состояние в вид, который ожидается получить после воздействия:

```
mediator push_media for
  specification bool push_spec(Integer* i) reads i updates stack_model
{
  ...
  state {
    add_List(stack_model, 0, create_Integer(push_spec));
  }
}
```

Блок синхронизации спецификационных функций исполняется тестовой системой непосредственно после *блока воздействия* перед проверкой *постусловия* в спецификационной функции.

Блок синхронизации отложенных реакций выполняется после возникновения *отложенной реакции* перед проверкой *постусловия*.

Сборщик реакций

Сборщик реакций служит для получение результата *отложенных реакций*.

Сборщики реакций являются реализационно-зависимыми компонентами. Их задача состоит в том, чтобы собрать все отложенные реакции целевой системы и зарегистрировать их в *регистраторе взаимодействий* (см. раздел "*Сервисы регистрации отложенных реакций*" главы "*Библиотека поддержки тестовой системы CTesK*" документа "*CTesK 2.2 Community Edition. Описание языка SeC*").

Сценарии

Обычная задача тестирования состоит в проверке поведения системы при воздействиях на нее через набор интерфейсных функций до достижения заданного уровня покрытия. Последовательность тестовых воздействий для решения такой задачи называется тестом.

Вызовы одной или нескольких *спецификационных функций* и способ перебора их параметров задаются в *сценарной функции*. Во время тестирования тестовая система находится в одном из состояний, называемых *сценарными состояниями*. Каждый вызов *сценарной функции* переводит тестовую систему из одного состояния в другое. Автоматически обеспечивается перебор всех параметров в каждом достигнутом сценарном состоянии.

Тестовый сценарий объединяет несколько *сценарных функций*, указывает механизм построения теста, задает функцию построения сценарного состояния, определяет способ инициализации и завершения работы тестовой и тестируемой систем.

На основании данных, содержащихся в *тестовом сценарии*, соответствующий тест генерируется автоматически.

Сценарная функция

Сценарные функции предназначены для описания наборов тестовых воздействий. Для этого сценарная функция определяет воздействия (вызовы *спецификационных функций*) и способ перебора их параметров. Все воздействия будут автоматически выполнены в каждом сценарном состоянии, достигнутом во время тестирования.

Также сценарные функции могут производить дополнительную проверку корректности поведения вызываемых функций.

Сценарная функция описывается как функция без параметров, возвращающая значение типа `bool` и помеченная ключевым словом **`scenario`**:

```
scenario bool f_scen();
```

В простейшем случае каждая сценарная функция соответствует ровно одной *спецификационной функции*. Если *спецификационная функция* не имеет аргументов, то тело сценарной функции должно содержать единственный вызов *спецификационной функции*. Сценарная функция должна возвратить `false`, если поведение системы некорректно, и `true` в нормальной ситуации. При этом нужно учесть, что проверка постусловий вызываемых *спецификационных функций* происходит автоматически и ее результаты учитывать не надо, т. е. проверка в сценарной функции носит дополнительных характер.

```
specification int f_spec(void);
scenario bool f_scen() {
  f_spec();
  return true;
}
```

Оператор итерации

Если у спецификационной функции есть аргументы, возникает необходимость их перебора. Для этого используются *операторы итерации*.

Оператор итерации используется только в сценарной функции и служит для параметризованного перебора тестовых воздействий. Синтаксически *оператор итерации* схож с циклом for:

```
iterate(декларация; усл_продолжения; инкремент; усл_фильтрации) тело;
```

Декларация является обязательным выражением и представляет собой декларацию итерационной переменной и ее инициализацию (в отличие от языка C, где переменная декларируется вне цикла). Итерационная переменная должна быть *допустимого типа* (см. раздел "*Допустимые типы SeC*"). Итерационная переменная не является обычной локальной переменной: поскольку итерация выполняется независимо во всех сценарных состояниях, для каждого состояния имеется свой экземпляр итерационной переменной со своим значением. При вызове сценарной функции в некотором состоянии, в качестве значения итерационной переменной используется то значение, которое эта переменная получила в предыдущий вызов в том же состоянии.

Условие продолжения и *инкремент* действуют так же, как аналогичные выражения цикла for.

Условие фильтрации представляет собой логическое выражение. Если оно равно `false`, то происходит переход к следующему значению итерационной переменной. Условие фильтрации можно опустить, в таком случае оно будет эквивалентно выражению `true`, т. е. ни одно значение итерационной переменной не будет отброшено.

Таким образом, следующая конструкция:

```
iterate(int i=0; i<10; i++; i&1==0) { ... }
```

в некотором смысле аналогична циклу for:

```
int i;
for(i=0; i<10; i++) {
  if (!(i&1==0)) continue;
  ...
}
```

Такой схемой можно пользоваться, чтобы представить последовательность вызовов, которая будет сгенерирована в пределах одного сценарного состояния. Однако следует понимать разницу между *оператором итерации* и циклом. Во-первых, значение итерационной переменной зависит от сценарного состояния, а переменная цикла — нет. Во-вторых, при одном вызове сценарной функции выполняется только одна итерация; она определяет переход из одного сценарного состояния в другое. Если же внутри сценарной функции будет использован обычный цикл, в рамках одного и того же перехода будут произведены все вызовы, перебираемые циклом.

Допускаются вложенные *операторы итерации*. Однако последовательные *операторы итерации* использовать нельзя.

При тестировании без отложенных реакций сценарное состояние изменяется в процессе тестового воздействия, происходящего в момент вызова *спецификационной функции*. Таким образом, если в результате работы *спецификационной функции* изменяются глобальные пере-

менные, после ее вызова в сценарной функции будут доступны уже измененные значения. Однако переменные состояния и итерационные переменные сохранят значения, соответствующие прежнему сценарному состоянию, до конца работы сценарной функции.

При тестировании с отложенными реакциями сценарное состояние не изменяется в ходе работы сценарной функции. Изменение происходит в конце работы сценарной функции, после того, как будут собраны все отложенные реакции.

Итерация по критерию покрытия

Важным частным случаем итерации является итерация по критерию покрытия *спецификационной функции*. Целью такой итерации является осуществление воздействия ровно по одному разу на каждую ветвь функциональности, определенную в критерии покрытия. Такая итерация позволяет сократить число воздействий, не теряя при этом качества покрытия, а также является средством борьбы с недетерминизмом, возникающем при обобщении в сценарии модельного состояния спецификации.

Итерация по критерию покрытия осуществляется с помощью конструкции **`coverage`**, вычисляющей достигнутую *ветвь функциональности* в *критерии покрытия* по умолчанию. Таким образом, итерация осуществляется по элементам (*ветвям функциональности* критерия покрытия, назначенном в тесте с помощью вызова функции `set_coverage_`*`имя_спецификационной_функции`*, либо, в противном случае, по элементам критерия, который явно объявлен критерием по умолчанию с помощью ключевого слова **`default`** или по последнему из критериев, определенных в спецификационной функции, если не один из них не объявлен критерием по умолчанию (см. раздел “Критерий покрытия”).

Пусть имеется некоторая итерация параметров:

```
scenario bool f_scen() {
  iterate(int i=0; i<10; i++; i&1==0) {
    iterate(double j=0.0; j<1.0; j+=0.01; j<0.4 || j>0.6) {
      f_spec(i,j);
    }
  }
  return true;
}
```

Для того, чтобы преобразовать ее таким образом, чтобы для каждой *ветви функциональности*, определенной в *критерии покрытия* C, был произведен ровно один вызов *спецификационной функции*, нужно ввести дополнительную итерацию по ветвям функциональности и заменить существующие операторы итерации на простые циклы:

```
scenario bool f_scen() {
  int i;
  double j;
  iterate(int cov=0; cov<get_coverage_size_f_spec(); cov++; ) {
    for(i=0; i<10; i++) {
      if (!(i&1==0)) continue;
      for(j=0.0; j<1.0; j+=0.01) {
        if (!(j<0.4 || j>0.6)) continue;
        if (cov == coverage f_spec(i,j)) {
          f_spec(i,j);
          goto done;
        }
      }
    }
    done:;
  }
  return true;
}
```

Функция `get_coverage_size_`*`имя_спецификационной_функции`*`()` возвращает количество ветвей функциональности в критерии покрытия по умолчанию. Конструкция **`coverage`** возвращает номер достигнутой на заданных параметрах *ветви*, а проверка этого номера на равенство итерационной переменной cov обеспечивает вызов *спецификационной функции*, соответствующий очередной *ветви функциональности*.

В более сложных случаях возникает необходимость в одной сценарной функции совершить несколько тестовых воздействий. Такая ситуация бывает обычной для тестирования систем с отложенными реакциями, когда в сценарной функции вызываются несколько спецификационных функций. В этом случае в блоке итерации просто записываются все необходимые вызовы.

Переменные сценарного состояния

Помимо итерационных переменных, существует и другой вид переменных, значение которых зависит от сценарного состояния — *переменные сценарного состояния*. Для каждого сценарного состояния имеется свой экземпляр такой переменной со своим значением. При вызове сценарной функции в некотором состоянии, в качестве значения переменной используется то значение, которое эта переменная получила в предыдущий вызов в том же состоянии.

Декларация переменных сценарного состояния начинается с модификатора **`stable`** и должна содержать инициализацию. Переменные сценарного состояния должны иметь *допустимый тип* (см. раздел “Допустимые типы SeC”).

```
stable int i = 0;
```

То, в каком месте сценарной функции объявлены такие переменные, влияет лишь на область их видимости, но не на функциональность.

Функция построения сценарного состояния

Во время работы тестовая система находится в одном из состояний. Функция построения сценарного состояния обычно вычисляет это состояние на основе значений переменных, определяющих модельное состояние. Сценарное состояние часто является обобщением модельного состояния, однако оно может и совпадать с модельным, а может быть вообще с ним не связано.

Функция не имеет параметров и возвращает ссылку на значение *спецификационного типа*:

```
typedef Object* (*PtrGetState)(void);
```

Пример функции, обобщающей модельное состояние, представляющее собой список, как его длину:

```
List* l;

Object* getState(void) {
  return create_Integer(size_List(l));
}
```

Функция определения стационарности состояния

Функция определения стационарности состояния используется только при тестировании систем с отложенными реакциями.

Функция не имеет параметров и возвращает логическое значение, показывающее, является ли текущее модельное состояние стационарным (стационарным называется состояние, в котором не ожидается возникновения отложенных реакций).

```
typedef bool (*PtrIsStationaryState)(void);
```

Пример:

```
List* expectedReactions;
...
bool isStationaryState() {
  return empty_List(expectedReactions);
}
```

Функция сохранения модельного состояния

Функция сохранения модельного состояния используется только при тестировании систем с отложенными реакциями.

Функция не имеет параметров и возвращает ссылку на значение *спецификационного типа*, содержащее текущее модельное состояние.

```
typedef Object* (*PtrSaveModelState)(void);
```

Если модельное состояние представлено переменной неспецификационного типа или несколькими переменными, потребуется определить новый *спецификационный тип*, включающий в себя все необходимые данные.

```
List* modelList;
int   modelInt;
specification typedef struct { List* a; int b; } ModelState = {};
...
Object* saveModelState() {
  return create(&type_ModelState, clone(modelList), modelInt);
}
```

Функция восстановления модельного состояния

Функция восстановления модельного состояния используется только при тестировании систем с отложенными реакциями.

Функция принимает один параметр — ссылку на значение *спецификационного типа*:

```
typedef void (*PtrRestoreModelState)(Object*);
```

На вход функции восстановления модельного состояния подается значение, возвращенное *функцией сохранения модельного состояния*. Задача функции состоит в том, чтобы привести текущее модельное состояние системы в соответствие с этим значением.

```
List* modelList;
int   modelInt;
specification typedef struct { List* a; int b; } ModelState = {};
...
void restoreModelState(Object* modelState) {
  ModelState* s = (ModelState*)modelState;
  modelList = clone(s->a);
  modelInt = s->b;
}
```

Функция инициализации

Функция инициализации предназначена для выполнения подготовительных работ перед проведением тестирования. Функция принимает два параметра, аналогичных параметрам функции main, а возвращает логическое значение, показывающее, успешно ли проведена инициализация:

```
typedef bool (*PtrInit)(int, char**);
```

Как правило, функция обеспечивает:

- инициализацию глобальных переменных модели
- инициализация реализации тестируемой системы
- установку медиаторов с помощью функций `set_mediator_`*`имя_спецификационной_функции`*
- если нужно, установку критерия покрытия по умолчанию для спецификационных функций с помощью функций `set_coverage_`*`имя_спецификационной_функции`*

При необходимости, функция инициализации может использовать переданные ей параметры запуска.

```
char   *impl_data;
String* model_data;

specification void f_spec(int a);
mediator f_media for specification void f_spec(int a);

bool init(int argc, char **argv) {
  impl_data  = (char*)malloc(SIZE);
  model_data = create_String("");
  set_mediator_f_spec(f_media);
  return (impl_date != NULL && model_data != NULL);
}
```

В случае тестирования систем с отложенными реакциями, функция инициализации дополнительно используется для:

- установки времени ожидания отложенных реакций
- если нужно, для распределения каналов обработки непосредственных и отложенных реакций

```
ChannelID chid;

bool init(int argc, char **argv) {
  chid = getChannelID();
  setStimulusChannel(chid);
  setWTime(1);
  ...
}
```

Функция завершения

Функция завершения предназначена для выполнения заключительных работ после проведения тестирования. Функция не имеет параметров и возвращаемого значения:

```
typedef void (*PtrFinish)(void);
```

Как правило, функция завершения освобождает ресурсы, выделенные в *функции инициализации*.

```
char   *impl_data;
String* model_data;

void finish(void) {
  free(impl_data);
  model_data = NULL;
  releaseChannelID(chid);
}
```

Тестовый сценарий

Тестовый сценарий предоставляет всю информацию, необходимую для автоматического построения теста. Он соответствует переменной специального структурного типа `dfsm` или `ndfsm`, помеченной модификатором **`scenario`**:

```
scenario dfsm testScenario;
```

В качестве имени типа выступает название обходчика, определяющего механизм построения теста:

- `dfsm` — Deterministic Finite State Machine (детерминированный конечный автомат);
- `ndfsm` — Nondeterministic Finite State Machine (недетерминированный конечный автомат).

Во время тестирования `dfsm` применяет тестовые воздействия, которые могут изменять состояние сценария. `dfsm` автоматически отслеживает все изменения состояния и строит конечный автомат, соответствующий процессу тестирования. Состояниями автомата являются все достижимые состояния сценария, а переходы автомата помечаются тестовыми воздействиями, которые их инициировали. Тестирование заканчивается только тогда, когда обходчик подаст все определенные пользователем тестовые воздействия во всех состояниях сценария достижимых из начального.

Чтобы достижение этой цели было возможно, необходимо выполнение следующих условий:

- **Конечность**

  Число состояний сценария, достижимых из начального путем применения тестовых воздействий из заданного набора, должно быть конечным.

- **Детерминированность**

  Для любого состояния сценария применение одного и того же тестового воздействия всегда должно переводить систему в одно и тоже состояние сценария.

- **Сильная связность**

  Из любого состояния сценария достижимо любое другое состояние сценария путем применения последовательности тестовых воздействий.

Обходчик `ndfsm` может корректно работать на более широком классе автоматов, а именно на конечных атоматах, имеющих детерминированный сильносвязный полный остовный подавтомат:

- **Остовный подавтомат**

  Остовный подавтомат содержит все состояния сценария, достижимые в процессе тестирования.

- **Полный подавтомат**

  Полный подавтомат для каждого состояния сценария и допустимого в нем тестового воздействия либо содержит все переходы из этим состояния, помеченные этим тестовым воздействием, либо не содержит таких переходов вовсе.

Обходчик `ndfsm` не предназначен для тестирования систем с отложенными реакциями.

Определение тестового сценария должно содержать инициализатор следующего вида:

```
scenario dfsm testScenario = {
  .init     = init,
  .getState = getState,
  .actions  = {
    f_scen,
    g_scen,
    NULL
  },
  .finish   = finish
};
```

В поле `init` задается имя *функции инициализации*. Это поле может быть опущено, однако обычно *функция инициализации* присутствует как минимум для установки *медиаторов*.

В поле `getState` задается имя *функции построения сценарного состояния*. Если это поле опущено, то тестирование ведется в одном состоянии.

В поле `actions` задается список сценарных функций, включенных в данный тест, завершенный значением `NULL`. Это поле является обязательным.

В поле `finish` задается имя *функции завершения*. Это поле может быть опущено.

Тестовый сценарий вызывается как функция с идентификатором тестового сценария, принимающая два параметра, аналогичных параметрам main, без возвращаемого значения. Обычно вызов производится в функции main:

```
int main(int argc, char **argv) {
  testScenario(argc,argv);
  return 0;
}
```

При вызове тестового сценария происходит разбор переданных ему параметров. Тестовая система обрабатывает следующие стандартные параметры:

- **-tc** — направить трассировку на консоль
- **-tt** — направить трассировку в файл с уникальным именем, составленным из имени сценария и текущего времени
- **-t *имя_файла*** — направить трассировку в файл с указанным именем
- **-nt** — выключить трассировку

Запуск теста без указанных параметров командной строки эквивалентен запуску с параметром **-tt**.

- **--trace-accidental** — включить трассировку недостоверных переходов
- **-uerr** — выполнять тестирование до возникновения первой ошибки (по умолчанию)
- **-uerr=*число*** — выполнять тестирования до возникновения ***число*** ошибок (только для `ndfsm`)
- **-uend** — выполнять тестирование до конца, несмотря на ошибки

- **--find-first-series-only**, **-ffso** — находить только первую успешную серию

Обработанные стандартные параметры удаляются из списка параметров, и измененный список передается *функции инициализации* сценария.

Допускается несколько вызовов сценариев из одной программы. Следует заметить, что если параметры, переданные исполняемому файлу теста из командной строки, будут просто переданы во все вызываемые сценарии, то при направлении трассировки в файл трасса очередного сценария перезапишется в тот же файл, что и трасса предыдущего. Для того, чтобы в одном файле оказались трассы всех сценариев, следует воспользоваться функциями `addTraceToFile()` и `removeTraceToFile()`[1].

```
int main(int argc, char **argv) {
  addTraceToFile("trace.utt");
  testScenario1(argc,argv);
  testScenario2(argc,argv);
  removeTraceToFile("trace.utt");
  return 0;
}
```

В случае тестирования систем с отложенными реакциями, в определении тестового сценария требуется задать большее число полей:

```
scenario dfsm testScenario = {
  .init              = init,
  .getState          = getState,
  .isStationaryState = isStationaryState,
  .saveModelState    = saveModelState,
  .restoreModelState = restoreModelState,
  .actions           = {
    f_scen,
    g_scen,
    NULL
  },
  .finish            = finish
};
```

Поля init, `getState`, `actions` и `finish` имеют тот же смысл, что и обычно. Дополнительные три поля являются обязательными.

В поле `isStationaryState` задается имя *функции определения стационарности состояния*.

В поле `saveModelState` задается имя *функции сохранения модельного состояния*.

В поле `restoreModelState` задается имя *функции восстановления модельного состояния*.

[1] Подробнее описаны в разделе “*Сервисы трассировки*” главы “*Библиотека поддержки тестовой системы CTesK*” документа “*CTesK 2.2. Описание языка SeC*”.

Трансляция и сборка тестов

Перед выполнением теста файлы с исходным кодом на языке SEC должны быть оттранслированы в файлы, содержащие код на языке программирования C. Полученные в результате трансляции файлы могут быть скомпилированы любым компилятором языка C, например, gcc. Для того чтобы получить исполняемый тест, необходимо скомпоновать полученные в результате компиляции объектные файлы с библиотеками инструмента CTesK.

В этой главе рассказывается как осуществить трансляцию SEC файлов в C файлы, описываются макроопределения, доступные в SEC файлах и позволяющие влиять на трансляцию, а также библиотеки инструмента CTesK.

Трансляция SEC файлов

Для трасляции SEC файлов в C файлы предназначена команда `sec`.

Для платформы Linux команда имеет следующий формат:

```
sec.sh имя_sec_файла имя_c_файла имя_препроцессированного_файла
[опции_препроцессора]
```

Для трансляции файла **account_scenario.sec** в файл, содержащий код на C, на платформах под управлением ОС Linux необходимо в командном интерпретаторе в каталоге **examples/account** в дереве каталогов установки CTesK выполнить команду:

```
sec.sh account_scenario.sec account_scenario.c account_scenario.sei
```

В результате в каталоге **examples/account** должен сгенерироваться файл **account_scenario.c**.

Стандартные макроопределения

В файлах, содержащих исходный код теста, можно использовать следующие макроопределения, позволяющие влиять на трансляцию:

- Индикатор SEC файла — `SEC`

  Позволяет определить тип файла, в котором расположен код: SEC файл, если `SEC` определен, или C файл, в противном случае.

- Версия инструмента CTesK — `CTESK_VERSION`

  Определяет версию используемого транслятора CTesK.

- Идентификатор сборки инструмента CTesK — `CTESK_BUILD`

  Определяет идентификатор сборки используемого транслятора CTesK.

Библиотеки CTesK

CTesK поставляется со следующим набором библиотек:

- **atl** — библиотека абстрактных типов данных;
- **tracer** — библиотека функций трассировки событий;
- **ts** — библиотека поддержки времени исполнения SEC;
- **utils** — библиотека поддержки тестовой системы.

Анализ результатов тестирования и отладка тестов

При выполнении теста автоматически генерируется *трасса*, в которую попадает подробная информация о событиях, происходящих во время тестирования. Для анализа результатов тестирования по трассе теста стоятся *тестовые отчеты*. *Статические отчеты* содержат информацию о найденных нарушениях, о достигнутом покрытии, о достигнутых состояниях и переходах между ними. *Графические отчеты* позволяют подробно изучить ход тестирования в динамике. Анализ результатов показывает, насколько полным было проведенное тестирование, есть ли ошибки в реализации и требуется ли дальнейшая доработка тестов.

В этой главе используется пример теста для системы, реализующей функциональность банковского кредитного счета, который можно найти в дистрибутиве CTesK в каталоге **examples/account**.

Трасса теста

Трасса теста генерируется в процессе тестирования и представляет собой файл в формате XML. Для анализа результатов тестирования значительно удобнее использовать *тестовые отчеты*, а не саму трассу.

Сообщения

Ниже перечислены основные типы сообщений, сбрасываемых в трассу:

1. Начало и конец трассы
2. Начало и конец сценария
3. Достижение сценарного состояния
4. Начало и конец перехода между состояниями
5. Значение сценарного объекта (состояние, имя сценарной функции, значение итерационной переменной и т. п.)
6. Начало и конец вызова спецификационной функции
7. Значение модельного объекта (значение аргумента функции и т. п.)
8. Начало и конец сериализации
9. Начало и конец работы оракула
10. Структура покрытия
    a. Логические формулы (инварианты и проверки доступа только на чтение)
    b. Критерии покрытия с ветвями функциональности
11. Значение логической формулы
12. Окончание проверки предусловия
13. Покрытие ветви функциональности
14. Возникновение ошибки
15. Пользовательское сообщение

Ниже в таблице приведен пример трассы теста:

<trace>	Начало трассы
<**scenario**_start trace="1" name="pqueue" time="1068567432000" host="CAMEL" os="Linux"/>	Начало сценария
<state trace="1" id="0"/>	Сценарное состояние
<**scenario**_value trace="1" kind="state" type="" name=""><![CDATA[struct { 0, 0 }]]></**scenario**_value>	Значение сценарного объекта
<transition_start trace="1" id="0"/>	Начало перехода
<**scenario**_value trace="1" kind="**scenario** method" type="" name=""> <![CDATA[enq_scen]]> </**scenario**_value>	Значение сценарного объекта
<**scenario**_value trace="1" kind="iteration variable" type="int" name="i"> <![CDATA[0]]> </**scenario**_value>	Значение сценарного объекта
<model_operation_start trace="1" signature="void enq_spec(Item item)"/>	Начало спецификационной функции
<model_value trace="1" kind="argument" type="Item" name="item"> <![CDATA[00303478]]> </model_value>	Значение модельного объекта

<oracle_start trace="1" signature="void enq_spec(Item item)"/>	Начало оракула
<**coverage**_structure trace="1">	Начало структуры покрытия
<formulae> <formula id="0"><![CDATA[**invariant** type Item (@item)]]></formula> <formula id="1"><![CDATA[**invariant** type Item (item)]]></formula> <formula id="2"><![CDATA[**reads** item]]></formula> </formulae>	Логические формулы
<**coverage** id="c1"> <element id="0" name="single branch "/> </**coverage**>	Критерии покрытия и ветви функциональности
</**coverage**_structure>	Конец структуры покрытия
<user_info trace="1"><![CDATA[Oracle call]]></user_info>	Пользовательское сообщение
<prime_formula trace="1" id="0" value="true"/>	Значение формулы
<**pre**condition_end trace="1"/>	Завершение проверки предусловия
<**coverage**_element trace="1" **coverage**="c1" id="0"/>	Покрытие ветви функциональности
<prime_formula trace="1" id="1" value="false"/>	Значение формулы
<exception trace="1" internal="false"> <where></where> <info><![CDATA[Postcondition failed]]></info> </exception>	Сообщение об ошибке
<oracle_end trace="1"/>	Конец оракула
<model_operation_end trace="1"/>	Конец спецификационной функции
<transition_end trace="1"/>	Конец перехода
<state trace="1" id="0"/>	Сценарное состояние
<**scenario**_value trace="1" kind="state" type="" name=""><![CDATA[struct { 0, 0 }]]></**scenario**_value>	Значение сценарного объекта
<**scenario**_end trace="1"/>	Конец сценария
</trace>	Конец трассы

Управление трассировкой

При запуске теста можно использовать следующие параметры командной строки, управляющие трассировкой:

- **-tc** — направить трассировку на консоль
- **-tt** — направить трассировку в файл с уникальным именем, составленным из имени сценария и текущего времени

- **-t *имя_файла*** — направить трассировку в файл с указанным именем
- **-nt** — выключить трассировку

Запуск теста без указанных параметров командной строки эквивалентен запуску с параметром **-tt**.

- **--trace-accidental** — включить трассировку недостоверных переходов

Программно трассировка на консоль назначается функциями `addTraceToConsole()` и `removeTraceToConsole()`, а в файл — функциями `addTraceToFile()` и `removeTraceToFile()`[2]:

```
int main(int argc, char **argv) {
  addTraceToConsole();
  addTraceToFile("trace.utt");
  testScenario(argc,argv);
  removeTraceToFile("trace.utt");
  removeTraceToConsole();
  return 0;
}
```

По умолчанию в трассу не попадают *недостоверные переходы* (переходы, в которых не было вызова оракула). Изменить это поведение можно с помощью функции `setTraceAccidental()`[3]:

```
int main(int argc, char **argv) {
  setTraceAccidental(true); // трассировка всех переходов
  testScenario(argc,argv);
  return 0;
}
```

Во время тестирования в трассу можно записывать произвольные пользовательские сообщения с помощью функции `traceUserInfo()`[4]:

```
Object* o;
String* s;
...
s = create_String("o = ");
s = concat_String(s, toString(o));
traceUserInfo(toCharArray_String(s));
```

Также имеется возможность форматированного вывода пользовательских данных в трассу с помощью функции `traceFormattedUserInfo()`[5]. В функции допускаются все спецификаторы преобразований, допустимые в стандартной функции `printf()`, а также спецификатор `$(obj)` для преобразования в строку спецификационного объекта. Спецификаторы `$(obj)` должны располагаться перед спецификаторами функции `printf()`:

```
int i;
Object* o;
...
traceFormattedUserInfo("o = $(obj), i = %d", o, i);
```

По умолчанию кодировка трасса UTF-8. Можно установить другую кодировку трассы с помощью функции `setTraceEncoding()`[6]:

---

[2] Подробнее описаны в разделе “*Сервисы трассировки*” главы “*Библиотека поддержки тестовой системы CTesK*” документа “*CTesK 2.2 Community Edition: Описание языка SeC*”.
[3] См. там же.
[4] См. там же.
[5] См. там же.
[6] См. там же.

```
int main(int argc, char **argv) {
  // установка кодировки трассы KOI8-R
  setTraceEncoding("KOI8-R");
  testScenario(argc,argv);
  return 0;
}
```

Статические отчеты

Статический отчет генерируется по одной или нескольким трассам и содержит информацию о результатах тестирования, о покрытии ветвей функциональности спецификационных функций, о достигнутых состояниях и переходах между ними. Трасса теста содержит всю информацию о выполнении теста, статический отчет представляет эту информацию в наглядной и компактной форме.

Отчет представляет собой набор связанных ссылками HTML-файлов. Логически он состоит из трех разделов:

- Пройденные сценарии тестирования
- Покрытие спецификационных функций
- Обнаруженные ошибки

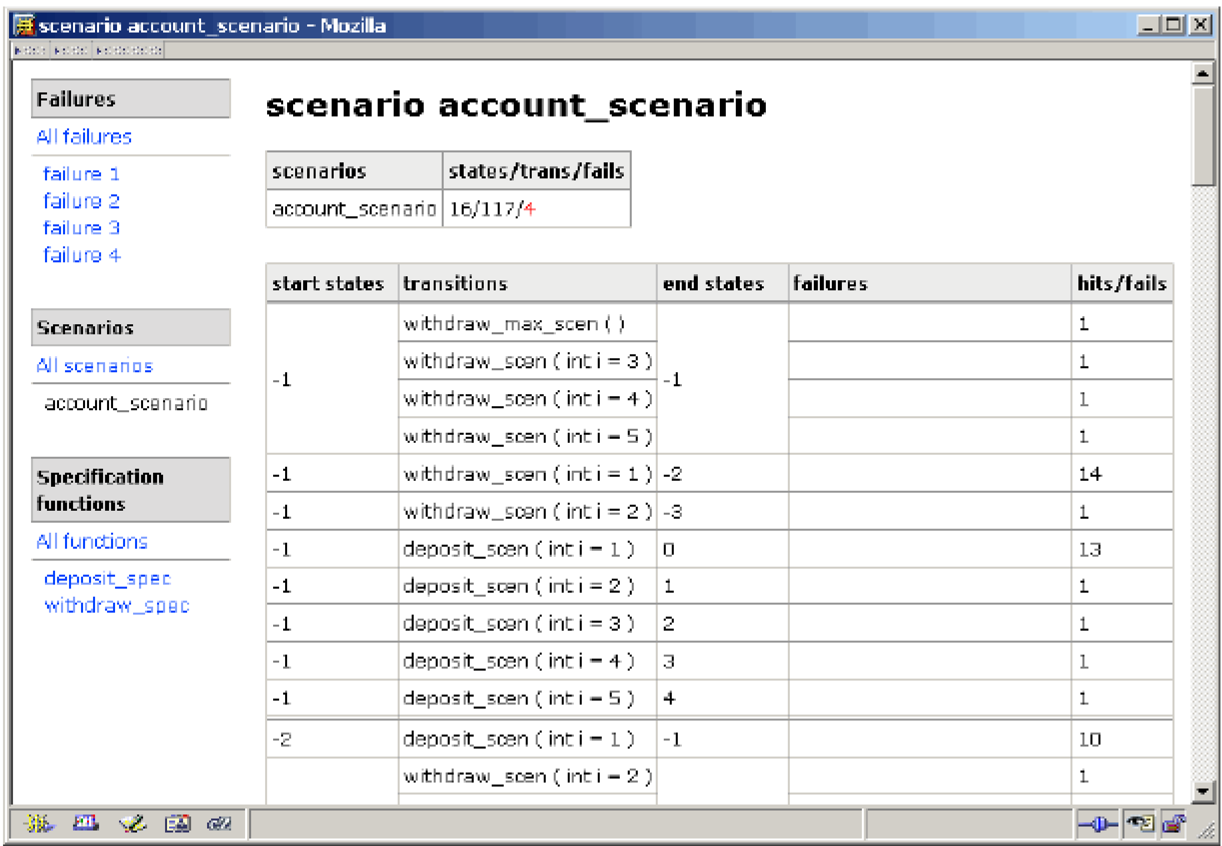
Каждый раздел содержит общую информацию и подразделы с подробной информацией о конкретных ошибках, сценариях или функциях.

Сценарии

Общая страница отчета о сценариях содержит сведения о количестве состояний, переходов и ошибок для каждого сценария.

Подробная информация о сценарии включает в себя:

- Количество состояний, переходов и ошибок
- Список ошибок (если имеются)
- Список переходов между состояниями с указанием количества попаданий на каждый переход



scenarios	states/trans/fails
account_scenario	16/117/4

start states	transitions	end states	failures	hits/fails
-1	withdraw_max_scen ( )	-1		1
	withdraw_scen (int i = 3)			1
	withdraw_scen (int i = 4)			1
	withdraw_scen (int i = 5)			1
-1	withdraw_scen (int i = 1)	-2		14
-1	withdraw_scen (int i = 2)	-3		1
-1	deposit_scen (int i = 1)	0		13
-1	deposit_scen (int i = 2)	1		1
-1	deposit_scen (int i = 3)	2		1
-1	deposit_scen (int i = 4)	3		1
-1	deposit_scen (int i = 5)	4		1
-2	deposit_scen (int i = 1)	-1		10
	withdraw_scen (int i = 2)			1

Рисунок 1. Отчет с подробной информацией о сценарии

Спецификационные функции

Общая страница *отчета* о покрытии спецификационных функций содержит таблицу, в которой для каждого покрытия каждой протестированной спецификационной функции указывается:

- Имя спецификационной функции
- Имя покрытия
- Процент покрытых ветвей функциональности (в скобках указывается количество покрытых ветвей и общее количество ветвей)
- Количество вызовов данной спецификационной функции и количество ошибок, обнаруженных при этих вызовах

Подробная информация о спецификационной функции включает в себя:

- Полную сигнатуру спецификационной функции
- Таблицу, в которой для каждой ветви функциональности каждого покрытия указывается количество попаданий в данную ветвь и количество обнаруженных в ней ошибок. Если нет ни одного попадания, то соответствующая строка подсвечивается красным цветом, а иначе — зеленым
- Список ошибок, относящихся к данной функции (если имеются)

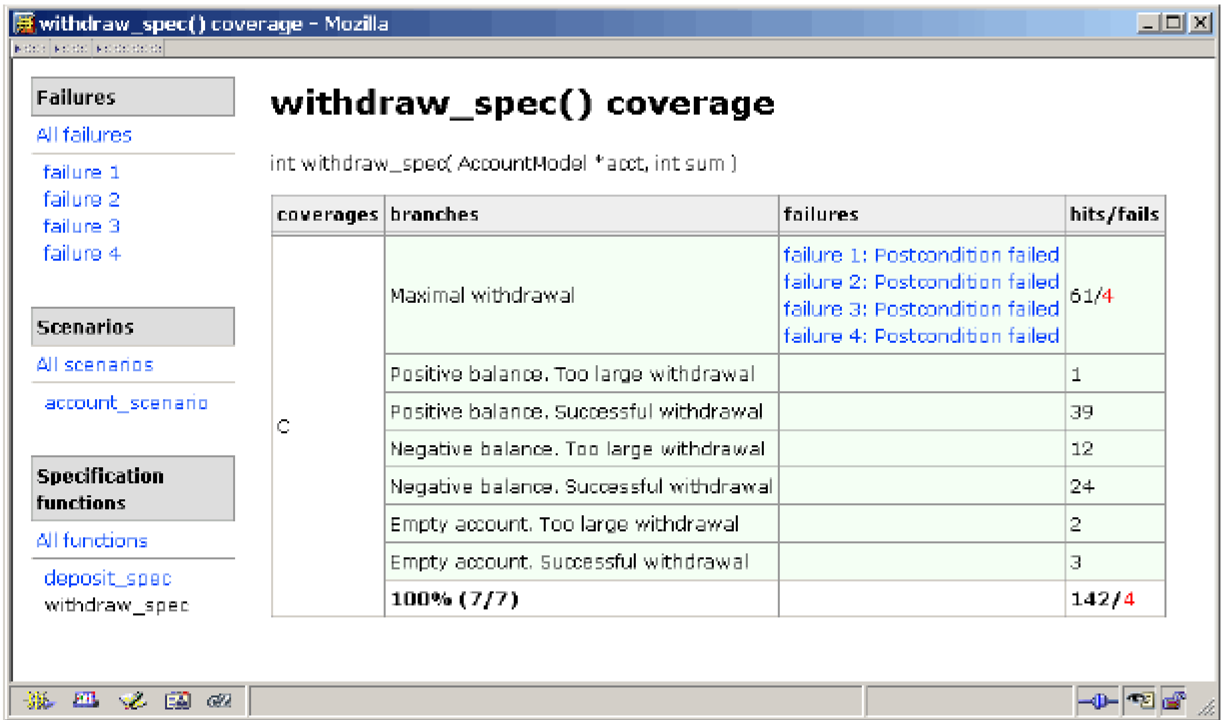


Рисунок 2. Отчет с подробной информацией о спецификационной функции

Ошибки

Общая страница отчета об ошибках содержит список всех обнаруженных ошибок (если данная ошибка была обнаружена в оракуле спецификационной функции, то указывается имя этой функции).

Подробная информация об ошибке включает в себя как минимум:

- Имя файла трассы и номер строки в нем
- Название сценария

В зависимости от контекста, в котором возникла ошибка, может присутствовать и другая уточняющая информация:

- Состояние
- Переход (имя сценарной функции и значения итерационных переменных)
- Имя спецификационной функции и ее параметры
- Информация о попадании в ветвь функциональности
- Значения логических формул

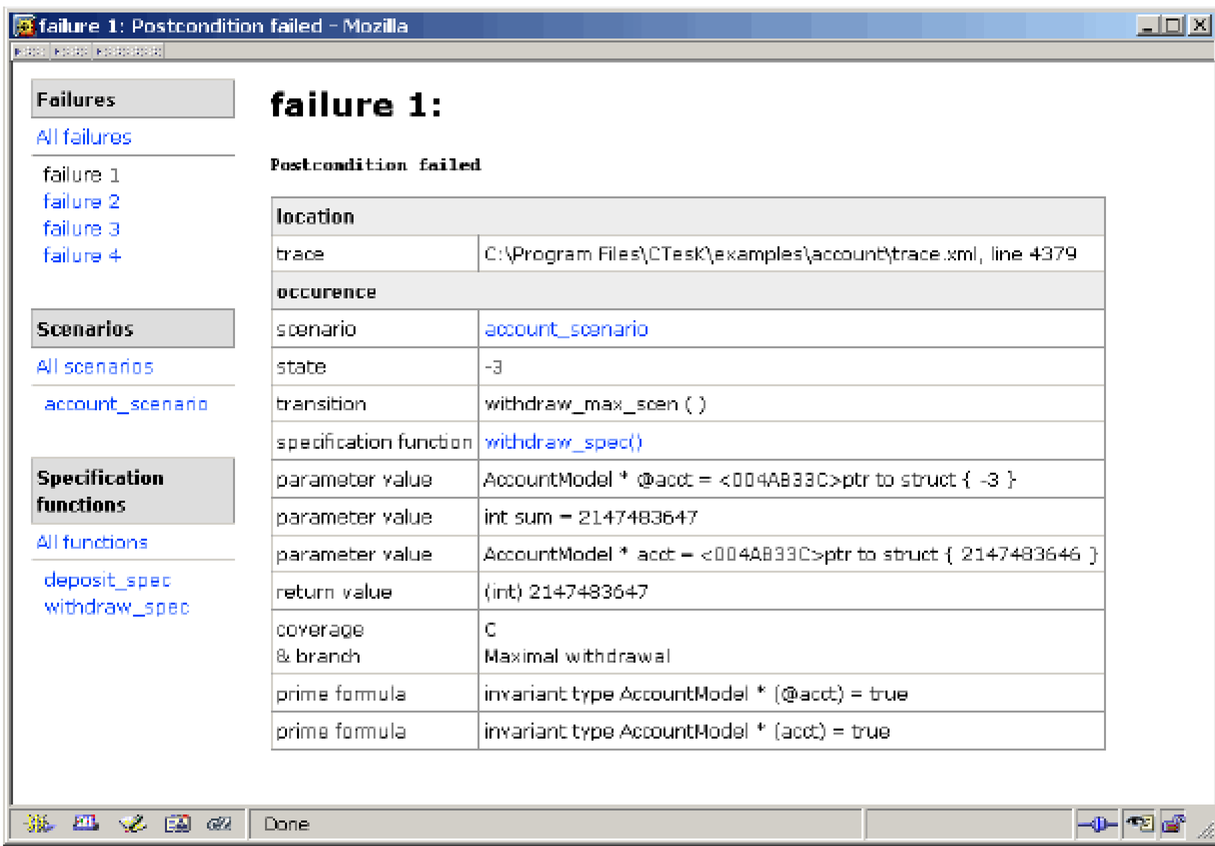


Рисунок 3. Отчет с подробной информацией о нарушении

Анализ результатов

Получив трассу теста, необходимо провести анализ результатов тестирования для того, чтобы выяснить, насколько успешно и полно было проверено выполнение требований спецификации, и не требуется ли доработка теста.

В ходе анализа решаются две задачи:

1. Поиск ошибок
2. Определение полноты покрытия

Поиск ошибок

Пусть при тестировании была обнаружена некоторая ошибка. Требуется выяснить причину возникновения ошибки и локализовать ее.

В ходе тестирования могут быть выявлены следующие типы ошибок:

- Нарушение постусловия
- Нарушение инварианта или ограничения доступа
- Недетерминированность графа состояний
- Нарушение сильной связности графа состояний

- Ошибка при инициализации
- Внутренние и пользовательские ошибки

Далее каждый тип подробно описан.

Нарушение постусловия

Нарушение постусловия свидетельствует о несоответствии между спецификацией и реализацией тестируемой системы.

Пусть в результате тестирования мы получили отчет, показанный на рисунке. Проанализируем его.

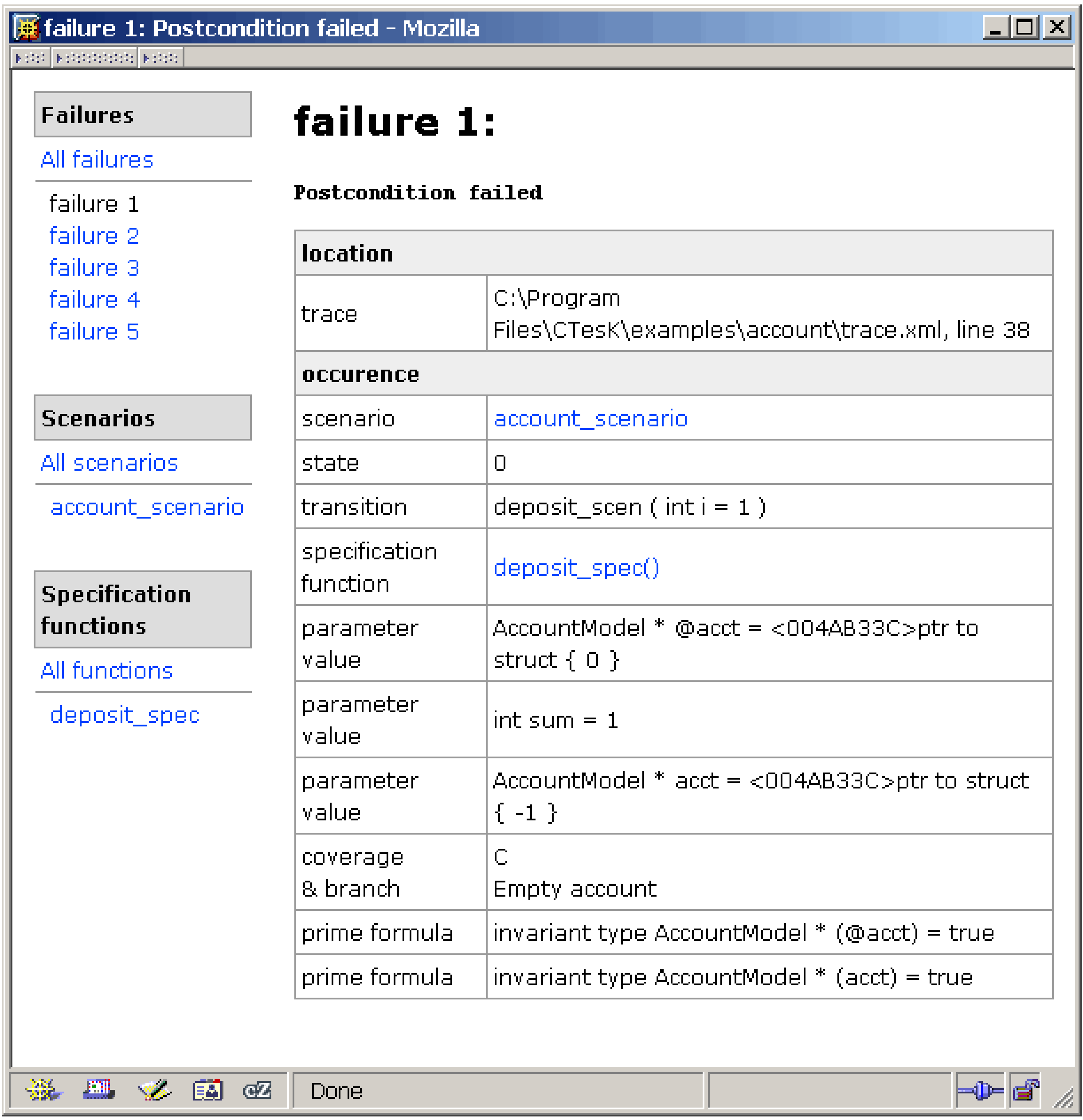


Рисунок 4. Отчет о нарушении постусловия

Мы видим, что имеет место нарушение постусловия (**Postcondition failed**), которое произошло в следующем контексте:

- Сценарий **account_scenario**
- Сценарная функция **deposit_scen** вызвана в состоянии **0** с параметром **i = 1**

- Оракул спецификационной функции **deposit_spec** вызван с параметр **sum = 1**, презначение параметра **acct** — указатель на структуру с нулевым полем, постзначение параметра **acct** — указатель на структуру с полем, равным **-1**
- Ошибка обнаружилась в ветви “**Empty account**” критерия покрытия “**C**”
- Инвариант параметра **sum** и инварианты пре- и постзначений параметра **acct** выполнены

Обратимся к коду спецификационной функции `deposit_spec()`:

```
specification void deposit_spec (AccountModel *acct, int sum)
  reads   sum
  updates balance = acct->balance
{
  pre { return (acct != NULL) && (sum > 0) && (balance <= INT_MAX - sum); }
  coverage C {
    if (balance + sum == INT_MAX)
      return {maximum, "Maximal deposition"};
    else if (balance > 0)
      return { positive, "Positive balance" };
    else if (balance < 0)
      if (balance == -MaximalCredit)
        return {minimum, "Minimal balance"};
      else
        return { negative, "Negative balance" };
    else
      return { zero, "Empty account" };
  }
  post {
    return balance == @balance + sum;
  }
}
```

Из него можно сделать вывод о том, что нарушено условие `balance == @balance + sum`, и действительно, неверно, что `-1 == 0 + 1`. Ошибку следует искать в реализации функции `deposit()`:

```
void deposit (Account *acct, int sum) {
  acct->balance -= sum;
}
```

Действительно, вместо того, чтобы добавлять сумму на счет, функция тестируемой системы снимает эту сумму со счета.

Нарушение инварианта или ограничения доступа

Нарушение инварианта или ограничения доступа проявляется в отчете следующим образом:

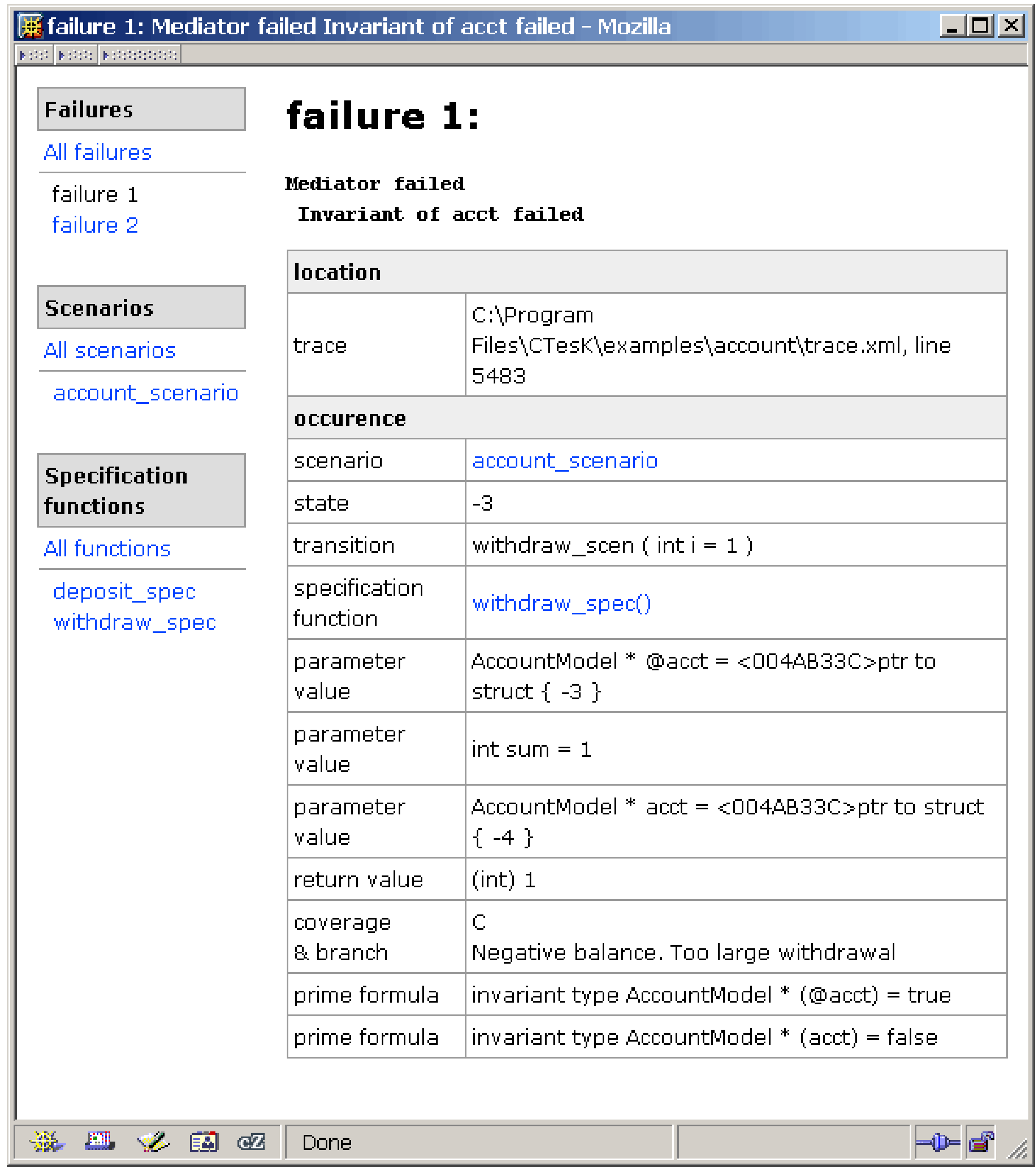


Рисунок 5. Отчет о нарушении постусловия из-за нарушения инварианта

Проанализируем отчет. Мы видим, что произошло нарушение постусловия в следующем контексте:

- Сценарий **account_scenario**
- Сценарная функция **withdraw_scen** вызвана в состоянии **-3** с параметром **i = 1**
- Оракул спецификационной функции **withdraw_spec** вызван с параметром **sum = 1**, пре-значение параметра **acct** — указатель на структуру с полем, равным минус трем, постзначение параметра **acct** — указатель на структуру с полем, равным минус четырем, возвращаемое значение **-1**
- Ошибка обнаружилась в ветви “**Negative balance. Too large withdrawal**” критерия покрытия “**C**”

- Инвариант типа для презначения параметра **acct** выполнен
- Инвариант типа для постзначения параметра **acct** нарушен

Для инварианта типа `AccountModel` определен следующий инвариант:

```
invariant (AccountModel acct) {
  return acct.balance >= -MaximalCredit;
}
```

Следовательно, нарушенным оказалось условие `acct.balance >= -MaximalCredit`. Можно сделать вывод о том, что в функции `withdraw()` произошла попытка снять со счета такую сумму, что кредит превысил максимальное значение, а тестируемая система, тем не менее, разрешила снятие. Обратимся к реализации функции `withdraw()`:

```
int withdraw (Account *acct, int sum) {
  //if (acct->balance - sum < -MAXIMUM_CREDIT)
  //  return 0;
  acct->balance -= sum;
  return sum;
}
```

В самом деле, код, не допускающий снятие сумм с превышением кредита, оказался закомментированным.

Недетерминированность графа состояний

Недетерминированность возникает, когда в одном и том же обобщенном состоянии одна и та же сценарная функция, вызванная при одних и тех же значениях итерационных переменных, в разных случаях переводит систему в разные обобщенные состояния.

Недетерминированность проявляется в отчете следующим образом:

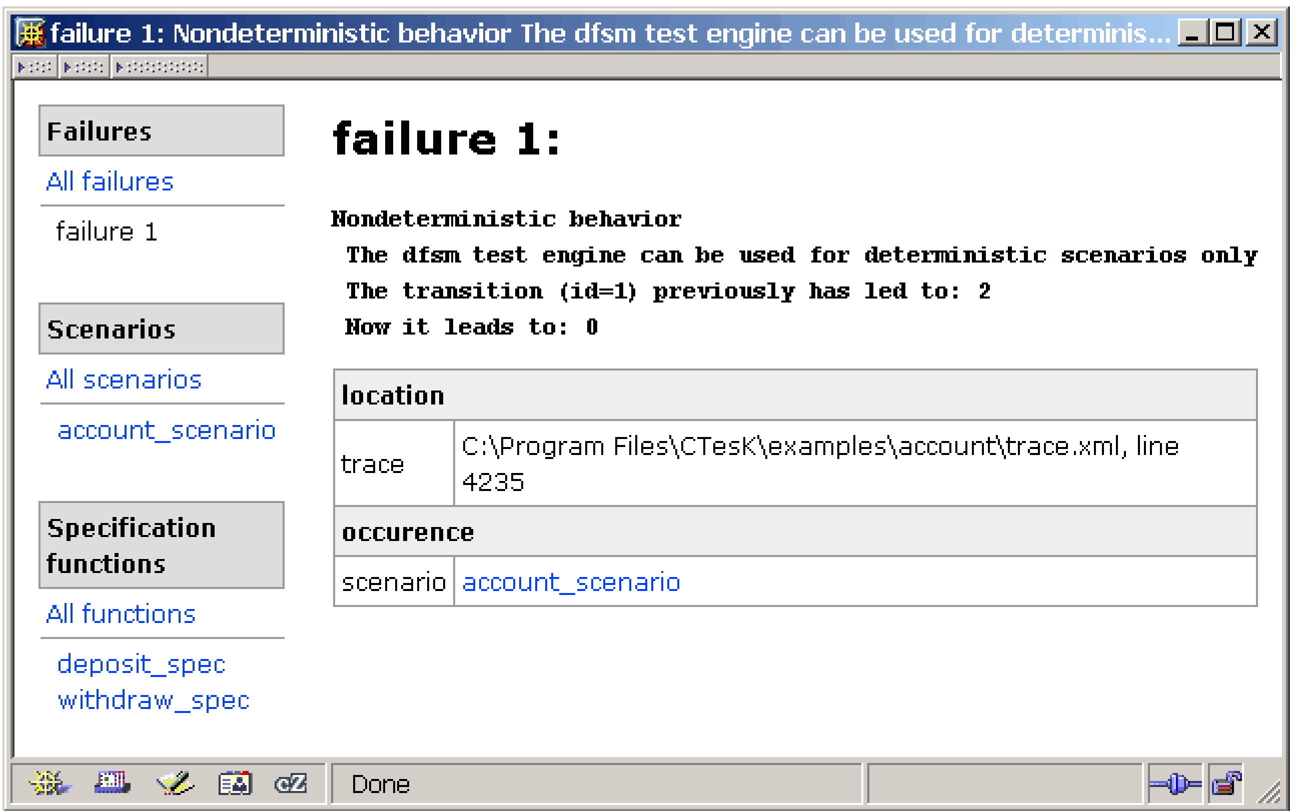


Рисунок 6. Отчет о нарушении детерминированности

Чтобы понять, какая функция привела систему в разные состояния, нужно посмотреть на отчет по сценарию, в котором представлена таблица всех переходов:

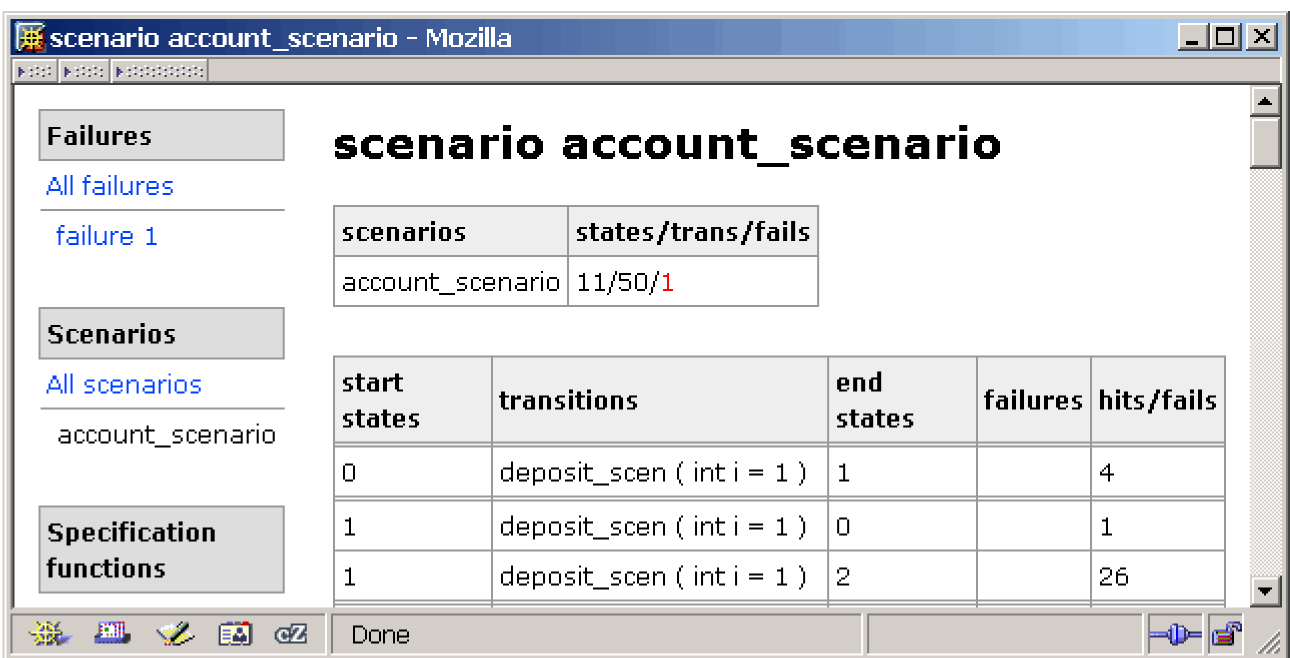


Рисунок 7. Отчет о сценарии с нарушением детерминированности

Как видно, сценарная функция **deposit_scen** вызывалась из состояния **1** с параметром **i = 1** всего 27 раз, причем **26** вызовов привели систему в состояние **2**, а один — в состояние **0**.

Причиной такого поведения может быть как реальная недетерминированность поведения функции, так и проблемы с обобщением состояния. В данном случае именно они имеют место, что становится ясно после рассмотрения кода функции, возвращающей модельное состояние:

```
static Integer* account_state() {
  return create_Integer(abs(acct.balance));
}
```

Здесь в качестве обобщенного состояния выбрано абсолютное значение баланса. Обобщенное состояние **1** в таком случае соответствует состояниям **1** и **-1**, но поведение функции в этих состояниях будет различно, что и приводит к ошибке.

Другое проявление недетерминированности, связанное с неправильным обобщением состояния, возможно в случае, когда в некотором обобщенном состоянии некоторая ветвь функциональности один раз достигается, а другой раз — нет. В этом случае будет зафиксирована ошибка — “**Can't find suitable parameters**”.

Это значит, что в данном обобщенном состоянии ранее была достигнута некоторая ветвь функциональности. Однако сейчас ни одно из значений параметров, перебираемых операторами итерации, не приводит к достижению этой ветви.

Нарушение сильной связности графа состояний

Нарушение сильной связности возникает в том случае, когда для некоторого перехода из одного обобщенного состояния в другое нет возможности вернуться в прежнее состояние с помощью какой-нибудь последовательности переходов.

При этом отчет выглядит следующим образом:

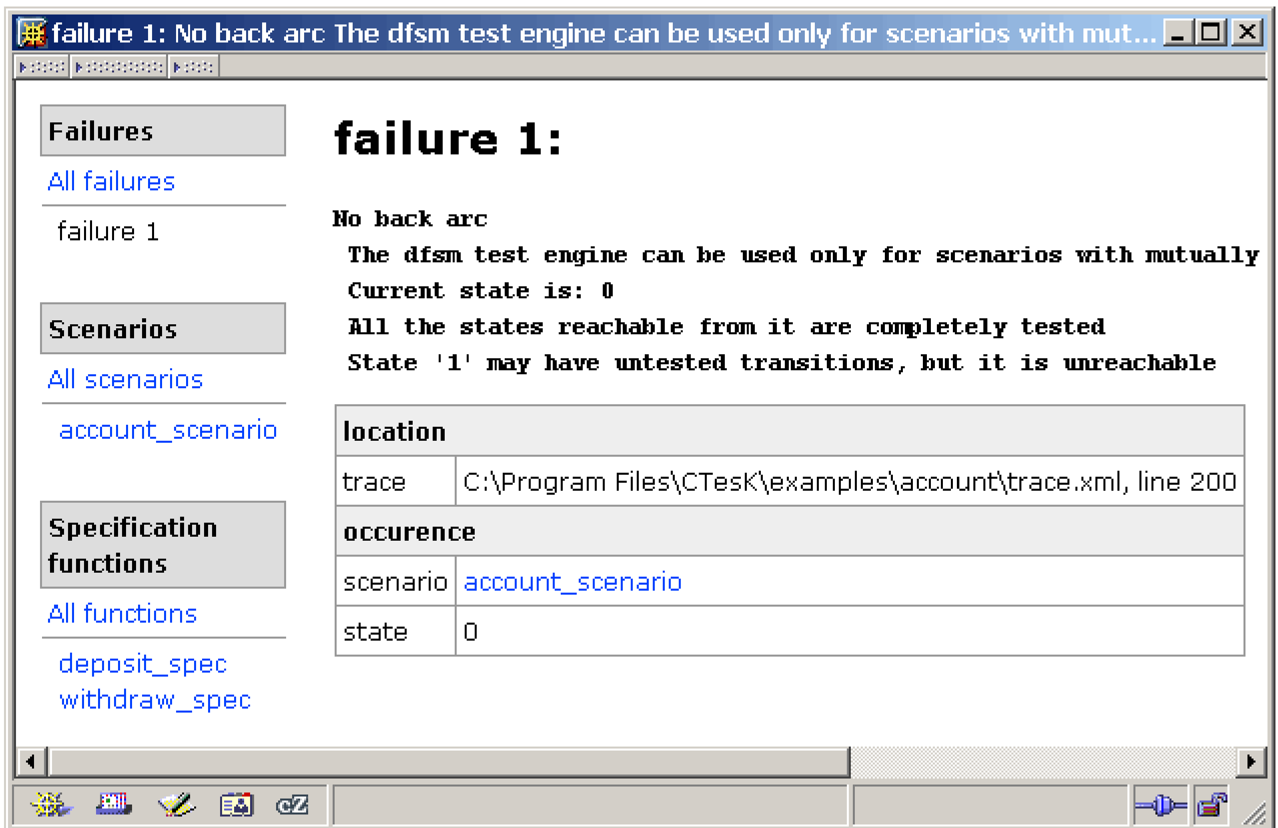


Рисунок 8. Отчет о нарушениии связности

Возникла ситуация, в которой был совершен некоторый переход, и нет возможности вернуться обратно.

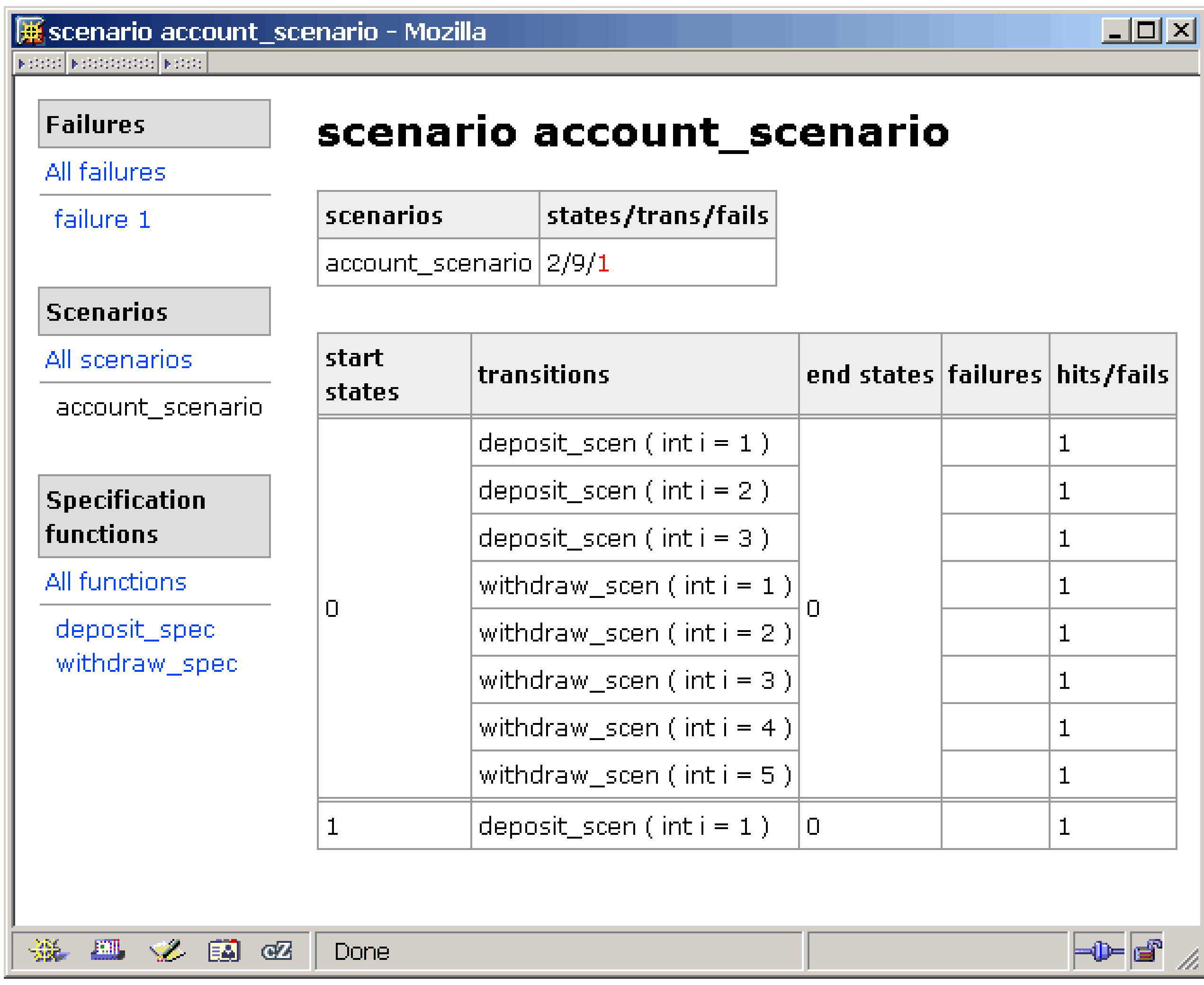


Рисунок 9. Отчет о сценарии с нарушением связности

Как видно, произошел переход из состояния **1** в состояние **0**, из которого все переходы ведут в то же состояние **0**. Это наводит на мысль о неправильной факторизации: обобщенное состояние было выбрано так, что условие сильной связанности оказалось нарушенным.

Посмотрим теперь на код функции, возвращающей модельное состояние:

```
static Integer* account_state() {
  return create_Integer(acct.balance == 0);
}
```

В самом деле, здесь определяются два обобщенных состояния, соответствующие нулевому и ненулевому балансу. Однако сценарные функции не обеспечивают параметры, при которых возможен переход обратно в состояние ненулевого баланса.

Ошибка при инициализации

Функция инициализации сценария может завершиться с ошибкой, если, например, не удалось инициализировать тестируемую систему или выделить необходимую память.

В отчете ошибка при инициализации показывается следующим образом:

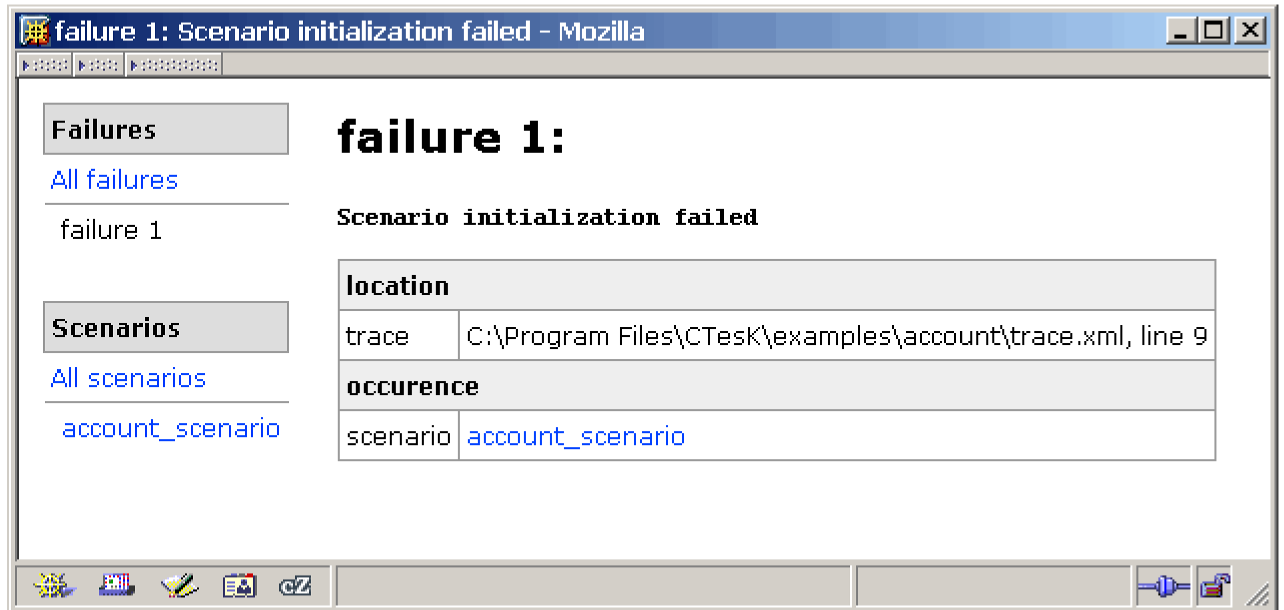


Рисунок 10. Отчет о нарушении при инициализации сценария

При возникновении такой ошибки ее причину следует искать в функции инициализации сценария.

Внутренние и пользовательские ошибки

Внутренняя ошибка возникает при сбое в тестовой системе. Пользователь также может вызвать появление ошибки в трассе с помощью `assertion()`:

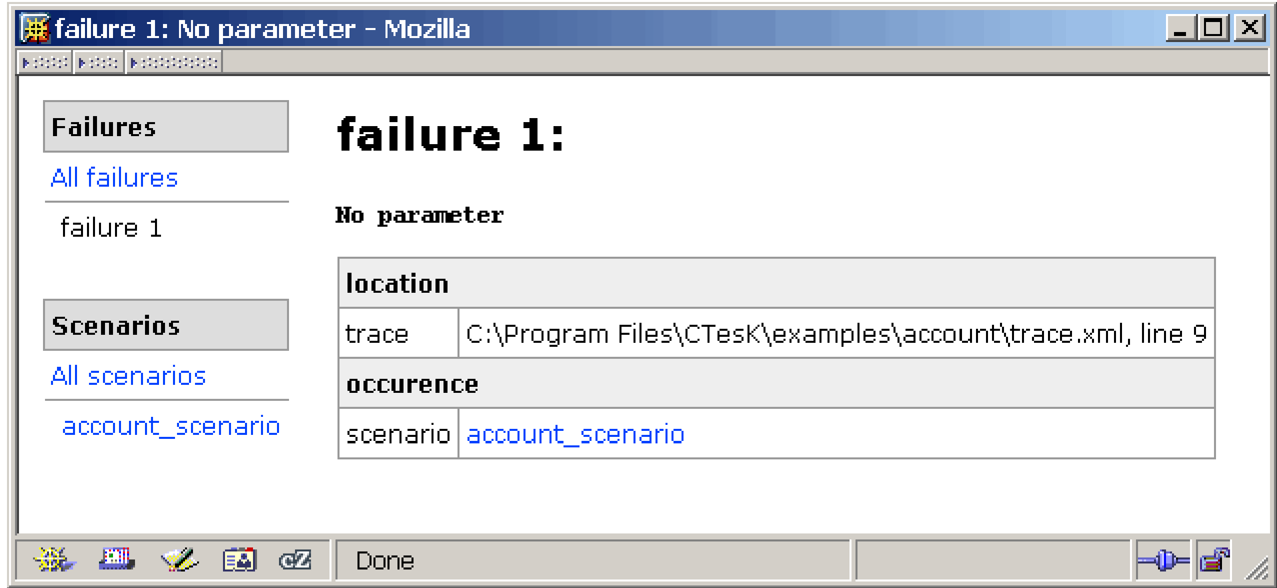


Рисунок 11. Отчет о пользовательской ошибке

Название такой ошибки будет содержать текст, указанный в функции `assertion()`. Контекст позволяет локализовать место появления ошибки: поскольку в контексте не содержатся ни состояние, ни переход, ошибка произошла до начала работы обходчика, т. е. в функции инициализации сценария.

Посмотрим код функции инициализации:

```
static bool account_init (int argc, char **argv) {
  assertion(argc > 1, "No parameter");
  ...
}
```

В данном случае тест был запущен без передачи ему необходимого параметра из командной строки, что и вызвало появление ошибки.

Анализ полноты покрытия

Анализ полноты покрытия, достигнутого в ходе тестирования, осуществляется по отчету о покрытии спецификационных функций. Рассмотрим отчет:

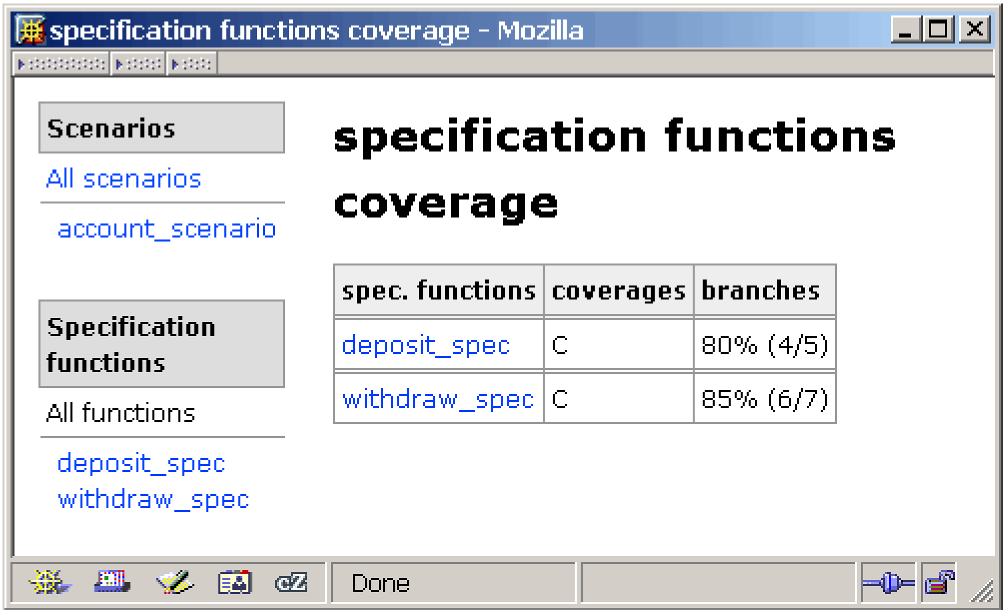


Рисунок 12. Отчет о покрытии в сценарии

В таблице перечислены все протестированные спецификационные функции. Для каждого критерия покрытия каждой функции, в таблице указан процент покрытия ветвей функциональности данного покрытия.

Например, здесь у функции **deposit_spec** задан один критерий покрытия **C**. В нем покрыто **80%** ветвей функциональности (**4** из **5**).

Чтобы узнать более подробную информацию о покрытии, необходимо посмотреть отчет о покрытии этой функции:

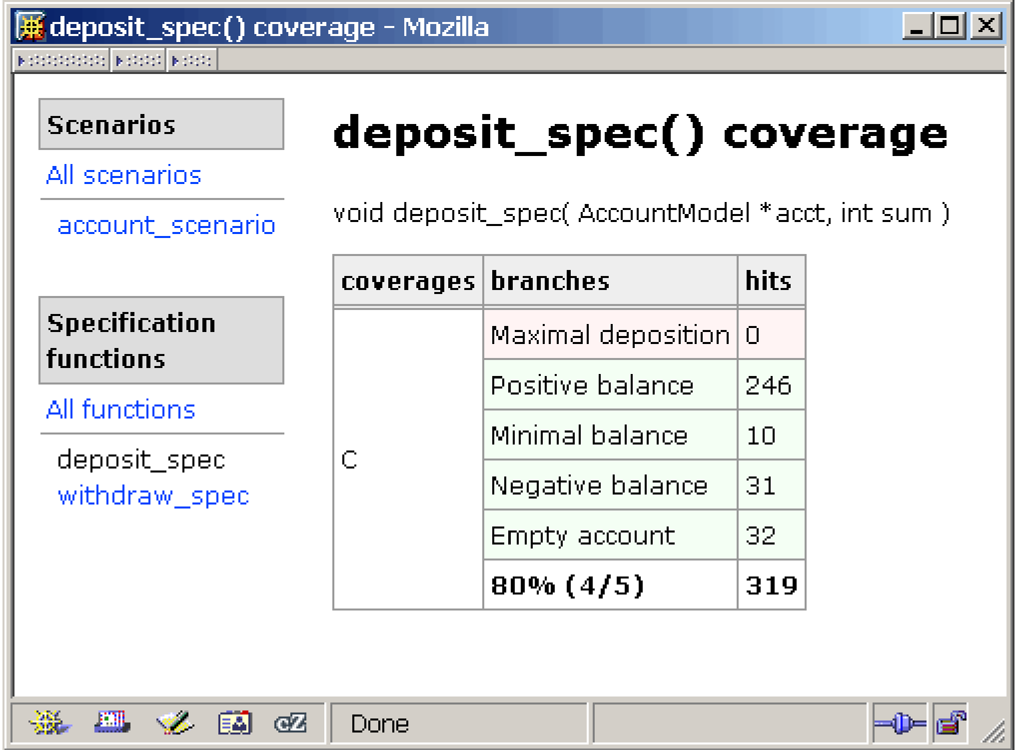


Рисунок 13. Отчет о покрытии функции

Из него видно, что ветвь "**Maximal deposition**" не пройдена ни разу (она подсвечена красным цветом). Остальные ветви пройдены хотя бы по разу (они подсвечены зеленым).

В случае, когда какая-либо ветвь не покрыта, следует либо изменить итерацию параметров спецификационной функции, либо сделать дополнительную сценарную функцию, либо написать отдельный сценарий для дополнительного тестирования данной ветви.

Примеры использования CTesK

Системы с прикладным программным интерфейсом

Рассмотренный в данном разделе подход применим к тестированию систем, имеющих прикладной программный интерфейс (API) — набор функций, с помощью которых прикладная программа может общаться с системой. Типичный пример такой системы — библиотека, содержащая математические функции или функции для работы с некоторыми структурами данных.

В качестве примера использования инструмента CTesK рассматривается процесс тестирования системы, обеспечивающей работу с очередью.

Описание интерфейса целевой системы

Очередь — это контейнер, реализующий семантику *FIFO* (*First In — First Out*). Для очереди определены две основные операции: добавление и удаление элемента, при этом первым из очереди извлекается элемент, который был добавлен в нее раньше других. Элементы передаются очереди через нетипизированные указатели на них.

Очередь реализована в виде односвязного списка, причем первый элемент списка не содержит элемента очереди.

```
struct queue {
  void *item;
  struct queue *next;
};
```

Интерфейс целевой системы состоит из следующих функций:

- `struct queue *create_queue (void)` — конструктор, создающий пустую очередь
- `void delete_queue (struct queue **queue)` — уничтожает очередь и обнуляет указатель на нее
- `int` empty (struct queue *queue) — проверяет, пуста ли очередь
- `void enq (struct queue *queue, void *item)` — добавляет ненулевой элемент item в конец очереди
- `void *deq (struct queue *queue)` — возвращает первый элемент и удаляет его из очереди

Функциональные требования к системе сформулированы следующим образом:

- В очереди могут храниться только указатели, не равные `NULL`.
- Функция `create_queue()` возвращает ненулевой указатель на новую пустую очередь. Этот указатель может быть передан в качестве параметра остальным функциям системы. Очереди, созданные ранее, не изменяются.
- Очередь может быть уничтожена функцией `delete_queue()`. После этого старое значение указателя на уничтоженную очередь не может быть использовано в качестве параметра других функций системы.
- Очередь проверяется на пустоту функцией `empty()`. Ненулевое значение, возвращенное этой функцией, соответствует пустой очереди, нулевое — не пустой.
- Элемент добавляется в очередь при помощи функции `enq()`.
- Элемент, хранящийся в очереди и помещенный туда ранее других, может быть извлечен из очереди функцией `deq()`. Функция возвращает этот элемент и удаляет его из очереди.

Разработка спецификаций

Спецификация состоит из *спецификационных функций*, описывающих поведение функций целевой системы, и *спецификационной модели данных*, содержащей дополнительную информацию, необходимую для описания поведения.

Спецификационная модель данных

Спецификационная модель содержит определение типов и данных, в терминах которых описывается поведение тестируемой системы.

Целевая система, рассматриваемая в данном примере, работает со следующими данными:

- элемент очереди
- сама очередь
- множество созданных очередей
- целочисленное значение, обозначающее пустоту очереди (возвращаемое функцией `empty()`)

Элемент очереди

Элемент очереди – указатель типа `void` *. Из требований к системе следует, что элемент очереди не может быть равным `NULL`. Язык SeC позволяет описывать тип данных с ограниче-

ниями на значения этого типа (*инвариантом*). Для этого используется конструкция `typedef` с ключевым словом **`invariant`**.

```
invariant typedef void *item_t;
```

Данное объявление вводит новый тип с инвариантом `item_t`, структурно эквивалентный типу `void` *. Сам инвариант описывается ниже при помощи следующей конструкции:

```
invariant (item_t item)
{
  return NULL != item;
}
```

Очередь

Очередь — это упорядоченный список содержащихся в ней элементов. Можно было бы воспользоваться реализационной структурой `queue`, однако, во-первых, это усложнит спецификацию, и во-вторых, такая спецификация будет реализационно зависимой. Гораздо удобнее воспользоваться типом `List`, определенным в библиотеке спецификационных типов CTesK[7].

Библиотека спецификационных типов содержит набор предопределенных типов (в том числе и контейнерных), набор стандартных функций работы с данными этих типов (копирование, сравнение и т. п.) и общий механизм управления памятью, занимаемой данными. Данные спецификационных типов всегда хранятся в динамической памяти, а доступ к ним осуществляется через *спецификационную ссылку* — указатель на спецификационный тип.

Тип `List` может хранить только элементы спецификационного типа. При помощи конструкции **`specification`** `typedef` можно создать спецификационный тип, соответствующий некоторому типу языка C.

```
specification typedef item_t Item = {};
```

Для создания объекта спецификационного типа используется библиотечная функция `create()`. Первый параметр этой функции — указатель на *дескриптор типа* создаваемого объекта — структуру, содержащую информацию, необходимую системе для работы с объектами этого типа. Для каждого спецификационного типа определена переменная, содержащая дескриптор типа, и имеющая имя, полученное из имени типа с префиксом `type_` (в данном случае — `type_Item`). Остальные параметры зависят от конкретного типа — в данном случае функции передается значение типа `item_t`. Для удобства можно описать вспомогательную функцию создания объекта типа `Item`.

```
Item *create_item_aux (item_t item)
{
  return create (&type_Item, item);
}
```

Для упрощения работы с моделями очередей опишем две функции — добавления и извлечения элемента.

```
void add_last_aux (List *list, item_t item);
item_t remove_first_aux (List *list);
```

Функция `add_list_aux()` должна создать на основе значения `item` объект типа `Item` и поместить его в конец списка `list`. Для последнего действия используется библиотечная функция `append_List()`.

[7] Более подродную информацию о библиотечных спецификационных типах см. в главе "*Библиотека спецификационных типов*" документа "*CTesK 2.2. Описание языка SeC*".

```
void add_last_aux (List *list, item_t item)
{
  append_List (list, create_item_aux (item));
}
```

Функция `remove_first_aux()` использует библиотечные функции `get_List()`, возвращающую элемент списка по заданному индексу, и `remove_List()`, которая удаляет из списка элемент по заданному индексу. Для того, чтобы по ссылке на объект типа `Item` получить значение типа `item_t`, можно просто разыменовать эту ссылку.

```
item_t remove_first_aux (List *list)
{
  Item *item = get_List (list, 0);
  remove_List (list, 0);
  return *item;
}
```

Множество созданных очередей

Так как очереди, с которыми работает тестируемая система, располагаются в динамической памяти, эта память является состоянием системы. Моделировать всю память невозможно, да и не имеет смысла. Для описания функциональности достаточно знать, что некоторый указатель — это указатель на очередь. Для моделирования набора существующих очередей можно воспользоваться отображением из указателей на реализационные очереди в их модели.

Для работы с отображениями используется библиотечный тип `Map`. Так как во-первых, для описания системы не важны данные, находящиеся по указателю на реализационную очередь, и во-вторых, с точки зрения языка SeC типизированный указатель ссылается на единственное значение соответствующего типа, ключом отображения следует сделать тип `void*`. Для удобства можно описать для этого типа `typedef`-имя: Так же для представления нулевого указателя можно описать соответствующую константу.

```
typedef void *queue_t;

const queue_t null_queue = NULL;
```

При использовании типа `Map` и ключи отображения, и его значения должны быть объектами спецификационных типов. Поэтому следует определить спецификационный тип, соответствующий типу `queue_t`. Для удобства можно также описать функцию создания объекта этого спецификационного типа.

```
specification typedef queue_t Queue;

Queue *create_queue_aux (queue_t queue)
{
  return create (&type_Queue, queue);
}
```

Для упрощения работы с моделью опишем следующие вспомогательные функции, работающие с идентификатором очереди:

- проверка корректности некоторого идентификатора, т. е. существования его в качестве ключа в заданном отображении;
- получение по идентификатору очереди ее модельного представления;
- удаления очереди из отображения;
- получение размера очереди по ее идентификатору.

Функция `exists_queue_aux()` проверяет, что заданный идентификатор очереди является ключом в заданном отображении при помощи библиотечной функции `containsKey_Map()`.

```
bool exists_queue_aux (Map *model_queues, queue_t queue)
{
  return containsKey_Map (model_queues, create_queue_aux (queue));
}
```

Функция `get_queue_aux()` возвращает модельную очередь, соответствующую заданному идентификатору при помощи функции получения значения в отображении по ключу `get_Map()`.

```
List *get_queue_aux (Map *model_queues, queue_t queue)
{
  return get_Map (model_queues, create_queue_aux (queue));
}
```

Для добавления очереди используется функция `add_queue_aux()`, которая вызывает библиотечную функцию `put_Map()`, добавляющую в отображение пару {*ключ, значение*}

```
void add_queue_aux (Map *model_queues, queue_t queue, List *list)
{
  put_Map (model_queues, create_queue_aux (queue), list);
}
```

Функция `remove_queue_aux()` использует библиотечную функцию `remove_Map()` для удаления из отображения ключа и соответствующего значения.

```
void remove_queue_aux (Map *model_queues, queue_t queue)
{
  remove_Map (model_queues, create_queue_aux (queue));
}
```

Для получения количества элементов в очереди функция `size_queue_aux()` использует вышеописанную функцию `get_queue_aux()` и библиотечную функцию `size_List()`, которая возвращает длину списка.

```
int size_queue_aux (Map *model_queues, queue_t queue)
{
  return size_List (get_queue_aux (model_queues, queue));
}
```

Теперь можно описать переменную модельного состояния — `model_queues`. Нулевой указатель не указывает ни на какую очередь, и это ограничение можно внести в инвариант переменной. Переменная с инвариантом описывается как обычная переменная с добавлением ключевого слова **`invariant`**. Сам инвариант описывается при помощи специальной конструкции.

```
invariant Map *model_queues;

invariant (model_queues)
{
  return !exists_queue_aux (model_queues, null_queue);
}
```

При тестировании модельное состояние должно быть создано до вызовов тестируемых функций. В данном случае нужно создать объект типа Map и присвоить ссылку на него переменной `model_queues`. Функция создания объекта типа `Map` принимает на вход два дополнительных параметра — указатели на дескрипторы типов ключей и значений отображения.

```
void init_state_queue (void)
{
  model_queues = create (&type_Map, &type_Queue, &type_List);
}
```

Значение, обозначающее пустоту очереди

В качестве значения, возвращаемого функцией проверки очереди на пустоту (`empty()`), будет использован тип `bool`, имеющий два значения — `true` и `false`.

Спецификация поведения

Поведение тестируемых функций описывается в *спецификационных функциях*. Определение спецификационной функции состоит из трех частей:

- сигнатура функции, аналогичная сигнатуре обычной функции (возвращаемое значение, имя функции и ее аргументы) и содержащая ключевое слово **`specification`**;
- ограничения доступа к глобальным переменным и параметрам;
- тело спецификационной функции.

Ограничения доступа к глобальным переменным бывают трех типов: на чтение, на запись и на изменение. Если поведение функции зависит от значения переменной, но это значение не изменяется в результате работы функции, то она осуществляет доступ на чтение к этой переменной. Если поведение функции не зависит от значения переменной, но после вызова функции эта переменная будет содержать результат ее работы, то функция осуществляет доступ на запись. В случае ограничения доступа на изменение поведение функции зависит от значения этой переменной, и это значение может быть изменено в результате работы функции.

Поведение тестируемой функции описывается в теле спецификационной функции в виде пред- и постусловия. *Предусловие* описывает область определения целевой функции. *Постусловие* описывает само поведение функции в терминах зависимости между значениями параметров и глобальных переменных до и после вызова функции. Кроме того, тело спецификационной функции может содержать описания критериев покрытия. Критерий покрытия определяет разбиение входных данных на подобласти, на которых поведение функции существенно отличается. Каждая из таких подобластей называется *ветвью функциональности*.

Функция создания очереди

Сигнатура спецификационной функции, описывающей поведение функции `create_queue()`, выглядит следующим образом:

```
specification void create_queue_spec (void)
```

Функция создания очереди изменяет состояние системы, добавляя новую очередь, следовательно ограничение доступа будет на изменение переменной `model_queues`.

```
specification void create_queue_spec (void)
    updates model_queues
```

В постусловии функции нужно формально описать следующее требование:

Функция `create_queue()` возвращает ненулевой указатель на новую пустую очередь. Этот указатель может быть передан в качестве параметра остальным функциям системы. Очереди, созданные ранее, не изменяются.

В постусловии должна быть проверка на то, что функция вернула идентификатор очереди, не равный `null_queue` (1), соответствующий пустой очереди (2), и не соответствовавший никакой очереди непосредственно перед вызовом функции (3).

Для доступа в постусловии к значению, возвращенному функцией, используется имя функции, в данном случае — `create_queue_spec`. Для проверки третьего утверждения требуется доступ к отображению `model_queues`, каким оно было до вызова функции — к его *предзначению*. Однако в постусловии глобальные переменные и изменяемые аргументы имеют *по-*

стзначения, т. е. значения, полученные в результате вызова функции. Для доступа к презначению некоторой переменной или выражения используется оператор `@`. В данном случае требуется получить презначение отображения:

```
if (  null_queue == create_queue_spec
   || !exists_queue_aux (model_queues, create_queue_spec)
   || 0 != size_queue_aux (model_queues, create_queue_spec)
   || exists_queue_aux (@model_queues, create_queue_spec)
   )
  return false;
```

Теперь следует проверить, что остальные очереди остались неизменными. Для этого можно создать копию текущего отображения `model_queues` (саму переменную изменять нельзя, так как спецификационная функция не должна иметь побочного эффекта), удалить из него только что созданную очередь, и сравнить с состоянием системы до вызова функции. Сравнение двух объектов спецификационного типа выполняется при помощи библиотечной функции equals.

```
Map *tmp_map = clone (model_queues);
...
remove_queue_aux (tmp_map, create_queue_spec);
return equals (tmp_map, @ model_queues);
```

Само постусловие описывается в теле спецификационной функции в составном операторе, предваренном ключевым словом **`post`**.

```
specification queue_t create_queue_spec (void)
    updates model_queues
{
  post {
    Map *tmp_map = clone (model_queues);

    if (  null_queue == create_queue_spec
       || !exists_queue_aux (model_queues, create_queue_spec)
       || 0 != size_queue_aux (model_queues, create_queue_spec)
       || exists_queue_aux ( @ model_queues
                           , create_queue_spec)
                           )
       )
      return false;

    remove_queue_aux (tmp_map, create_queue_spec);
    return equals (tmp_map, @ model_queues);
  }
}
```

Функция уничтожения очереди

Сигнатура и ограничения доступа функции выглядят так:

```
specification void delete_queue_spec (queue_t *pqueue)
    updates model_queues
```

Ниже приводятся требования к функции уничтожения очереди:

Функция delete_queue принимает на вход указатель на указатель на очередь. Если значение по указателю равно `NULL`*, функция не делает ничего. В противном случае это значение должно указывать на существующую очередь. Соответствующая очередь уничтожается. По указателю-параметру записывается значение* `NULL`*. После этого старое значение указателя на уничтоженную очередь не может быть использовано в качестве параметра других функций системы.*

Из требований следует, что параметр функции должен указывать либо на значение `NULL`, либо на указатель на существующую очередь. Такие требования формулируются в предусловии спецификационных функций. Предусловие описывается в теле спецификационной функции в блоке, помеченном ключевым словом **`pre`**.

```
pre {
  return null_queue == *pqueue
      || exists_queue_aux (model_queues, *pqueue);
}
```

В требованиях к функции явно выделено две ветви функциональности в зависимости от значения по указателю `pqueue`. Эти ветви следует описать в *критерии покрытия*. Критерий покрытия описывается в составном операторе, предваренном ключевым словом **`coverage`** и своим именем:

```
coverage C { ... }
```

Для каждой ветви функциональности в этом блоке должен быть оператор `return` с парой значений, окруженной фигурными скобками. Первое значение — идентификатор ветви функциональности, второе — строка, содержащая описание ветви.

```
coverage C {
  if (null_queue == *pqueue) return {null, "null queue"};
  else return {not_null, "non-null queue"};
}
```

В постусловии для двух этих ветвей должны быть сформулированы различные проверки. Для того, чтобы не повторять вычисления, сделанные при описании критерия покрытия, используется псевдофункция **`coverage`**, которой передается идентификатор покрытия. Возвращаемое значение — идентификатор ветви функциональности, которая была достигнута в данном покрытии.

```
if (coverage (C) == null) {
  ...
} else {
  ...
}
```

В случае нулевого указателя следует проверить, что значение отображения `model_queues` и значение по указателю pqueue не изменились.

```
return @*pqueue == *pqueue
    && equals (@model_queues, model_queues);
```

В противном случае значение по указателю должно быть равно `null_queue`. Для проверки того, что очередь удалилась, можно создать копию отображения до вызова функции, удалить из него соответствующие ключ и значение и сравнить с текущим значением.

```
Map *tmp_map = @clone (model_queues);

remove_queue_aux (tmp_map, @*pqueue);
return null_queue == *pqueue
    && equals (tmp_map, model_queues);
```

В итоге получается следующая спецификационная функция:

```
specification void delete_queue_spec (queue_t *pqueue)
    updates model_queues
{
  pre {
    return null_queue == *pqueue
        || exists_queue_aux (model_queues, *pqueue);
  }
  coverage C {
    if (null_queue == *pqueue) return {null, "null queue"};
    else return {not_null, "non-null queue"};
  }
  post {
    if (coverage (C) == null) {
      return @*pqueue == *pqueue
          && equals (@model_queues, model_queues);
    } else {
      Map *tmp_map = @clone (model_queues);

      remove_queue_aux (tmp_map, @*pqueue);
      return null_queue == *pqueue
          && equals (tmp_map, model_queues);
    }
  }
}
```

Функция проверки очереди на пустоту

Сигнатура спецификационной функции, описывающей поведение функции проверки очереди на пустоту, приведена ниже.

```
specification bool empty_spec (queue_t queue)
```

Функция проверки очереди на пустоту не должна изменять очередь, однако ее поведение зависит от состояния. Поэтому она осуществляет доступ к переменной `model_queues` на чтение.

```
specification bool empty_spec (queue_t queue)
    reads model_queues
```

Функция должна соответствовать перечисленным ниже требованиям.

Функция empty принимает на вход указатель на существующую очередь и возвращает ненулевое значение, если очередь пуста, и ноль в противном случае.

Функция имеет предусловие, проверяющее, что очередь существует:

```
pre {
  return null_queue != queue
      && exists_queue_aux (model_queues, queue);
}
```

Для этой функции можно выделить две ветви функциональности, соответствующих пустой и непустой очереди.

```
coverage C {
  if (0 == size_queue_aux (model_queues, queue)
    return {empty, "empty queue"};
  else
    return {not_empty, "non-empty queue"};
}
```

В постусловии должно быть сформулировано следующее утверждение: возвращаемое значение должно быть истинным, если очередь пуста, и ложным в противном случае.

```
post {
  return (coverage (C) == empty) == empty_spec;
}
```

Целиком спецификационная функция приведена ниже.

```
specification bool empty_spec (queue_t queue)
    reads model_queues
{
  pre {
    return null_queue != queue
        && exists_queue_aux (model_queues, queue);
  }
  coverage C {
    if (0 == size_queue_aux (model_queues, queue))
      return {empty, "empty queue"};
    else
      return {not_empty, "non-empty queue"};
  }
  post {
    return empty_spec == (coverage (C) == empty);
  }
}
```

Функция добавления элемента

Функция enq принимает на вход указатель на существующую очередь и ненулевой указатель на помещаемый элемент. Очередь не перестает существовать. Элемент добавляется в соответствующую очередь. Остальные очереди не изменяются.

Функция изменяет очередь, следовательно она осуществляет доступ на изменение к переменной `model_queues`.

```
specification void enq_spec (queue_t queue, item_t item)
    updates model_queue
```

В соответствии с требованиями, у функции есть предусловие: очередь должна существовать, и элемент не должен быть нулевым. Так как параметр `item` имеет тип `item_t`, для которого определен инвариант (неравенство значению `NULL`), то это ограничение будет проверено автоматически, и предусловие функции — проверка очереди на существование:

```
pre {
  return null_queue != queue
      && exists_queue_aux (model_queues, queue);
}
```

Для этой функции можно определить критерий покрытия, аналогичный критерию покрытия функции `empty_spec()`.

```
coverage C {
  if (0 == size_queue_aux (model_queues, queue))
    return {empty, "empty queue"};
  else
    return {not_empty, "non-empty queue"};
}
```

Элементы в модели очереди расположены в соответствии с порядком их помещения в очередь — элементы с меньшим индексом были помещены в очередь раньше. Следовательно, новый элемент должен быть добавлен в конец списка, а остальные элементы должны остаться прежними. Чтобы проверить, что остальные очереди не изменились, можно в копию презначения `model_queues` поместить полученную очередь (`tmp_list`) и сравнить с постзначением `model_queues`.

```
post {
  Map *tmp_map = @clone (model_queues);
  List *tmp_list = @clone (get_queue_aux (model_queues, queue));

  add_last_queue (tmp_list, item);
  add_queue_aux (tmp_map, queue, tmp_list);

  return equals (tmp_list, get_queue_aux (model_queues, queue))
      && equals (tmp_map, model_queues);
}
```

Теперь можно записать целиком спецификацию функции `enq()`.

```
specification void enq_spec (queue_t queue, item_t item)
    updates model_queues
{
  pre {
    return null_queue != queue
        && exists_queue_aux (model_queues, queue);
  }
  coverage C {
    if (0 == size_queue_aux (model_queues, queue))
      return {empty, "empty queue"};
    else
      return {not_empty, "non-empty queue"};
  }
  post {
    Map *tmp_map = @clone (model_queues);
    List *tmp_list =
      @clone (get_queue_aux (model_queues, queue));

    add_last_queue (tmp_list, item);
    add_queue_aux (tmp_map, queue, tmp_list);

    return equals (tmp_list, get_queue_aux (model_queues, queue))
        && equals (tmp_map, model_queues);
  }
}
```

Функция извлечения элемента

Функция deq принимает на вход указатель на существующую непустую очередь. Очередь не перестает существовать. Функция удаляет из очереди элемен, добавленный в очередь ранее остальных находящихся в ней элементов.Функция возвращает удаленный элемент. Остальные очереди не изменяются.

В критерии покрытия можно выделить две ветви функциональности — первая соответствует извлечению последнего элемента, вторая — остальным случаям.

Поступая аналогичным с предыдущей функцией образом, можно записать следующую спецификационную функцию.

```
specification item_t deq_spec (queue_t queue)
    updates model_queues
{
  pre {
    return null_queue != queue
        && exists_queue_aux (model_queues, queue)
        && 0 < size_queue_aux (model_queues, queue);
  }
  coverage C {
    if (1 == size_queue_aux (model_queues, queue))
      return {last, "last element"};
    else
      return {not_last, "non-last element"};
  }
  post {
    Map *tmp_map = @clone (model_queues);
    List *tmp_list =
      @clone (get_queue_aux (model_queues, queue));
    item_t tmp_item = remove_first_aux (tmp_list);

    add_queue_aux (tmp_map, queue, tmp_list);

    return deq_spec == tmp_item
        && equals (tmp_list, get_queue_aux (model_queues, queue))
        && equals (tmp_map, model_queues);
  }
}
```

В CTesK имеется возможность группировать спецификационные функции в подсистемы. Эта информация используется при генерации отчетов о проведенном тестировании. Для того чтобы сгруппировать разработанные спецификационные функции в подсистему **queue**, необходимо в начале файла, содержащего исходный код спецификации поместить следующую строчку:

```
#pragma SEC subsystem queue "queue"
```

Строковый литерал "queue" будет использоваться для отображения названия подсистемы в отчетах о проведенном тестировании. В случае, если он совпадает с идентификатором подсистемы, как в нашем случае, его можно опустить. Между идентификаторами и названиями подсистем должно существовать взаимнооднозначное соответствие.

Полностью спецификация приведена в “*Приложении А*”. Спецификация содержится в двух файлах — **queue_specification.seh**, который содержит декларации типов, переменных и функций, и **queue_specification.sec**, содержащий их определения. Эти файлы можно также найти в дистрибутиве системы CTesK в каталоге **examples/queue**.

Разработка медиаторных функций

Медиаторные функции служат для установки соответствия между реализационными функциями и соответствующими им спецификационными. В общем случае спецификация может использоваться для тестирования различных реализаций систем, имеющих одинаковую функциональность. В спецификации не содержится никаких предположений о том, как устроена реализация. Например, разработанная спецификация подходит для системы, которая хранит элементы очереди в массиве, а не в виде односвязного списка. Для тестирования такой системы достаточно написать другие медиаторы.

Каждая медиаторная функция должна вызывать соответствующую реализационную функцию и привести модельное состояние в соответствие с результатами вызова. В случае рассматриваемой системы на основе данного идентификатора очереди (значения типа

queue_t), который фактически является указателем на реализационную очередь, можно построить модельное представление очереди. Для этого следует пройти по односвязному списку реализационной очереди (игнорируя первый “ложный” элемент), добавляя в объект типа List соответствующие элементы. Для создания объекта спецификационного типа List функции create() следует передать указатель на дескриптор типа type_List и указатель на дескриптор типа элементов списка (type_Item).

```
List *queue_to_list (queue_t queue)
{
  struct queue *q;
  List *model;

  if (null_queue == queue) return NULL;

  q = ((struct queue *)queue)->next;
  model = create (&type_List, &type_Item);

  while (NULL != q) {
    add_last_aux (model, q->item);
    q = q->next;
  }

  return model;
}
```

Функцию queue_to_list() следует вызвать для всех идентификаторов очередей — ключей отображения model_queues. Для перебора ключей отображения используется функция key_Map(). Если эта функция вызывается в цикле для некоторого отображения размером *size* со значениями индекса от 0 до size - 1, то гарантируется, что будут перебраны все ключи отображения (в некотором произвольном порядке).

```
void queue_map_state_up (void)
{
  int qi;
  Map *old_model_queues = model_queues;

  model_queues = create (&type_Map, &type_Queue, &type_List);

  for (qi = 0; qi < size_Map (old_model_queues); qi++) {
    Queue *q = key_Map (old_model_queues, qi);
    put_Map (model_queues, q, queue_to_list (*q));
  }
}
```

Определение медиаторной функции состоит из следующих частей:

- ключевое слово **mediator**, имя медиатора и ключевое слово **for**;
- сигнатура соответствующей спецификационной функции и ее ограничения доступа;
- тело медиаторной функции.

Медиаторная функция для функции создания очереди должна выглядить следующим образом:

```
mediator create_queue_media for
specification queue_t create_queue_spec (void)
    writes model_queue
{
  ...
}
```

Тело медиаторной функции состоит из двух блоков — блока воздействия, в котором осуществляется вызов тестируемой функции, и блока синхронизации, который в соответствии с воздействием изменяет модельное состояние.

В блоке воздействия функции `create_queue_media()` должны быть вызов функции `create_queue()` и возвращение модельного представления результата.

```
call {
  return (queue_t)create_queue ();
}
```

Блок синхронизации состояния должен содержать вызов функции `queue_map_state_up()`, которая выполнит преобразование состояния для ранее существующих очередей, и вызов `add_queue_aux()`, который добавит в состояние модельное представление вновь созданной очереди.

```
state {
  queue_map_state_up ();
  add_queue_aux ( model_queues
                , create_queue_spec
                , queue_to_list (create_queue_spec)
                );
}
```

Медиаторная функция удаления очереди выглядит следующим образом:

```
mediator delete_queue_media for
specification void delete_queue_spec (queue_t *queue)
    updates model_queues
{
  ...
}
```

Ее блок воздействия — вызов функции `delete_queue()`.

```
call {
  delete_queue ((struct queue **)queue);
}
```

В блоке синхронизации следует удалить из отображения соответствующую очередь и вызвать функцию вычисления состояния. Так как значение `*queue` изменяется при вызове функции, его надо сохранить в локальной переменной.

```
queue_t tmp_queue = *queue;
call { ... }
state {
  remove_queue_aux (model_queues, tmp_queue);
  queue_map_state_up ();
}
```

Блок воздействия медиаторной функции проверки очереди на пустоту должен содержать преобразование возвращаемого значения из типа `int` в тип `bool`.

```
mediator empty_media for
specification bool empty_spec (queue_t queue)
    reads model_queues
{
  call {
    return empty ((struct queue *)queue) ? true : false;
  }
  state {
    queue_map_state_up ();
  }
}
```

Для функций помещения и извлечения элемента в блоках воздействий возвращаются результаты вызовов соответствующих реализационных функций, а блоки синхронизации состоят только из вызова функции `queue_map_state_up()`.

```
mediator enq_media for
specification void enq_spec(queue_t queue, item_t item)
    updates model_queues
{
  call {
    enq ((struct queue *)queue, item);
  }
  state {
    queue_map_state_up ();
  }
}

mediator deq_media for
specification item_t deq_spec(queue_t queue)
  updates model_queues
{
  call {
    return deq ((struct queue *)queue);
  }
  state {
    queue_map_state_up ();
  }
}
```

Полностью определение медиаторных функций приведено в “*Приложении А*”. Как и спецификация, описание медиаторных функций, работающих с открытым состоянием, содержится в двух файлах – **queue_media.seh**, который содержит декларации медиаторных функций, и **queue_media.sec**, содержащий их определения и определения вспомогательных функций и данных. Эти файлы можно также найти в дистрибутиве системы CTesK в каталоге **examples/queue**.

Разработка тестового сценария

Для того, чтобы обеспечить полноту тестирования, поведение целевых функций должно быть проверено в различных состояниях. Например, функции проверки на пустоту, добавления и извлечения элемента толжны тестироваться при различных длинах очереди. Для этого должна быть разработана тестовая последовательность — последовательность вызовов целевых функций с различными значениями параметров.

CTesK автоматически строит тестовую последовательность на основе *тестового сценария*, разрабатываемого вручную.

В тестовом сценарии определяются следующие функции:

- функция инициализации сценария;
- функция завершения сценария;
- функция вычисления состояния сценария;
- сценарные функции, определяющие перебор параметров тестируемых функций и их вызовы.

В данном примере будет рассмотрен сценарий для тестирования трех реализационных функций: `enq()`, `deq()` и `empty()`.

Для этого понадобятся определения дополнительных данных. Во-первых, следует ограничить размер тестируемой очереди, чтобы тестирование не длилось “бесконечно”. Во-вторых, нужно определить данные, которые будут помещаться в очередь.

Для ограничения длины очереди определяется целочисленная переменная `queue_max_size`, а для элементов очереди — переменная, определяющая их количество `queue_items_num`, и массив `queue_items`.

```
int queue_max_size = 10;

int queue_items_num = 20;
item_t *queue_items;
```

В третьих, следует описать переменную для хранения идентификатора единственной очереди, на которой будет происходить тестирование функций.

```
queue_t queue;
```

Функция инициализации сценария должна выделить память под эти данные и заполнить ее нужными значениями. Этой функции передаются параметры командной строки, чтобы она могла их обработать и извлечь необходимые сведения. Кроме того, тестируемая очередь должна быть создана, а ее модельное представление должно быть синхронизировано с ней. Для этого следует вызвать спецификационную функцию создания очереди. Тогда функцию инициализации сценария можно записать следующим образом:

```
bool queue_scenario_init (int argc, char **argv)
{
  int i;

  if (argc > 1) queue_max_size = atoi (argv[1]);
  if (argc > 2) queue_items_num = atoi (argv[2]);

  queue_items =
   (item_t *)calloc (queue_items_num, sizeof (item_t));
  for (i = 0; i < queue_items_num; i++)
    queue_items[i] = malloc (i);

  queue = create_queue_spec ();

  return true;
}
```

Функция завершения сценария должна освободить ресурсы, выделенные сценарием.

```
void queue_scenario_finish ()
{
  int i;

  delete_queue_spec (&queue);

  for (i = 0; i < queue_items_num; i++)
    free (queue_items[i]);
  free (queue_items);
}
```

Если определять сценарное состояние как последовательность элементов, находящихся в очереди, то таких состояний будет слишком много. С другой стороны, достаточно протестировать функции при различных длинах очереди. Таким образом сценарное состояние определяется как целое число, равное текущей длине очереди. Для того, чтобы функции протестировались во всех таких состояниях, нужно определить функцию вычисления текущего состояния и перебор параметров для тестируемых функций. Используя эти функции тестовая система CTesK сама обеспечит тестирование в различных состояниях.

Функция вычисления состояния должна возвращать объект спецификационного типа. Для целочисленных значений библиотека спецификационных типов CTesK предоставляет тип `Integer`. Для создания объекта этого типа нужно передать функции `create()` указатель на дескриптор типа `type_Integer` и значение, инициализирующее этот объект – текущее число элементов в очереди.

```
Object *queue_scenario_state ()
{
  return create ( &type_Integer
                , size_queue_aux (model_queues, queue)
                );
}
```

Теперь следует написать сценарные функции для каждой тестируемой функции системы. В простейшем случае сценарная функция осуществляет единственное *тестовое воздействие* на целевую систему в каждом сценарном состоянии. Тестовое воздействие описывается как вызов спецификационной функции, описывающей поведение тестируемой функции. Сценарная функция не имеет параметров и возвращает значение типа `bool`. В рамках этого примера сценарные функции всегда будут возвращать `true`, как признак успешной работы.

Сценарная функция `empty_scen()` состоит только из вызова функции `empty_spec()` с идентификатором тестируемой очереди в качестве параметра и возврата истинного значения.

```
scenario bool empty_scen ()
{
  empty_spec (queue);
  return true;
}
```

Функции `enq_spec()` передается параметр — элемент, который требуется поместить в очередь. Для этого заготовлен массив `queue_items`, каждый из элементов которого нужно в каждом состоянии поместить в очередь. Для того, чтобы вызвать функцию `enq_spec()` в каждом состоянии со значениями параметра `item` по всем индексам в массиве `queue_items`, следует воспользоваться конструкцией языка SeC **`iterate`**.

```
iterate (int i = 0; i < queue_items_num; i++; )
  enq_spec (queue, queue_items[i]);
```

Цикл **`iterate`** синтаксически похож на цикл `for` языка C. Первое выражение заголовка цикла (`int i = 0`) — это объявление и инициализация *итерационной переменной*. У сценарной функции для каждого сценарного состояния существует отдельный экземпляр итерационной переменной. Второе выражение (`i < queue_items_num`) — условие продолжения выполнения цикла. Третье выражение (`i++`) — вычисление следующего значения итерационной переменной. У цикла **`iterate`** есть еще четвертое выражение, отсутствующее в данном случае — это условие фильтрации. При невыполнении этого условия на очередном витке тело цикла не выполняется.

При каждом выполнении сценарной функции в некотором состоянии тело цикла **`iterate`** выполняется с очередным значением итерационной переменной, соответствующей данному состоянию. Тело данного цикла будет выполнено в каждом состоянии сценария со значениями переменной `i`, равными `0`, `1`, ..., `queue_items_num - 1`.

Чтобы ограничить длину очереди, при достижении максимальной ее длины функция добавления элемента вызываться не должна. Для этого в сценарной функции выполняется проверка:

```
if (size_queue_aux (model_queues, queue) != queue_max_size)
  ...
```

Полностью сценарная функция приведена ниже.

```
scenario bool enq_scen ()
{
  if (size_queue_aux (model_queues, queue) != queue_max_size)
    iterate (int i = 0; i < queue_items_num; i++; )
      enq_spec (queue, queue_items[i]);
  return true;
}
```

Функция извлечения элемента из очереди не имеет параметров, которые нужно перебирать, однако для нее определено предусловие, требующее, чтобы очередь была непустой. Исходя из этого, сценарную функцию можно определить так:

```
scenario bool deq_scen ()
{
  if (size_queue_aux (model_queues, queue) != 0)
    deq_spec (queue);
  return true;
}
```

Теперь следует объявить *сценарную переменную* и проинициализировать ее описанными функциями. Сценарная переменная имеет тип `dfsm` и инициализируется в стиле стандарта C99 — перечислением имен полей и их значений. Поля, соответствующие функциям инициализации и завершения сценария, называются соответственно `init` и `finish`. Функцией вычисления обобщенного состояния инициализируется поле `getState`. Поле с именем `actions` — массив, в котором содержится список сценарных функций, завершающийся нулевым указателем.

```
scenario dfsm queue_scenario =
{
  .init     = queue_scenario_init,
  .finish   = queue_scenario_finish,
  .getState = queue_scenario_state,
  .actions  = {empty_scen, enq_scen, deq_scen, NULL}
};
```

Полностью заголовочный и основной файлы, содержащие тестовый сценарий (**queue_scen.seh** и **queue_scen.sec**), приведены в “*Приложении A*”. Эти файлы можно также найти в дистрибутиве системы CTesK в каталоге **examples/queue**.

Главная функция теста

Теперь следует описать функцию main, которая запустит выполнение тестового сценария.

Перед запуском самого сценария должна быть обеспечена установка медиаторов для спецификационных функций. Для этого используются функции, автоматически сгенерированные из спецификационных.

```
set_mediator_create_queue_spec (create_queue_media);
set_mediator_delete_queue_spec (delete_queue_media);
set_mediator_empty_spec (empty_media);
set_mediator_enq_spec (enq_media);
set_mediator_deq_spec (deq_media);
```

Так же следует создать модельное состояние — для этого была написана функция `init_state_queue()`.

```
init_state_queue ();
```

Для запуска тестового сценария следует вызвать функцию с именем, совпадающим со сценарной переменной (`queue_scenario` в данном случае), передав ей параметры командной строки.

```
queue_scenario (argc, argv);
```

Вот как выглядит функция main целиком:

```
void main (int argc, char **argv)
{
  set_mediator_create_queue_spec (create_queue_media);
  set_mediator_delete_queue_spec (delete_queue_media);
  set_mediator_empty_spec (empty_media);
  set_mediator_enq_spec (enq_media);
  set_mediator_deq_spec (deq_media);

  queue_scenario (argc, argv);
}
```

Сборка и запуск теста

Для того, чтобы получить исполняемый тест, разработанные файлы следует оттранслировать в язык C, полученные файлы и файл, содержащий реализацию, откомпилировать и собрать в исполнимый модуль с библиотеками CTesK.

Полученный исполнимый тест может сбрасывать отчет о своем выполнении для последующего анализа. Для того, чтобы трасса попадала на консоль, тесту следует передать параметр **–tc**. При указании параметра **–t** с последующим именем файла трасса теста попадет в указанный файл.

На платформе Linux трансляция производится при помощи следующих команд:

% sec.sh queue_spec.sec queue_spec.c
% sec.sh queue_media.sec queue_media.c
% sec.sh queue_scen.sec queue_scen.c
% sec.sh queue_main.sec queue_main.c

Компиляция выполняется такими командами:

% gcc -c queue_spec.c
% gcc -c queue_media.c
% gcc -c queue_scen.c
% gcc -c queue_main.c
% gcc –c queue.c

Сборка теста:

% gcc –o queue_test –L$CTESK_HOME/lib/linux ↵
queue_spec.o queue_media.o queue_scen.o queue_main.o ↵
queue.o -latl –lts –ltracer –lutils

Запуск полученного теста:

% queue_test –tc –t queue.trace 5 10

Генерация отчета и анализ результатов

Тест сбрасывает трассу о своем выполнении в формате XML. Для того, чтобы извлечь из трассы информацию о тестовом покрытии и найденных несоответствиях между спецификацией и реализацией, используется *анализатор трассы*. Анализатору трассы нужно указать каталог, в который требуется сгенерировать HTML-файлы с тестовым отчетом, и файл, содержащий анализируемую трассу.

На платформе Linux:

% ctesk-rg.sh –d report queue.trace

Для просмотра тестового отчета надо вызвать HTML-браузер и открыть в нем файл **index.html** из указанного каталога.

На платформе Linux:

% mozilla report/index.html

В появившемся окне браузера будет отображен отчет обо всех сценариях, выполнявшихся в данном тесте (Рисунок 14).

Рисунок 14. Общий вид тестового отчета.

Следуя ссылке `queue_scenario`, можно получить информацию об обходе состояний тестируемой очереди (Рисунок 15). Первая таблица содержит информацию о сценариях, выполнявшихся в рамках этого теста — имя, количество состояний и количество переходов между состояниями. Далее следует детальное описание переходов для каждого сценария. Для перехода описывается начальное состояние, сценарная функция, при помощи которой выполнялся переход, значения итерационных переменных, конечное состояние перехода, и количество выполнений этого перехода за время работы сценария.

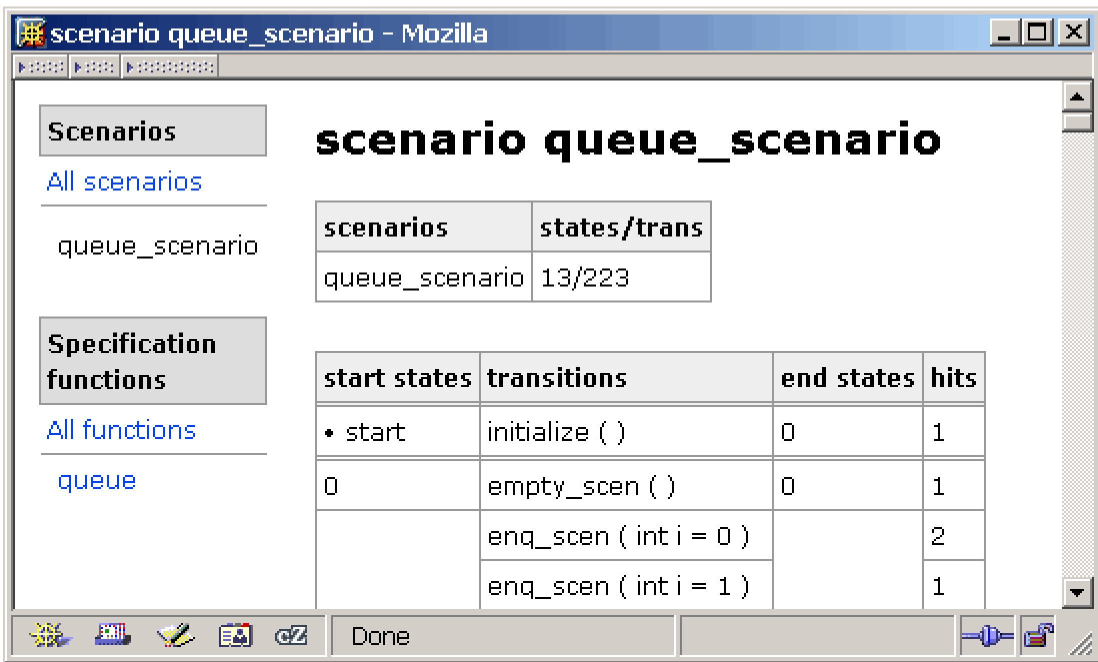


Рисунок 15. Отчет о работе тестового сценария.

По ссылке **All functions** находится таблица, содержащая информацию о покрытии тестируемых подсистем (Рисунок 16). В первой колонке содержатся имена подсистем. Для каждой подсистемы во второй колонке содержатся критерии покрытия, определенные в спецификационных функциях этой подсистемы. Для каждого критерия покрытия функции в колонке **branches** содержится процент покрытия ветвей и количество попаданий в данную ветвь.

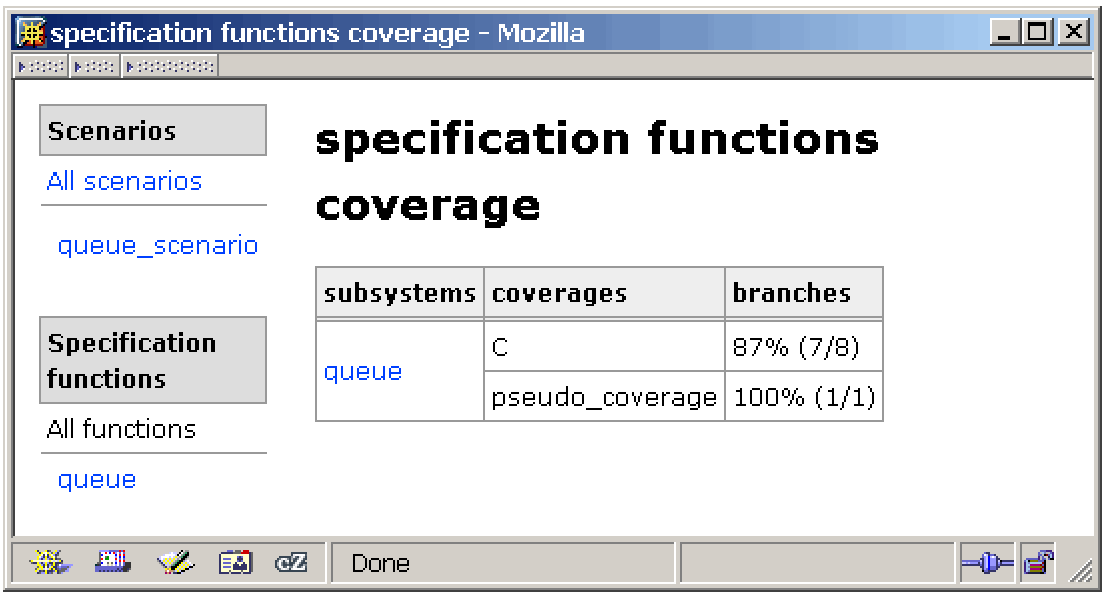


Рисунок 16. Отчет о покрытии подсистем.

По ссылке **queue** находится таблица, содержащая информацию о покрытии спецификационных функций подсистемы **queue**. В первой колонке содержатся имена спецификационных функций. Для каждой функции во второй колонке содержатся критерии покрытия, определенные для нее. Для каждого критерия покрытия функции в колонке **branches** содержится процент покрытия ветвей и количество попаданий в данную ветвь.

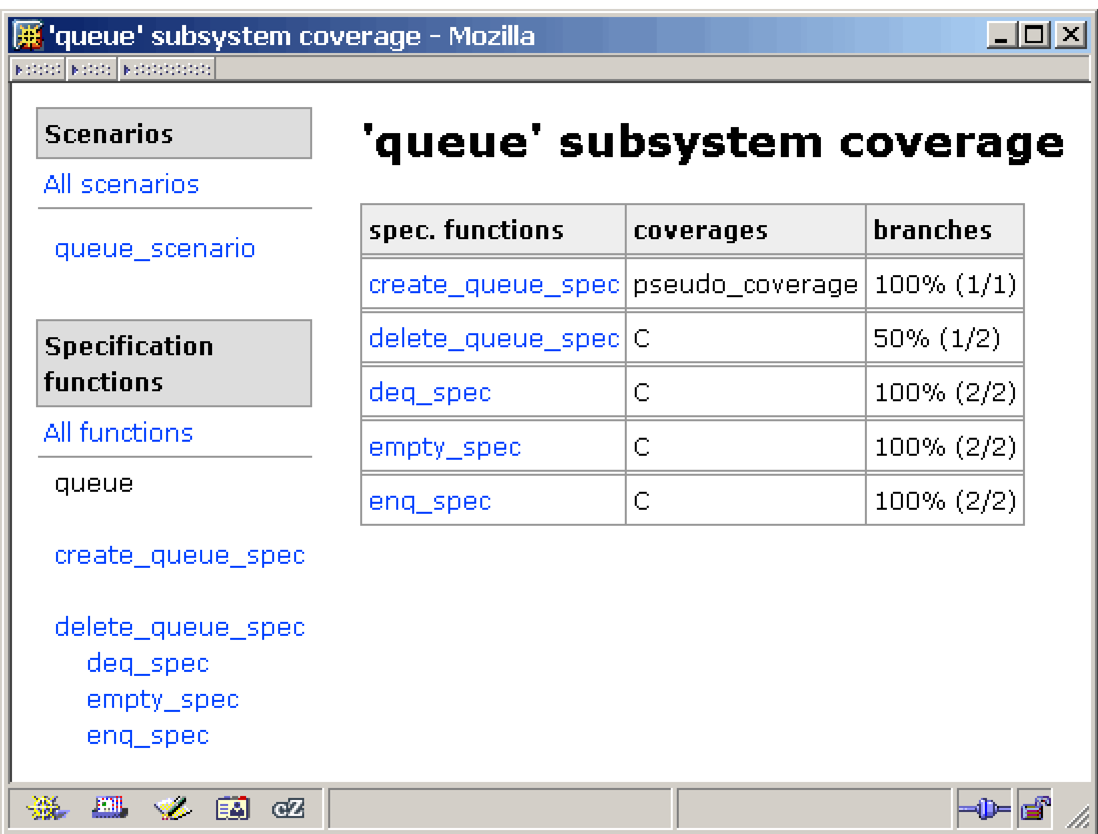


Рисунок 17. Отчет о покрытии спецификационных функций.

Более детальную информацию о покрытии функции можно получить, кликнув на ее имя (Рисунок 18). Первая колонка отображенной таблицы содержит имена покрытий, вторая — имена ветвей функциональности в каждом критерии покрытия, и третья — количество попаданий в эти ветви. Последняя строка таблицы, соответствующая критерию покрытия, содержит информацию о выполнении данного критерия — процент покрытия по ветвям и общее количество вызовов функции.

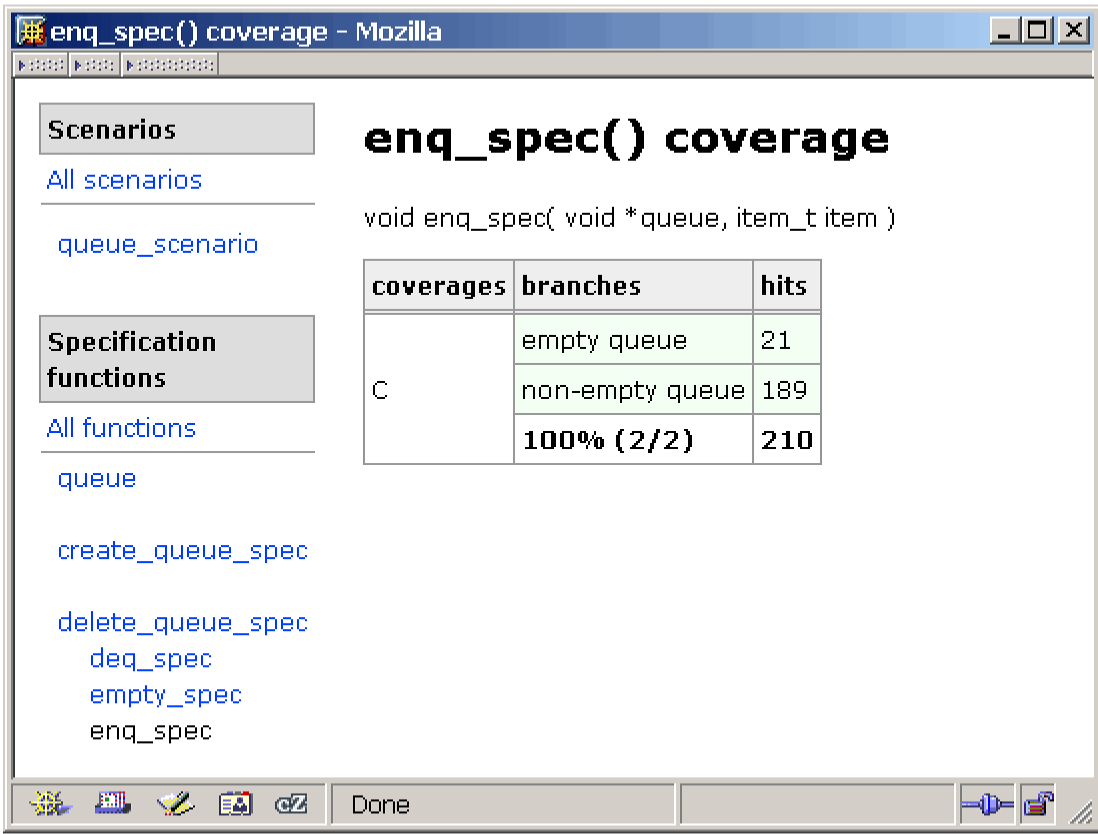


Рисунок 18. Детальный отчет о покрытии спецификационной функции.

Ошибок данным тестом найдено не было, поэтому в отчете информации о них нет. Для демонстрации отчета об обнаруженных несоответствиях можно изменить реализацию. В качестве примера изменим функцию `enq()` таким образом, чтобы она помещала элемент не в конец списка, а в его начало.

```
...

void enq (struct queue *queue, void *item)
{
  struct queue *q = queue->next;
  queue = queue->next =
    (struct queue *)malloc (sizeof (struct queue));
  queue->item = item;
  queue->next = q;
}

...
```

Теперь следует откомпилировать реализационный файл **queue.c**, пересобрать исполнимый модуль, запустить тест и перегенерировать тестовый отчет (Рисунок 19).

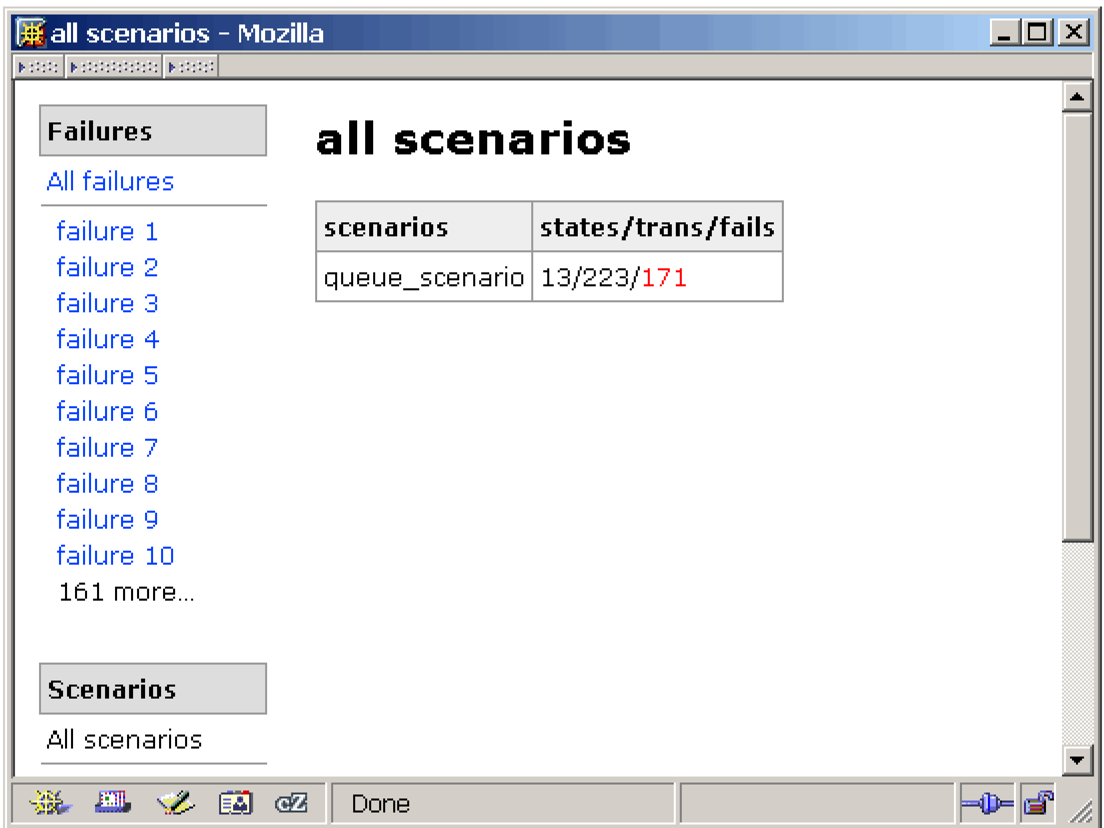


Рисунок 19. Общий вид тестового отчета с ошибкой.

Слева появится список обнаруженных несоответствий, для каждого из которых отчет содержит следующую информацию (Рисунок 20):

- Вид ошибки (в данном случае — нарушение постусловия)
- Позиция ошибки в трассе теста
- Компоненты тестовой системы, в которых проявилось несоответствие (имя сценария, состояние, сценарная и спецификационная функции)
- Значения параметров и элементов покрытия

Рисунок 20. Отчет о найденном несоответствии.

Приложение А: Код теста очереди

Реализация

queue.h

```
#ifndef __queue_h__
#define __queue_h__

struct queue {
  void *item;
  struct queue *next;
};

struct queue *create_queue (void);
void delete_queue (struct queue **queue);

int empty (struct queue *queue);
void enq (struct queue *queue, void *item);
void *deq (struct queue *queue);

#endif / __queue_h__ */
```

queue.c

```
#include "queue.h"
#include <stdlib.h>

struct queue *create_queue (void)
{
  struct queue *q =
    (struct queue *)malloc (sizeof (struct queue));

  q->item = NULL;
  q->next = NULL;

  return q;
}

void delete_queue (struct queue **queue)
{
  while (*queue != NULL) {
    struct queue *q = (*queue)->next;
    free (*queue);
    *queue = q;
  }
  *queue = NULL;
}

int empty (struct queue *queue)
{
  return !queue->next;
}

void enq (struct queue *queue, void *item)
{
  while (queue->next != NULL) queue = queue->next;

  queue = queue->next =
    (struct queue *)malloc (sizeof (struct queue));
  queue->item = item;
  queue->next = NULL;
}

void *deq (struct queue *queue)
{
  void *res;

  struct queue *q = queue->next;
  queue->next = q->next;
  res = q->item;
  free (q);

  return res;
}
```

Спецификации

queue_spec.seh

```
#ifndef __queue_spec_seh__
#define __queue_spec_seh__

#include <atl/map.h>
#include <atl/list.h>

invariant typedef void *item_t;

extern invariant Map *model_queues;

typedef void *queue_t;

const queue_t null_queue;

specification queue_t create_queue_spec (void)
    updates model_queues;

specification void delete_queue_spec (queue_t *queue)
    updates model_queues;

specification bool empty_spec (queue_t queue)
    reads model_queues;

specification void enq_spec (queue_t queue, item_t item)
    updates model_queues;

specification item_t deq_spec (queue_t queue)
    updates model_queues;

void init_state_queue (void);

specification typedef item_t Item;
specification typedef queue_t Queue;

bool exists_queue_aux (Map *model_queues, queue_t queue);
List *get_queue_aux (Map *model_queues, queue_t queue);
void add_queue_aux(Map *model_queues, queue_t queue, List *list);
void remove_queue_aux (Map *model_queues, queue_t queue);
int size_queue_aux (Map *model_queues, queue_t queue);

void add_last_aux (List *list, item_t item);
item_t remove_first_aux (List *list);

Item *create_item_aux (item_t item);
Queue *create_queue_aux (queue_t queue);

#endif /* __queue_spec_seh__ */
```

queue_spec.sec

```
#include "queue_spec.seh"

#pragma SEC subsystem queue "queue"
```

```
const queue_t null_queue = NULL;

specification typedef item_t Item = {};
specification typedef queue_t Queue = {};

invariant (item_t item)
{
  return NULL != item;
}

invariant Map *model_queues;

invariant (model_queues)
{
  return !containsKey_Map ( model_queues
                          , create (&type_Queue, null_queue)
                          );
}

specification queue_t create_queue_spec (void)
    updates model_queues
{
  post {
    Map *tmp_map = clone (model_queues);

    if (  null_queue == create_queue_spec
       || !exists_queue_aux (model_queues, create_queue_spec)
       || 0 != size_queue_aux (model_queues, create_queue_spec)
       || exists_queue_aux ( @ model_queues
                           , create_queue_spec
                           )
       )
      return false;

    remove_queue_aux (tmp_map, create_queue_spec);
    return equals (tmp_map, @ model_queues);
  }
}

specification void delete_queue_spec (queue_t *pqueue)
    updates model_queues
{
  pre {
    return null_queue == *pqueue
        || exists_queue_aux (model_queues, *pqueue);
  }
  coverage C {
    if (null_queue == *pqueue) return {null, "null queue"};
    else return {not_null, "non-null queue"};
  }
  post {
    if (coverage (C) == null) {
      return @*pqueue == *pqueue
          && equals (@model_queues, model_queues);
    } else {
      Map *tmp_map = @clone (model_queues);

      remove_queue_aux (tmp_map, @*pqueue);
      return null_queue == *pqueue
          && equals (tmp_map, model_queues);
    }
  }
}
```

```
specification bool empty_spec (queue_t queue)
    reads model_queues
{
  pre {
    return null_queue != queue
        && exists_queue_aux (model_queues, queue);
  }
  coverage C {
    if (0 == size_queue_aux (model_queues, queue))
      return {empty, "empty queue"};
    else
      return {not_empty, "non-empty queue"};
  }
  post {
    return empty_spec == (coverage (C) == empty);
  }
}

specification void enq_spec (queue_t queue, item_t item)
    updates model_queues
{
  pre {
    return null_queue != queue
        && exists_queue_aux (model_queues, queue);
  }
  coverage C {
    if (0 == size_queue_aux (model_queues, queue))
      return {empty, "empty queue"};
    else
      return {not_empty, "non-empty queue"};
  }
  post {
    Map *tmp_map = @clone (model_queues);
    List *tmp_list =
      @clone (get_queue_aux (model_queues, queue));

    add_last_aux (tmp_list, item);
    add_queue_aux (tmp_map, queue, tmp_list);

    return equals (tmp_list, get_queue_aux (model_queues, queue))
        && equals (tmp_map, model_queues);
  }
}

specification item_t deq_spec (queue_t queue)
    updates model_queues
{
  pre {
    return null_queue != queue
        && exists_queue_aux (model_queues, queue)
        && 0 < size_queue_aux (model_queues, queue);
  }
  coverage C {
    if (1 == size_queue_aux (model_queues, queue))
      return {last, "last element"};
    else
      return {not_last, "non-last element"};
  }
  post {
    Map *tmp_map = @clone (model_queues);
    List *tmp_list =
      @clone (get_queue_aux (model_queues, queue));
```

```
    item_t tmp_item;

    tmp_item = remove_first_aux (tmp_list);
    add_queue_aux (tmp_map, queue, tmp_list);

    return deq_spec == tmp_item
        && equals (tmp_list, get_queue_aux (model_queues, queue))
        && equals (tmp_map, model_queues);
  }
}

void init_state_queue (void)
{
  model_queues = create (&type_Map, &type_Queue, &type_List);
}

bool exists_queue_aux (Map *model_queues, queue_t queue)
{
  return containsKey_Map(model_queues, create_queue_aux (queue));
}

List *get_queue_aux (Map *model_queues, queue_t queue)
{
  return get_Map (model_queues, create_queue_aux (queue));
}

void add_queue_aux (Map *model_queues, queue_t queue, List *list)
{
  put_Map (model_queues, create_queue_aux (queue), list);
}

void remove_queue_aux (Map *model_queues, queue_t queue)
{
  remove_Map (model_queues, create_queue_aux (queue));
}

int size_queue_aux (Map *model_queues, queue_t queue)
{
  return size_List (get_queue_aux (model_queues, queue));
}

void add_last_aux (List *list, item_t item)
{
  append_List (list, create_item_aux (item));
}

item_t remove_first_aux (List *list)
{
  Item *item = get_List (list, 0);
  remove_List (list, 0);
  return *item;
}

Item *create_item_aux (item_t item)
{
  return create (&type_Item, item);
}

Queue *create_queue_aux (queue_t queue)
{
  return create (&type_Queue, queue);
}
```

Медиаторы

queue_media.seh

```
#ifndef __queue_media_seh__
#define __queue_media_seh__

#include "queue_spec.seh"

mediator create_queue_media for
specification queue_t create_queue_spec (void)
    updates model_queues;

mediator delete_queue_media for
specification void delete_queue_spec (queue_t *queue)
    updates model_queues;

mediator empty_media for
specification bool empty_spec (queue_t queue)
    reads model_queues;

mediator enq_media for
specification void enq_spec (queue_t queue, item_t item)
    updates model_queues;

mediator deq_media for
specification item_t deq_spec (queue_t queue)
    updates model_queues;

#endif /* __queue_media_seh__ */
```

queue_media.sec

```
#include "queue_media.seh"
#include "queue.h"

static List *queue_to_list (queue_t queue)
{
  struct queue *q;
  List *model;

  if (null_queue == queue) return NULL;

  q = ((struct queue *)queue)->next;
  model = create (&type_List, &type_Item);

  while (NULL != q) {
    append_List (model, create_item_aux (q->item));
    q = q->next;
  }

  return model;
}

void queue_map_state_up (void)
{
  int qi;
  Map *old_model_queues = model_queues;
```

```
  model_queues = create (&type_Map, &type_Queue, &type_List);

  for (qi = 0; qi < size_Map (old_model_queues); qi++) {
    Queue *q = key_Map (old_model_queues, qi);
    put_Map (model_queues, q, queue_to_list (*q));
  }
}

mediator create_queue_media for
specification queue_t create_queue_spec (void)
    updates model_queues
{
  call {
    return (queue_t)create_queue ();
  }
  state {
    queue_map_state_up ();
    add_queue_aux (model_queues
                 , create_queue_spec
                 , queue_to_list (create_queue_spec)
                 );
  }
}

mediator delete_queue_media for
specification void delete_queue_spec (queue_t *queue)
    updates model_queues
{
  queue_t tmp_queue = *queue;
  call {
    delete_queue ((struct queue **)queue);
  }
  state {
    remove_queue_aux (model_queues, tmp_queue);
    queue_map_state_up ();
  }
}

mediator empty_media for
specification bool empty_spec (queue_t queue)
    reads model_queues
{
  call {
    return empty ((struct queue *)queue) ? true : false;
  }
  state {
    queue_map_state_up ();
  }
}

mediator enq_media for
specification void enq_spec(queue_t queue, item_t item)
    updates model_queues
{
  call {
    enq ((struct queue *)queue, item);
  }
  state {
    queue_map_state_up ();
  }
}
```

```
mediator deq_media for
specification item_t deq_spec(queue_t queue)
  updates model_queues
{
  call {
    return deq ((struct queue *)queue);
  }
  state {
    queue_map_state_up ();
  }
}
```

Сценарий

queue_scen.seh

```
#ifndef __queue_scen_h__
#define __queue_scen_h__

extern scenario dfsm queue_scenario;

#endif /* __queue_scen_h__ */
```

queue_scen.sec

```
#include "queue_scen.seh"
#include "queue_spec.seh"
#include <atl/integer.h>

static int queue_max_size = 10;

static int queue_items_num = 20;
static item_t *queue_items;

static queue_t queue;

bool queue_scenario_init (int argc, char **argv)
{
  int i;

  if (argc > 1) queue_max_size = atoi (argv[1]);
  if (argc > 2) queue_items_num = atoi (argv[2]);

  queue_items =
   (item_t *)calloc (queue_items_num, sizeof (item_t));
  for (i = 0; i < queue_items_num; i++)
    queue_items[i] = malloc (i);

  queue = create_queue_spec ();

  return true;
}

void queue_scenario_finish ()
{
```

```
  int i;

  delete_queue_spec (&queue);

  for (i = 0; i < queue_items_num; i++)
    free (queue_items[i]);
  free (queue_items);
}

Object *queue_scenario_state ()
{
  return create (&type_Integer
                , size_queue_aux (model_queues, queue)
                );
}

scenario bool empty_scen ()
{
  empty_spec (queue);
  return true;
}

scenario bool enq_scen ()
{
  if (size_queue_aux (model_queues, queue) != queue_max_size)
    iterate (int i = 0; i < queue_items_num; i++; )
      enq_spec (queue, queue_items[i]);
  return true;
}

scenario bool deq_scen ()
{
  if (size_queue_aux (model_queues, queue) != 0)
    deq_spec (queue);
  return true;
}

scenario dfsm queue_scenario =
{
  .init     = queue_scenario_init,
  .finish   = queue_scenario_finish,
  .getState = queue_scenario_state,
  .actions  = {empty_scen, enq_scen, deq_scen, NULL}
};
```

Функция main

queue_main.sec

```
#include "queue_spec.seh"
#include "queue_media.seh"
#include "queue_scen.seh"

void main (int argc, char **argv)
{
  set_mediator_create_queue_spec (create_queue_media);
  set_mediator_delete_queue_spec (delete_queue_media);
  set_mediator_empty_spec (empty_media);
  set_mediator_enq_spec (enq_media);
  set_mediator_deq_spec (deq_media);

  init_state_queue ();

  queue_scenario (argc, argv);
}
```