

CTesK 2.2 Community Edition: Быстрое знакомство

Содержание

Введение	1
Соглашения о форматировании	2
Другие документы	2
Пример тестируемой системы: Банковский кредитный счет	3
Спецификация функциональности системы Банковского кредитного счета	4
Медиаторы для системы Банковского кредитного счета	11
Тестовый сценарий для системы Банковского кредитного счета.....	13
Выполнение теста для системы Банковского кредитного счета	17
Анализ результатов выполнения теста для системы Банковского кредитного счета	19
Генерация тестовых отчетов	19
Итоговый отчет	19
Подробный отчет сценария	19
Итоговый отчет по покрытию функций	20
Подробный отчет по покрытию функций	21
Итоговый отчет об обнаруженных нарушениях	24
Подробный отчет об обнаруженном нарушении	25
Приложение А: Использование CTesK с компилятором GCC.....	29
Использование утилиты GNU Make для сборки теста	29
Выполнение тестов	29
Генерация тестового отчета.....	30

Введение

Данный документ знакомит с основными понятиями CTesK и языка SeC, предоставляя возможность быстрого старта разработки тестов в среде CTesK.

CTesK реализует технологию разработки тестов UniTesK, написанного на языке программирования C. UniTesK — промышленная технология автоматизированной разработки тестов, основанная на формальных методах.

Данная технология поддерживает разработку тестов для *функционального тестирования*. Функциональное тестирование обеспечивает проверку поведения тестируемой программы на соответствие *функциональным требованиям*.

Любая программная система предоставляет интерфейс, через который окружение взаимодействует с этой системой. Поведение системы соответствует функциональным требованиям, если любые наблюдаемые снаружи результаты ее работы согласуются с этими требованиями. То есть функциональные требования *не определяют* как должна быть реализована программная система, они определяют какие видимые снаружи результаты должны производиться при взаимодействии окружения с системой через ее интерфейс.

Автоматизация разработки функциональных тестов, проверяющих выполнение функциональных требований, возможна только при строгом формальном определении требований. Здесь под “формальным” подразумевается способ определения, при котором требования имеют однозначную интерпретацию и могут обрабатываться компьютером. То есть, в данном случае, разница между неформальными и формальными спецификациями требований скорее подобна разнице между естественными языками и языками программирования, чем разнице между языками программирования и математическими формальными языками.

CTesK реализует технологию UniTesK для программного обеспечения, реализованного на языке программирования C. В CTesK используется язык SeC (произносится [sek]) — специально разработанное спецификационное расширение языка программирования C (Specification Extension of C). SeC расширяет язык C нотацией для определения пред- и постусловий, критериев покрытия, медиаторов и тестовых сценариев. SeC позволяет разработчикам тестов определять и генерировать компоненты тестовой системы, из которых могут собираться качественные тесты. Так же SeC позволяет определять полностью независимые от реализации спецификации и сценарии, что дает возможность их повторного использования.

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

Транслятор 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 поддержку трассировки выполнения тестов. *Библиотека спецификационных типов* поддерживает типы интегрированные со стандартными функциями создания, инициализации, копирования, сравнения и уничтожения данных

этих типов. Так же библиотека содержит набор уже определенных спецификационных типов. *Генератор тестовых отчетов* предоставляет возможность автоматического анализа трассы выполнения теста и генерацию различных содержательных тестовых отчетов.

Соглашения о форматировании

Курсивом выделяются термины основных понятий и части текста с важной информацией.

“*Курсивом в двойных кавычках*” выделяются ссылки на другие документы по CTesK.

Примеры на SeC представлены в отформатированных абзацах.

Шрифтом с фиксированной 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.

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: Руководство пользователя*” и “*CTesK 2.2 Community Edition: Описание языка SeC*”. Сайт по UniTesK <http://www.unitesk.com/> содержит информацию по UniTesK, CTesK и другим инструментам, поддерживающим UniTesK.

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

Пример тестируемой системы: Банковский кредитный счет

Предполагается, что на Вашем компьютере установлен CTesK 2.2 Community Edition. Если это не так, то установите инструмент, следуя указаниям в документе “*CTesK 2.2 Community Edition: Инструкция по установке и удалению*”.

В документе рассматривается процесс разработки теста с использованием CTesK на примере разработки теста для системы, реализующей функциональность банковского кредитного счета: вклад и снятие денег со счета при заданном максимально допустимом размере кредита.

Кредитный счет реализован как структура `Account`, определенная в файле `account.h` из каталога `examples/account` в дереве каталогов установки CTesK

```
typedef struct Account {
    int balance;
} Account;
```

Допустимый размер кредита должен быть не меньше нуля и определяется макросом `MAXIMUM_CREDIT` в `account.h`.

Тестируемая реализация находится в файле `examples/account/account.c`.

Интерфейс системы состоит из двух функций:

- `void deposit(Account *acct, int sum)` выполняет вклад положительной суммы `sum` на счет, увеличивая баланс счета на эту сумму;
- `int withdraw(Account *acct, int sum)` выполняет снятие положительной суммы `sum` со счета; если разница текущего баланса и суммы `sum` укладывается в допустимый размер кредита, метод возвращает `sum`, иначе — 0.

Далее в документе демонстрируется как, используя CTesK, разработать тест для системы банковского кредитного счета, выполнить тестирование и проанализировать полученные результаты.

Ниже описывается разработка теста, которая состоит из следующих шагов:

- Разработка спецификации тестируемой системы
- Разработка медиаторов
- Разработка тестового сценария
- Выполнение теста;
- Анализ результатов тестирования.

Спецификация функциональности системы Банковского кредитного счета

Поддерживаемая CTesK технология разработки тестов UniTesK предполагает, что требования к тестируемой системе записаны в четкой однозначно интерпретируемой форме. Такая форма представления требований называется *формальной спецификацией*. Формальные спецификации могут использоваться для автоматической генерации программных компонентов тестовой системы, проверяющих соответствие между требованиями и реальным поведением интерфейсных функций тестируемой системы.

В CTesK формальные спецификации разрабатываются на специальном языке SeC¹, являющимся расширением языка программирования C. SeC позволяет описывать *функциональные требования*, которые определяют *функциональность* интерфейсных функций, то есть то, что тестируемая система должна делать при вызовах интерфейсных функций.

Спецификации на SeC имеют синтаксис похожий на синтаксис C. Файлы, содержащие код на SeC имеют расширения **.sec** или **.seh**.

Спецификация системы кредитного счета находится в файле **examples/account/account_model.sec** в дереве каталогов установки CTesK.

Спецификация кредитного счета начинается с включения заголовочного файла **account_model.seh** из каталога **examples/account**:

```
#include <limits.h>
#include "account.h"

extern invariant int MaximalCredit;

invariant typedef Account AccountModel;

specification void deposit_spec (AccountModel *acct, int sum)
  reads    MaximalCredit
  updates balance = acct->balance
;

specification int withdraw_spec (AccountModel *acct, int sum)
  reads    MaximalCredit
  updates balance = acct->balance
;
```

В заголовочном файле **account_model.seh** включаются файлы **limits.h** и **account.h**, объявляется внешняя *переменная с инвариантом* и описываются декларации *типа с инвариантом* и *спецификационных функций*.

Файл **limits.h** включается, чтобы иметь возможность использовать константу **INT_MAX**.

¹ Произносится [сек].

Включение файла **account.h** позволяет использовать в спецификации константу `MAXIMUM_CREDIT` и структуру `Account`:

```
#define MAXIMUM_CREDIT 3

typedef struct Account {
    int balance;
} Account;
```

Тип с инвариантом `AccountModel` вводится для того, чтобы компактно описать требование, накладывающее ограничение на допустимые значения баланса счета. Тип `Account` реализует счет как структуру с единственным полем типа `int`. Значение поля `balance` должно быть не меньше отрицательного числа, модуль которого равен значению, заданным макросом `MAXIMUM_CREDIT`. Чтобы описать формально это требование, в файле **account_model.seh** тип `AccountModel` декларируется как `typedef` типа `Account` с инвариантом. Сам инвариант типа `AccountModel` определяется в **account_model.sec**:

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

Инвариант этого типа возвращает `true`, если значение поля `balance` проверяемой структуры соответствует требованию, и `false` в противном случае.

Инвариант должен быть выполнен до и после вызова любой интерфейсной функции, которая использует данные типа с ограничениями на значения, описанными в инварианте. То есть инвариант содержит общие части спецификаций ограничений этих интерфейсных функций.

Так как множество значений типа `AccountModel` не совпадает с множеством значений типа `Account`, тип `AccountModel` является подтипом типа `Account`.

Переменная с инвариантом `MaximalCredit` объявлена в файле **account_model.sec**.

Там же определен инвариант этой переменной.

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

В инварианте переменной `MaximalCredit` описывается требование к значению допустимого размера кредита — оно не должно быть отрицательным. Инвариант переменной должен выполняться для ее значений до и после вызова любой интерфейсной функции.

Далее в файле **account_model.sec** определяются ограничения на поведение тестируемой системы. На SeC такие ограничения описываются в специальных функциях, помеченных ключевым словом **specification** — *спецификационных функциях*.

Интерфейсной функции `deposit` соответствует спецификационная функция `deposit_spec`.

Интерфейсная функция `deposit` без возвращаемого значения имеет два параметра. Первый является ненулевым указателем на структуру `Account`, представляющую счет, на который должны быть вложены деньги. Второй параметр типа `int` является суммой, которая должна быть вложена на счет. Функция должна прочитать второй параметр и изменить поле `balance` структуры, на которую ссылается первый параметр: после вызова поле `balance` должно быть увеличено в точности на число, переданное во втором параметре. Функция работает правильно при выполнении следующих условий: второй параметр должен быть больше нуля и в сумме с балансом, переданного счета, должен быть не больше максимального допустимого значения типа `int`.

На SeC эти требования описываются в спецификационной функции `deposit_spec`.

```

specification void deposit_spec (AccountModel *acct, int sum)
  reads    MaximalCredit
  updates balance = acct->balance
{
  pre { return (acct != NULL) && (sum > 0) && (balance <= MAX_INT - sum); }

  coverage C {
    if (balance + sum == MAX_INT); 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; }
}

```

Определение спецификационной функции начинается с сигнатуры:

```

specification void deposit_spec (AccountModel *acct, int sum)

```

Сигнатура любой спецификационной функции должна содержать ключевое слово **specification**. Кроме имени, сигнатура спецификационной функции `deposit_spec` отличается от сигнатуры интерфейсной функции `deposit` только типом первого параметра. Он является указателем на тип `AccountModel` — подтип реализационного типа `Account`.

После сигнатуры следуют *ограничения доступа*.

```

specification void deposit_spec (AccountModel *acct, int sum)
  reads    MaximalCredit
  updates balance = acct->balance

```

Они описывают, что при вызове функции `deposit`

- поведение системы зависит от значения переменной `MaximalCredit`², и что ее видимое снаружи значение не должно меняться после вызова;
- поведение системы зависит от значения поля `balance` переданной структуры `acct*`, и что видимое снаружи значение этого поля может измениться после вызова.

Кроме того в описании ограничения доступа поля `balance` определяется его *псевдоним*. Псевдоним используется в теле спецификационной функции для упрощения выражений и улучшения читаемости кода.

Ключевое слово **reads** определяет доступ “только на чтение”, то есть видимые снаружи значения параметров и переменных с доступом **reads** должны оставаться неизменными после вызова специфицируемой функции.

В SeC так же как в C изменение видимого снаружи значения любого параметра, переданного по значению, невозможно, то есть такие параметры всегда имеют ограничение доступа **reads**.

Параметры, переданные через указатели, в SeC интерпретируются более строго. Если указатель отличается от типа `void*`, он рассматривается как указатель на единственное

² В данном случае в соответствии с требованиями реакции тестируемой системы на воздействия определены только при неотрицательном значении максимального кредита, что описано в инварианте переменной `MaximalCredit`. То есть, описание ограничения доступа этой переменной необходимо для автоматической проверки ее инварианта при вызове каждой интерфейсной функции.

значение типа, на который он ссылается, а не как адрес первого элемента в массиве³. Указатели на тип `void` используются так же как в C — только как значения адресов.

Во время выполнения после каждого вызова соответствующей интерфейсной функции CTesK обеспечивает автоматическую проверку того, что значения переменных и параметров с ограничением доступа `reads` не меняются. Так же во время выполнения перед каждым вызовом соответствующей интерфейсной функции CTesK обеспечивает автоматическую проверку выполнения инвариантов типов и переменных для значений параметров и переменных с ограничением доступа `reads`.

Ключевое слово `updates` определяет доступ “на чтение и запись”, то есть видимые снаружи значения параметров и переменных с доступом `updates` могут измениться в результате вызова соответствующей интерфейсной функции. В SeC так же как в C видимые снаружи значения могут изменяться только у параметров, переданных через указатель. Причем, в SeC такие параметры должны рассматриваться строго как указатели на единственное значение соответствующего типа.

Во время выполнения до и после каждого вызова соответствующей интерфейсной функции CTesK обеспечивает автоматическую проверку выполнения инвариантов типов и переменных для значений параметров и переменных с ограничением доступа `updates`. По умолчанию любой параметр, переданный через указатель, имеет доступ “на чтение и запись”.

В теле спецификационной функции `deposit_spec` описывается поведение системы при вызове интерфейсной функции `deposit`. Другими словами, тело спецификационной функции содержит описание функциональных требований в форме *пред- и постусловий* и *критерии покрытия*.

При вызове интерфейсной функции `deposit` указатель на структуру, представляющую счет, должен быть ненулевым, вкладываемая сумма должна быть положительной, и сумма текущего баланса счета и второго параметра должна быть не больше максимально допустимого значения типа `int`. В SeC такие требования описываются в предусловии.

```
pre { return (acct != NULL) && (sum > 0) && (balance <= MAX_INT - sum); }
```

Предусловие содержится в блоке, помеченным ключевым словом `pre`. Оно возвращает `true`, если параметры имеют допустимые входные значения, и `false` в противном случае. То есть предусловие задает область определения функции. Если входные значения параметров вне этой области, поведение функции не определено.

Предусловие должно быть без побочных эффектов. В спецификационной функции может быть только одно предусловие. Оно должно быть перед критериями покрытия и до постусловия. Если нет ограничений на входные значения параметров предусловие может быть опущено.

После вызова интерфейсной функции `deposit` баланс счета `acct` должен быть равен балансу до вызова, увеличенному на сумму `sum`. В SeC это требование описывается в постусловии.

```
post { return balance == @balance + sum; }
```

Постусловие — блок, помеченный ключевым словом `post`. Оно возвращает `true`, если входные и выходные значения параметров и значение возвращаемого результата после

³ Чтобы специфицировать функцию с параметрами, являющимися динамическими массивами, необходимо использовать контейнеры *спецификационных типов*. Более подробную информацию о спецификационных типах и библиотечных контейнерных типах можно найти в “CTesK 2.2: Руководство пользователя” и “CTesK 2.2: Описание языка SeC”

вызова функции соответствуют функциональным требованиям, и `false` в противном случае. То есть постусловие определяет код, проверяющий правильность поведения функции.

Чтобы получить доступ в постусловии к входному значению псевдонима поля `balance`, используется специальный оператор `@`. То есть в постусловии `'balance'` означает значение псевдонима поля `balance` после вызова функции `deposit`, а `'@balance'` — его значение до вызова.

Данный оператор применим к выражениям только внутри блока постусловия. Ключевое слово `post` определяет точку вызова соответствующей интерфейсной функции. В теле спецификационной функции выражения до ключевого слова `post` вычисляются до вызова реализации. Выражения после ключевого слова `post` вычисляются после вызова реализации за исключением выражений, к которым применен оператор `@`.

Постусловие должно быть без побочных эффектов. В спецификационной функции должно быть ровно одно постусловие. Оно должно следовать после предусловия и критериев покрытия.

В соответствии с требованиями функция `deposit` имеет одинаковое поведение на всей области определения. Можно предположить, что поведение любой реализации этой функции не зависит от абсолютного значения текущего баланса счета и величины вкладываемой суммы. Однако оно может зависеть от знака значения баланса. Так же поведение функции должно быть проверено при значениях параметров на границах области определения. Поэтому критерий покрытия спецификационной функции `deposit_spec` определяет пять различных тестовых ситуаций, в каждой из которых поведение функции должно быть проверено.

```
coverage C {
  if (balance + sum == MAX_INT)
    return { maximum, "Maximal deposition" };
  else if (balance > 0)
    return { positive, "Positive balance" };
  else if (balance < 0)
    if (balance == -MAXIMUM_CREDIT)
      return { minimum, "Minimal balance" };
    else
      return { negative, "Negative balance" };
  else
    return { zero, "Empty account" };
}
```

Критерий покрытия — *именованный* блок, помеченный ключевым словом `coverage`. В нем определяется разбиение поведения функции на *ветви функциональности*. Каждая ветвь определяется оператором `return`, возвращающим конструкцию аналогичную конструкции инициализации в декларации переменной структурного типа из двух полей. В первом поле должен быть идентификатор — *идентификатор ветви*, во втором — строковый литерал — *имя ветви*.

Разбиение, определяемое блоком `coverage` должно быть полным и непротиворечивым. То есть любой допустимый набор входных значений параметров должен соответствовать единственной ветви функциональности.

В спецификационной функции могут быть определены несколько критериев покрытия с различными именами. Блоки `coverage` должны следовать после предусловия перед постусловием. Если в спецификационной функции не определено ни одного критерия покрытия, это эквивалентно критерию покрытия с единственной ветвью функциональности.

Интерфейсной функции `withdraw` соответствует спецификационная функция `withdraw_spec`.

```

specification int withdraw_spec (AccountModel *acct, int sum)
reads MaximalCredit
updates balance = acct->balance {
  pre { return (acct != NULL) && (sum > 0); }
  coverage C {
    if (sum == INT_MAX) return {max, "Maximal withdrawal"};
    if (balance > 0)
      if (balance < sum - MaximalCredit)
        return {pos_too_large, "Positive balance. Too large withdrawal"};
      else
        return {positive_ok, "Positive balance. Successful withdrawal"};
    else if (balance < 0)
      if (balance >= sum - MaximalCredit)
        return {neg_too_large, "Negative balance. Too large withdrawal"};
      else
        return {negative_ok, "Negative balance. Successful withdrawal"};
    else
      if (balance < sum - MaximalCredit)
        return {zero_too_large, "Empty account. Too large withdrawal"};
      else
        return {zero_ok, "Empty account. Successful withdrawal"};
  }
  post {
    if (balance >= sum - MaximalCredit)
      return balance == @balance - sum && withdraw_spec == sum;
    else
      return balance == @balance && withdraw_spec == 0;
  }
}

```

Интерфейсная функция `withdraw` возвращает результат типа `int` и имеет два параметра. Первый параметр — ненулевой указатель на структуру `Account`, представляющую счет, с которого должны быть сняты деньги. Второй параметр типа `int` является суммой, которая должна быть снята со счета. Функция должна прочитать второй параметр и изменить поле `balance` структуры, на которую ссылается первый параметр. После вызова поле `balance` должно быть уменьшено в точности на число, переданное во втором параметре, если снимаемая сумма денег не превышает максимально допустимый кредит, в противном случае поле `balance` не изменяется. Функция возвращает сумму снятую со счета в первом случае и 0 во втором.

В предусловии спецификационной функции `withdraw_spec` описываются ограничения на входные значения параметров: указатель `acct` не должен быть равен нулю и сумма снимаемых денег `sum` должна быть положительной.

В блоке `coverage` с функциональность разбивается на семь ветвей. Данный критерий покрытия определяет, что поведение функции должно быть проверено в двух существенно разных ситуациях: когда снятие запрашиваемой суммы возможно, и когда это невозможно. Причем в обеих этих ситуациях поведение функции должно быть проверено при значениях текущего баланса счета из разных подмножеств области определения, в частности, на границе множеств допустимых значений параметров.

В постусловии описываются те же два случая: когда снятие запрашиваемой суммы не приводит к превышению допустимого кредита, и когда снятие запрашиваемой суммы невозможно. В первом случае в постусловии проверяется, что баланс счета уменьшается на переданную сумму `sum` и функция возвращает значение этой суммы. Во втором случае проверяется, что баланс счета не изменяется и функция возвращает 0. Для доступа к

возвращаемому значению в постусловии используется имя спецификационной функции — в данном примере `withdraw_spec`.

Чтобы получить компоненты, проверяющие поведение системы при вызовах описанных интерфейсных функций, спецификации должны быть транслированы в код на С.

Для трансляции файла **`account_model.sec`** в код на С на платформах под управлением ОС Linux необходимо в командном интерпретаторе в каталоге **`examples/account`** в дереве каталогов установки CTesK выполнить команду

```
>sec.sh account_model.sec account_model.c account_model.sei
```

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

Медиаторы для системы Банковского кредитного счета

Чтобы обеспечить возможность проверки соответствия между реализацией и спецификацией тестируемой системы, они должны быть связаны некоторым образом.

В UniTesK для этого используются специальные компоненты — *медиаторы*.

В SeC медиаторы реализуются *медиаторными функциями* — специального вида функциями, помеченными ключевым словом `mediator`.

Медиаторы для тестирования кредитного счета, находятся в файле `account_mediator.sec` из каталога `examples/account` в дереве каталогов установки CTesK.

Файл `account_mediator.sec` начинается с включения заголовочного файла `examples/account/account_mediator.seh`.

```
#include "account_model.seh"

mediator deposit_media for
specification void deposit_spec (AccountModel *acct, int sum)
  reads MaximalCredit
  updates acct->balance
;
mediator withdraw_media for
specification int withdraw_spec (AccountModel *acct, int sum)
  reads MaximalCredit
  updates acct->balance
;
```

Медиаторы должны иметь доступ к типам и функциям как реализации, так и спецификации. Поэтому кроме предварительных деклараций медиаторных функций в файле `account_mediator.seh` включается заголовочный файл спецификации `account_model.seh`, который в свою очередь содержит включение заголовочного файла реализации `account.h`.

Файл `account_mediator.sec` содержит определения двух медиаторных функций.

```
mediator deposit_media for
specification void deposit_spec (AccountModel *acct, int sum)
  reads MaximalCredit
  updates acct->balance
{
  call { deposit (acct, sum); }
}

mediator withdraw_media for
specification int withdraw_spec (AccountModel *acct, int sum)
  reads MaximalCredit
  updates acct->balance
{
  call { return withdraw (acct, sum); }
}
```

Первая медиаторная функция связывает спецификационную функцию `deposit_spec` с интерфейсной функцией реализации `deposit`. Вторая связывает спецификационную функцию `withdarw_spec` с интерфейсной функцией реализации `withdraw`.

Сигнатура медиаторной функции должна содержать ключевое слово **mediator**, имя медиаторной функции, ключевое слово **for** и сигнатуру и ограничения доступа спецификационной функции, связываемую данным медиатором.

В теле медиаторной функции, связывающей спецификационную функцию, должен быть блок, помеченный ключевым словом **call**.

Блок **call** содержит реализацию функциональности, описанной в спецификационной функции, при помощи вызова соответствующей интерфейсной функции реализации. То есть в этом блоке

- параметры медиаторной функции, которые совпадают с параметрами спецификационной функции, должны быть преобразованы в параметры интерфейсной функции реализации;
- интерфейсная функция реализации должна быть вызвана с полученными значениями параметров;
- возвращенное значение и выходные значения параметров после вызова реализации должны быть преобразованы в возвращаемое значение и выходные параметры медиаторной функции.

Для трансляции файла **account_mediator.sec** в 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_mediator.sec account_mediator.c account_mediator.sei
```

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

Тестовый сценарий для системы Банковского кредитного счета

Спецификации являются формальным описанием функциональных требований к тестируемой системе. Из них генерируются компоненты, проверяющие отдельные вызовы интерфейсных функций системы. Медиаторы связывают спецификации и тестируемую реализацию. Они позволяют использовать одну и ту же спецификацию для тестирования различных реализаций одной и той же функциональности.

Чтобы проверить поведение системы в различных ситуациях, нужно оказать последовательность воздействий на нее, вызывая различные интерфейсные функции с различными значениями параметров.

В CTesK последовательность тестовых воздействий строится автоматически при помощи *обходчика*. Для построения тестовой последовательности обходчику необходимо краткое описание теста — *тестовый сценарий*.

Тестовый сценарий для кредитного счета находится в файле **account_scenario.sec** из каталога **examples/account** в дереве каталогов установки CTesK.

```
#include "account_mediator.she"
#include <atl/integer.h>

AccountModel Acct;

static bool account_init (int argc, char **argv) {
    Acct.balance = 0;
    set_mediator_deposit_spec (deposit_media);
    set_mediator_withdraw_spec (withdraw_media);
    return true;
}

static Integer* account_state() { return create_Integer(Acct.balance); }

scenario bool deposit_scen() {
    if (Acct.balance <= 5) {
        iterate (int i = 1; i <= 5; i++;) deposit_spec(&Acct, i);
    }
    return true;
}

scenario bool withdraw_scen() {
    iterate (int i = 1; i <= 5; i++;) withdraw_spec(&Acct, i);
    return true;
}

scenario dfsm account_scenario = {
    .init      = account_init,
    .getState = account_state,
    .actions  = { deposit_scen, withdraw_scen, NULL }
};
```

Так как в сценарии используются спецификации, медиаторы и реализация, в файл со сценарием должны быть включены соответствующие заголовочные файлы. Файл **account.h** включается в файле **account_model.seh**, который в свою очередь включается в файле **account_mediator.seh**, поэтому только последний включен в файле **account_scenario.sec**.

В данном тестовом сценарии используется единственный экземпляр модельной структуры кредитного счета в виде глобальной переменной. Эта же переменная используется и как экземпляр реализационной структуры кредитного счета.

```
AccountModel Acct;
```

Далее определяется функция инициализации теста.

```
static bool account_init (int argc, char **argv) {
    Acct.balance = 0;
    set_mediator_deposit_spec (deposit_media);
    set_mediator_withdraw_spec (withdraw_media);
    return true;
}
```

Функция `account_init` определяет начальное значение поля `balance` переменной `Acct` равным нулю, устанавливает медиаторы, используемых в сценарии спецификационных функций, и возвращает `true`. Для установки медиаторов используются функции `set_mediator_deposit_spec` и `set_mediator_withdraw_spec`, которым в качестве аргументов передаются указатели на медиаторные функции.

Функция, устанавливающая медиатор спецификационной функции, неявно определяется одновременно с соответствующей спецификационной функцией и имеет имя `set_mediator_<имя спецификационной функции>`.

Функция `account_state` определяет набор состояний, которые считаются разными в данном сценарии — разными считаются состояния при разных значениях баланса счета.

```
#include <atl/integer.h>
...
static Integer* account_state() { return create_Integer(acct.balance); }
```

Поведение тестируемой системы при вызовах интерфейсных функций, соответствующих спецификационным функциям, должно проверяться в различных ситуациях или, другими словами, в различных состояниях сценария. В CTesK требуется, чтобы функция определяющая состояние сценария возвращала указатель на спецификационный тип. В рассматриваемом сценарии состояние строится из значения `acct.balance` и возвращается в виде спецификационной ссылки типа `Integer*` — спецификационный тип из библиотеки CTesK, для использования которого необходимо включить заголовочный файл `atl\integer.h`.

В *сценарных функциях* задается набор отдельных тестовых воздействий, которые совершаются в каждом состоянии достигнутом при выполнении сценария.

```
scenario bool deposit_scen() {
    if (Acct.balance <= 5)
        iterate (int i = 1; i <= 5; i++;) deposit_spec(&Acct, i);
    return true;
}

scenario bool withdraw_scen() {
    iterate (int i = 1; i <= 5; i++;) withdraw_spec(&Acct, i);
    return true;
}
```

В первой функции `deposit_scen` определяется набор воздействий через интерфейсную функцию `deposit` при помощи вызова соответствующей спецификационной функции. Во второй функции `withdraw_scen` определяется набор воздействий через интерфейсную функцию `withdraw`.

Сценарные функции в каждом достижимом состоянии должны обеспечить максимально возможное покрытие по критериям, определенным в спецификационных функциях. То есть в каждом состоянии каждая интерфейсная функция должна быть вызвана с такими наборами значений параметров, чтобы покрылись все функциональные ветви, которые возможно покрыть в этом состоянии.

Для задания последовательности воздействий используется оператор `iterate(;;;)`. Данный оператор имеет синтаксис похожий на синтаксис оператора языка C `for(;;)` за исключением последнего выражения, которое задает условие фильтрации воздействий. В теле оператора `iterate` задаются воздействия, которые должны быть произведены в каждом достигнутом состоянии сценария.

Обе сценарные функции итерируют целочисленное значение и вызывают соответствующую спецификационную функцию, передавая ей в качестве параметров указатель на структуру кредитного счета `Acct` и значение итерационной переменной.

Единственное отличие этих функций в том, что в функции `deposit_scen` необходимо условие остановки итерации, чтобы не допустить слишком большого количества вызовов функции `deposit_spec` и переполнения в итерационной переменной. Без этого условия значение вкладываемой суммы будет увеличиваться вплоть до максимально возможного значения типа `int`. В условии остановки итерации определяется, что функция `deposit_spec` вызывается, если значение баланса счета не больше 5. В функции `withdraw_spec` условия остановки итерации не требуется, так как итерация ограничена естественным образом при достижении максимально допустимого значения кредита.

На основе дополнительной информации в сценарии о состоянии окружения в сценарной функции могут производиться дополнительные проверки правильности поведения тестируемой системы. Сценарная функция возвращает булевский результат этих проверок. В рассматриваемом случае никаких дополнительных проверок не требуется, поэтому обе функции всегда возвращают `true`.

После определения всех необходимых функций следует определение самого сценария.

```
scenario dfsM account_scenario = {
    .init      = account_init,
    .getState = account_state,
    .actions  = { deposit_scen, withdraw_scen, NULL }
};
```

В SeC тестовый сценарий определяется при помощи декларации с инициализацией глобальной переменной, помеченной ключевым словом `scenario`. Тип такой переменной должен соответствовать типу обходчика, используемому в сценарии. В рассматриваемом сценарии используется тип обходчика `dfsM`⁴.

Данный тип является структурой со следующими полями:

- `init` — указатель на функцию типа `bool (PtrInit)(int, char**)`;
- `state` — указатель на функцию типа `Object* (PtrGetState)(void)`, `Object` — специальный библиотечный спецификационный тип, ссылка любого

⁴ В CTesK 2.2 Community Edition также реализован тип обходчика `ndfsM`.

спецификационного типа может использоваться как ссылка этого типа без явного приведения типов;

- `actions` — массив указателей на сценарные функции, заканчивающийся нулевым указателем;
- `finish` — указатель на функцию типа `void (PtrFinish) (void)`.

Поле `init` инициализируется функцией инициализации сценария `account_init`.

Поле `getState` инициализируется функцией `account_state`, возвращающей текущее состояние сценария.

Поле `actions` инициализируется массивом из двух сценарных — `deposit_scen` и `withdraw_scen`.

Поле `finish` должно инициализироваться указателем на функцию, которая производит необходимые действия по окончании работы теста.

В рассматриваемом сценарии никаких завершающих действий не требуется, поэтому поле `finish` в сценарной переменной не инициализируется.

Для трансляции файла **`account_scenario.sec`** в 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`**.

Выполнение теста для системы Банковского кредитного счета

Последний компонент теста для системы банковского кредитного счета находится в файле **account_main.sec** каталога **examples/account** в дереве каталогов установки CTesK. В этом файле содержится определение функции `main` тестовой программы.

```
#include "account_scenario.seh"

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

Включаемый заголовочный файл **account_scenario.seh** содержит декларацию внешней сценарной переменной.

```
extern scenario dfsml account_scenario;
```

Функция `main` запускает сценарий, передавая ему опции командой строки в качестве аргументов.

```
account_scenario(argc, argv);
```

Для трансляции файла **account_main.sec** в 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_main.sec account_main.c account_main.sei
```

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

Теперь имеются все файлы с кодом на языке C необходимые для компиляции и сборки теста: исходный файл реализации — **account.c**, и сгенерированные файлы — **account_model.c**, **account_mediator.c**, **account_scenario.c** и **account_main.c**.

Эти файлы должны быть откомпилированы в объектные файлы, которые должны быть собраны в исполняемый файл. Тест компилируется и собирается так же как любая другая программа на языке C. Единственным дополнительным требованием, которое должно быть выполнено при сборке теста, разработанного при помощи CTesK, является обязательное включение в сборку статических библиотек CTesK **libatl.a**, **libts.a**, **libtracer.a** и **libutils.a**. Эти библиотеки находятся в каталоге **lib/linux** дерева каталогов установки CTesK.

Для сборки исполняемого файла с помощью GCC необходимо в командном интерпретаторе в каталоге **examples/account** дерева каталогов установки CTesK запустить

```
>make
```

В результате в каталоге **examples/account** должен появиться исполняемый файл **account**.

Опции тестовой программы передаются из функции `main` в тестовый сценарий. Обходчик типа `dfsm` обрабатывает следующие стандартные опции⁵:

- t <file-name>** — перенаправить трассу в файл '**<file-name>**'
- tc** — выводить трассу на консоль
- tt** — перенаправить трассу в файл '**<scenario-name>--YY-MM-DD--HH-MM-SS.utt'**
- nt** — не создавать трассу

Остальные опции передаются без изменений в функцию инициализации сценария, указанную разработчиком в поле `init` сценарной переменной.

Запустим полученный исполняемый файл, перенаправив трассу в файл **trace.utt**.

Для запуска теста на платформах под управлением ОС Linux необходимо в командном интерпретаторе в каталоге **examples/account** в дереве каталогов установки CTesK выполнить команду

```
>account -t trace.utt
```

В результате выполнения теста в каталоге **examples/account** должен сгенерироваться файл **trace.utt**.

⁵ По умолчанию используется опция `-tt`, то есть трасса перенаправляется в файл '**<scenario-name>--YY-MM-DD--HH-MM-SS.utt'**.

Анализ результатов выполнения теста для системы Банковского кредитного счета

Генерация тестовых отчетов

На платформах под управлением ОС Linux в командном интерпретаторе в каталоге, содержащем файл с трассой теста, нужно выполнить команду
`>ctesk-rg.sh -d trace.report trace.utt`

В результате в каталоге, содержащем файл с трассой теста, сгенерируется директория, с тестовыми отчетами. В случае генерации отчета из командной строки нужно запустить программу просмотра документов в формате html и открыть в нем файл **index.html** из сгенерированного каталога **trace.report**.

Итоговый отчет

Стартовая страница содержит *итоговый отчет*. В нем отображены количество проверенных состояний и переходов и количество обнаруженных нарушений при выполнении каждого сценария теста.

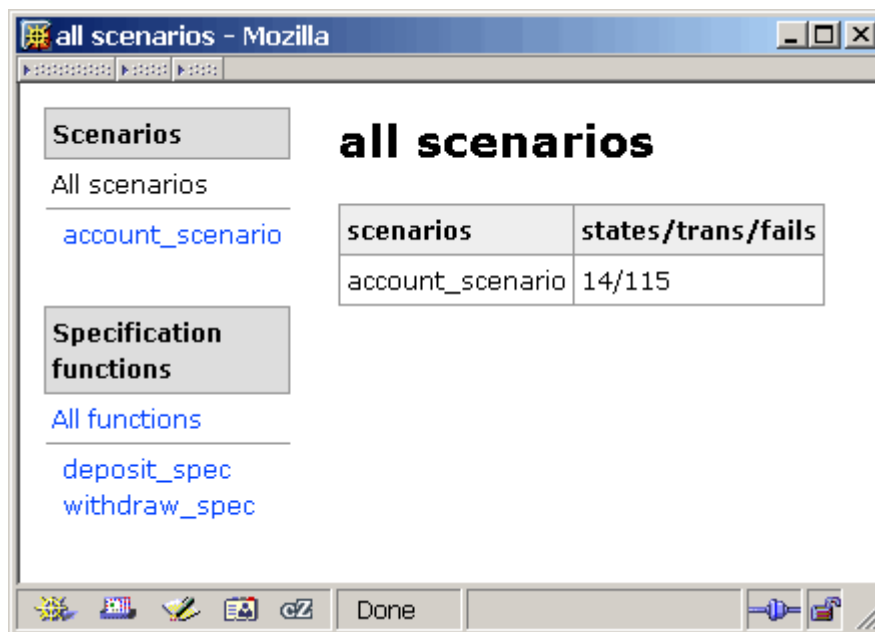


Рисунок 1. Итоговый отчет.

В рассматриваемом случае тест состоит из одного сценария. Было проверено **14** состояний и **115** переходов. Нарушений не обнаружено.

Подробный отчет сценария

Подробный отчет сценария открывается при переходе по ссылке с именем сценария. Отчет описывает все состояния и переходы, проверенные во время выполнения данного сценария. Второй столбец таблицы описывает переходы. Первый и второй столбцы

таблицы описывают начальные и конечные состояния при переходах. Последний столбец показывает сколько было проходов по данному переходу из данного начального состояния в данное конечное, и сколько при этом было обнаружено нарушений.

scenarios	states/trans
account_scenario	14/115

start states	transitions	end states	hits
-1	withdraw_scen (int i = 3)	-1	1
	withdraw_scen (int i = 4)		1
	withdraw_scen (int i = 5)		1
-1	withdraw_scen (int i = 1)	-2	10
-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

Рисунок 2. Подробный отчет сценария.

Так при выполнении теста по сценарию для кредитного счета было десять переходов из состояния сценария **-1**, то есть при текущем значении баланса **-1**.

Переход, помеченный как **deposit_scen(int i = 1)** переводит сценарий в состояние **0**. Переход совершается при вызове сценарной функции `deposit_scen` в состоянии **-1** со значением итерационной переменной *i* равным **1**. Данный переход был совершен **13** раз, нарушений при этом обнаружено не было.

Итоговый отчет по покрытию функций

Итоговый отчет по покрытию функций открывается при переходе по ссылке с именем **All functions**. Отчет показывает процент достигнутого покрытия функциональных ветвей для каждой функции, которая тестировалась в тесте.

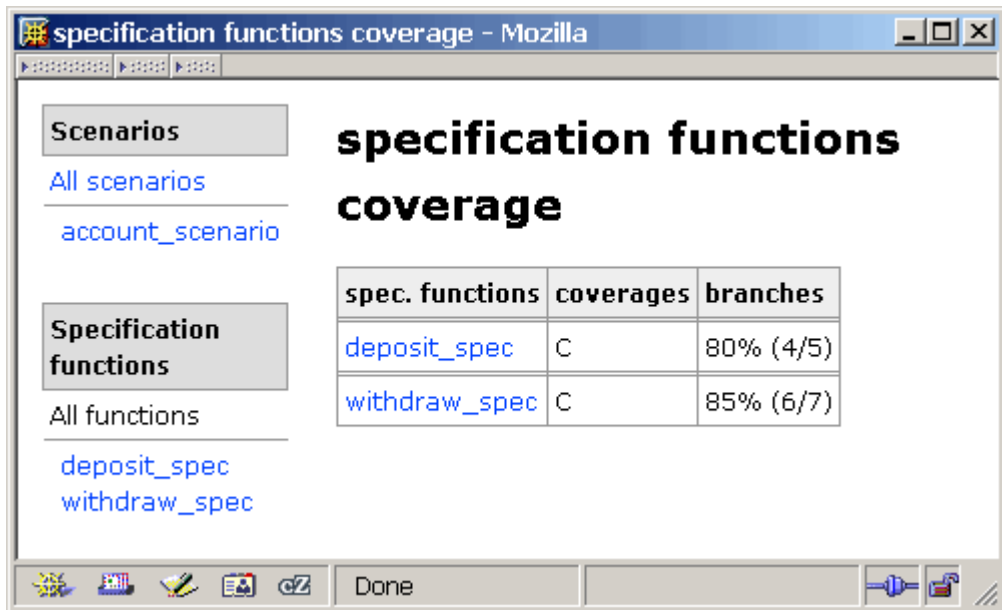
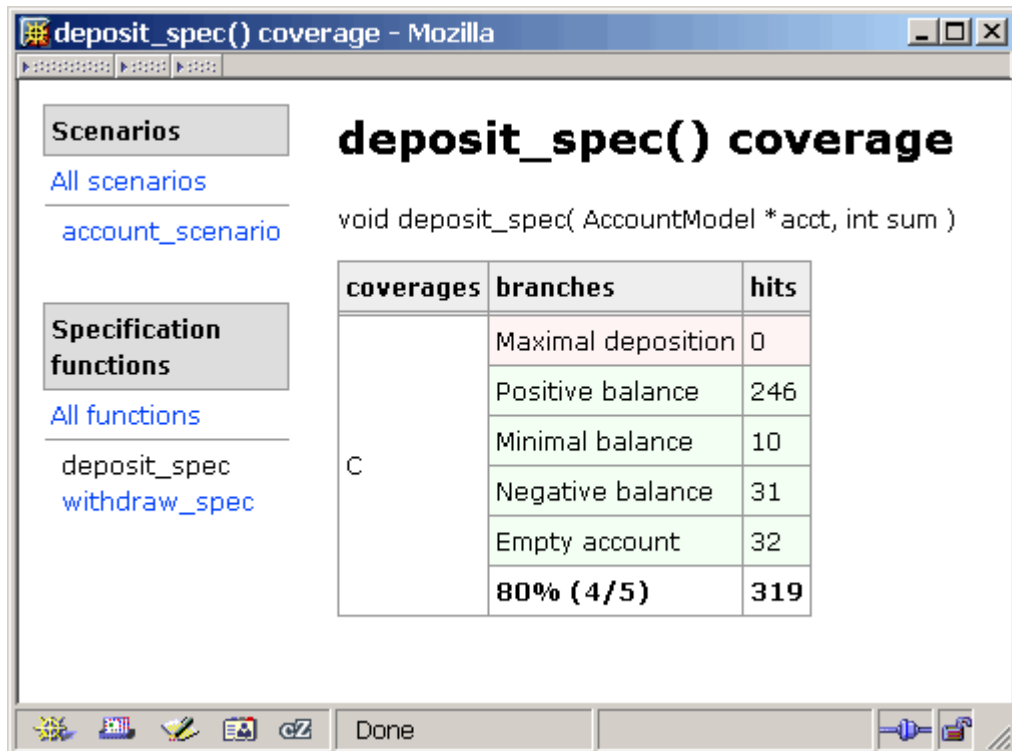


Рисунок 3. Итоговый отчет по покрытию функций.

В тесте для кредитного счета тестируется две функции. Из пяти функциональных ветвей функции `deposit_spec` покрывается четыре ветви. Из семи ветвей функции `withdraw_spec` покрываются шесть.

Подробный отчет по покрытию функций

Подробный отчет по покрытию функции открывается при переходе по ссылке с именем функции. Отчет содержит информацию о количестве попаданий в каждую функциональную ветвь каждой функции.

Рисунок 4. Подробный отчет по покрытию функции `deposit_spec`.

Из отчета о покрытии функции `deposit_spec` видно, что было сделано всего **319** вызовов функции, причем из них **246**, **10**, **31** и **32** были сделаны со значениями параметров, соответствующими ветвям **Positive balance**, **Minimal balance**, **Negative balance** и

Empty account соответственно. Со значениями параметров, соответствующими функциональной ветви **Maximal deposition**, не было сделано ни одного вызова.

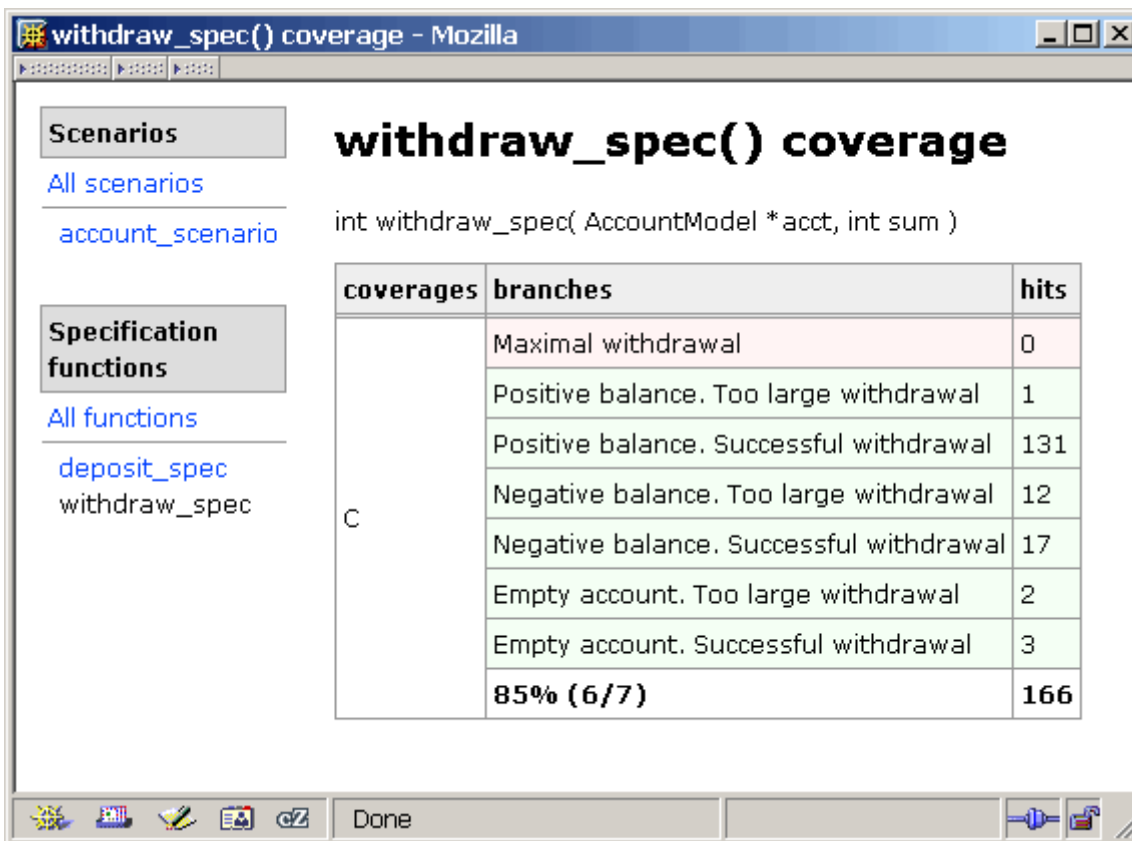


Рисунок 5. Подробный отчет по покрытию функции `withdraw_spec`.

Из отчета о покрытии функции `withdraw_spec` видно, что были сделаны вызовы функции со значениями параметров, соответствующими всем ветвям, за исключением функциональной ветви **Maximal withdrawal**.

Чтобы обеспечить покрытие непокрытых ветвей функций `deposit_spec` и `withdraw_spec` нужно в сценарии определить сценарные функции, которые поставляют значения параметров, при которых происходит вкладывание на счет максимально допустимой суммы и попытка снятия со счета суммы равной максимальному значению типа `int`.

```

scenario bool deposit_max_scen() {
if (0 < acct.balance && acct.balance < INT_MAX)
    deposit_spec(&acct, INT_MAX - acct.balance);
return true;
}

scenario bool withdraw_max_scen() {
    withdraw_spec(&acct, INT_MAX);
    return true;
}

scenario dfsm account_scenario = {
    .init = account_init,
    .getState = (PtrGetState)account_state,
    .actions = { deposit_scen, withdraw_scen,
                deposit_max_scen, withdraw_max_scen,
                NULL
            }
};

```

Условие в функции `deposit_max_scen` необходимо для того, чтобы во время тестирования не возникло переполнения при вычислении выражения `INT_MAX - acct.balance`, и не нарушилось предусловие функции `deposit_spec` при вкладе на счет нулевой суммы.

Однако теперь количество состояний сценария будет равно сумме `INT_MAX` и `MaximalCredit`. Чтобы уменьшить количество состояний до приемлемого числа необходимо изменить сценарную функцию `withdraw_scen`:

```
scenario bool withdraw_scen() {
    if (acct.balance <= 5)
        iterate (int i = 1; i <= 5; i++;) withdraw_spec(&acct, i);
    return true;
}
```

То есть при значениях баланса больших 5 будут вызываться только две новые функции.

После этого надо оттранслировать измененный файл `account_scenario.sec` в код на языке C, заново собрать исполняемый файл, запустить снова тест, и сгенерировать отчеты.

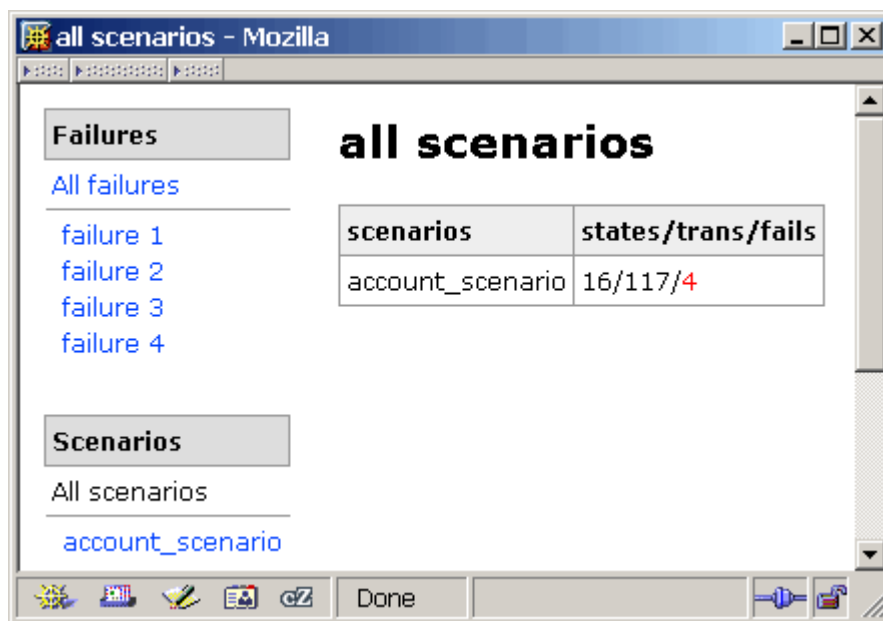


Рисунок 6. Итоговый отчет после изменения сценария.

В навигационном списке итогового отчета появляются новые элементы: ссылки на отчет о нарушениях и отчеты о каждом из них. Кроме того, в таблице появляется число, выделенное красным цветом, — число обнаруженных нарушений в сценарии.

Из отчета о покрытии функций видно, что в функциях `deposit_spec` и `withdraw_spec` покрылись все ветви.

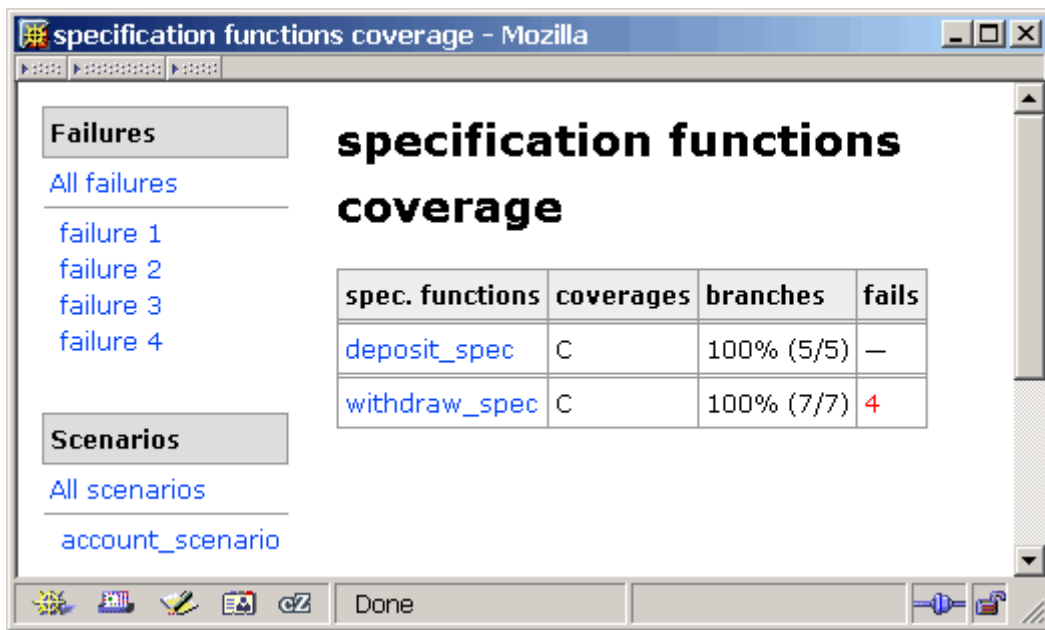


Рисунок 7. Покрытие функций после изменения сценария.

В одной из ветвей обнаружены нарушения.

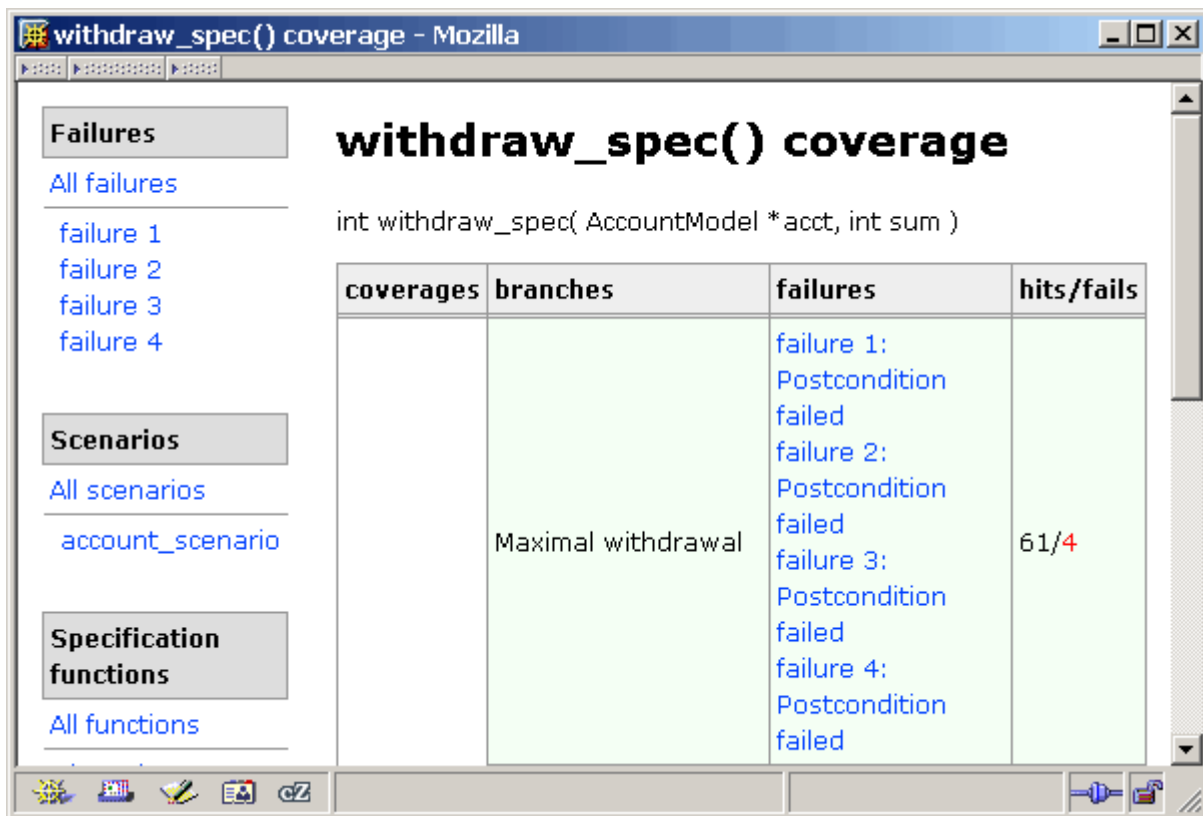


Рисунок 8. Отчет о покрытии функции withdraw_spec после изменения сценария.

Из отчета о покрытии функции видно, что, во-первых, после изменения сценария покрылась ветвь **Maximal withdrawal**, которая оставалась непокрытой ранее. Во-вторых, именно в этой ветви обнаружены нарушения.

Итоговый отчет об обнаруженных нарушениях

Итоговый отчет об обнаруженных нарушениях открывается по ссылке с именем 'All failures'. Отчет содержит список нарушений с краткими описаниями типов нарушений

и мест, где они обнаружены: обнаружено нарушение в постусловии функции `withdraw_спец`.

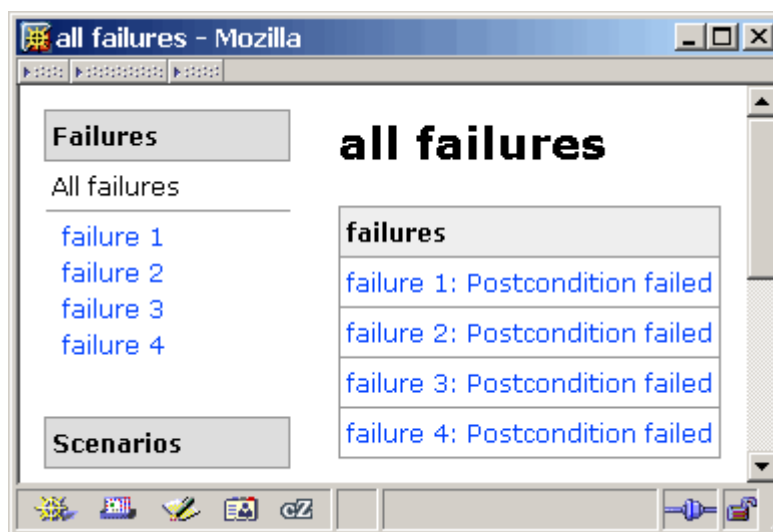


Рисунок 9. Итоговый отчет об обнаруженных ошибках.

Подробный отчет об обнаруженном нарушении

Подробный отчет об обнаруженном нарушении открывается по ссылке с именем `'failure <номер нарушения>'`.



Рисунок 10. Подробный отчет об обнаруженном нарушении.

Отчет содержит подробное описание одного обнаруженного нарушения:

- **location** — место обнаружения нарушения: в трассе, файл **trace.xml**, номер строки **4379**;
- **scenario** — сценарий, в котором обнаружено нарушение: **account_scenario**;
- **state** — состояние сценария перед обнаружением нарушения: **-3** (значение текущего баланса перед вызовом функции);
- **transition** — сценарная функция и значения итерационных переменных, при вызове с которыми указанной сценарной функции происходит нарушение: **withdraw_max_scen()**;
- **specification function** — спецификационная функция, в которой обнаружено нарушение: **withdraw_spec()**;

- **parameter value** — значение параметров спецификационной функции при обнаружении нарушения: `acct = <004AB33C>ptr to struct { -3 }`;
- **coverage & branch** — ветви критериев покрытия, которым соответствуют значения параметров при обнаружении нарушения: `C Maximal withdraw`;
- **prime formula** — значения проверок инвариантов и сохранения значений параметров и переменных с доступом `reads`: все инварианты выполняются и значение переменной с доступом `reads` сохраняется.

Дополнительная информация о нарушении может быть найдена в подробном отчете сценария.

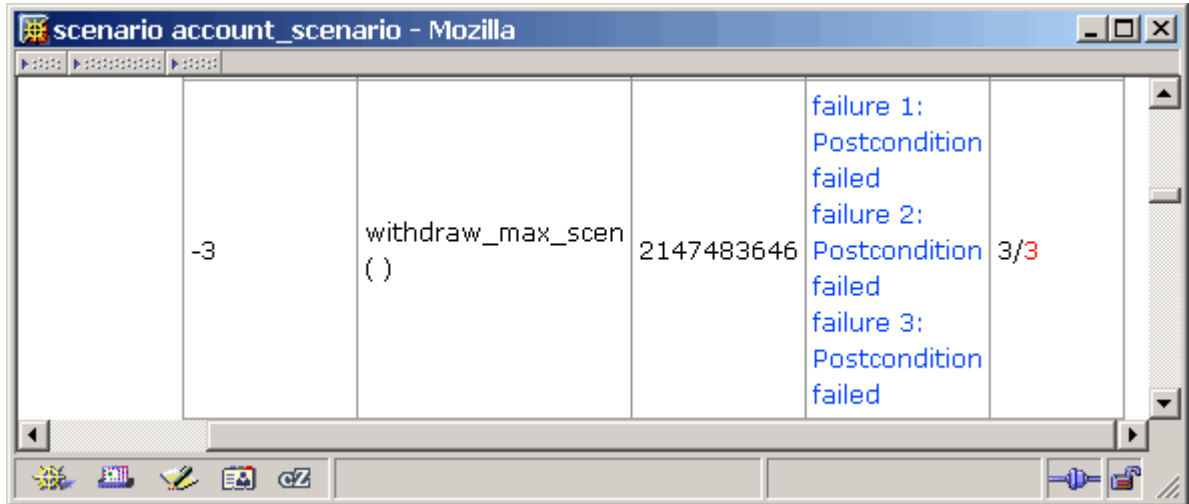


Рисунок 11. Нарушение в отчете сценария.

В этом отчете зафиксировано, что в состоянии при текущем балансе равным `-3` после вызова `withdraw_scen` получено конечное состояние с текущим балансом `2147483646`. То есть, снятие максимально возможной суммы со счета с отрицательным балансом, приводит к счету с очень большим балансом. Хотя из постусловия `withdraw_spec` следует, что в этом случае `withdraw` должно не изменять баланс и возвращать ноль:

```
post {
    if (balance >= sum - MaximalCredit)
        return balance == @balance - sum && withdraw_spec == sum;
    else
        return balance == @balance && withdraw_spec == 0;
}
```

Таким образом получена полная информация о нарушении, из которой следует, что обнаружена ошибка в реализации.

Реализация находится в файле `account.c` каталога `examples\account` в дереве каталогов установки CTesK. Код реализации функции `withdraw` имеет следующий вид:

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

Из кода видно, что при отрицательном значении `acct->balance` и значении `sum` равным, например, `INT_MAX`, происходит переполнение в операции вычитания. Правильный код реализации `withdraw` приведен ниже.

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

В этом случае переполнения не происходит и функция работает в соответствии с требованиями.

Соберите тест с исправленной реализацией и сгенерируйте заново отчеты. В отчетах должно быть показано, что нарушений не найдено и покрытие обеих функций равно 100%.

Приложение А: Использование CTesK с компилятором GCC

Транслятор SeC расположен в поддиректории **bin** установочной директории CTesK. Транслятор можно запустить используя следующую команду:

```
> sec.sh [опции] <sec файл> <с файл>
```

Он транслирует файл SeC <sec файл> в файл C <с файл>.

Опция **--sei** <sei файл> устанавливает используемый промежуточный файл. Все остальные опции передаются препроцессору.

Использование утилиты GNU Make для сборки теста

Для сборки теста разработанного с помощью системы CTesK можно использовать утилиту GNU Make. Для упрощения создания файлов сборки (makefiles) можно использовать шаблон содержащийся в примерах CTesK. Он находится в файле **examples/example.make**.

Для его использования, файл для сборки (с именем **Makefile** или **GNUmakefile**) должен быть создан в той директории, в которой находятся тесты.

В этом файле должны быть определены следующие переменные:

sec_sources

Эта переменная должна содержать список **.sec** файлов разделенных пробелами, разработанных для тестирования.

c_sources

Эта переменная должна содержать список **.c** файлов разделенных пробелами, которые должны быть слинкованы с тестами.

example

Эта переменная должна содержать имя исполняемого файла теста.

Затем нужно включить файл **example.make** находящийся в примерах CTesK используя директиву **include**:

```
include $(CTESK_HOME)/examples/example.make
```

После этого нужно запустить утилиту GNU Make используя программу **make** или **gmake** (в зависимости от установленной среды разработки).

Переменные **XINCLUDE** и **XLIB** позволяют определить дополнительные пути для включения заголовочных файлов (**-I<path>**) и дополнительные библиотеки и пути к ним (**-l<lib>** и **-L<path>**).

Примеры файлов сборки можно найти в директориях **examples/account**, **examples/pqueue**, и **examples/stack**.

Выполнение тестов

В директории содержащей исполняемый файл теста, запустите следующую команду

> **<test file>** [**<trace options>**] [**<options>**]

<test file> — исполняемый файл теста. **<trace options>** — опции трассировки теста. **<options>** — опции запуска, определяемые пользователем при разработке теста. Трассировка теста изменяется с помощью следующих опций:

-t <trace file> — трасса будет направлена в файл **<trace file>**;

-tc — трасса будет выводиться на консоль;

-tt — трасса будет направлена в файл **<scenario name>--YY-MM-DD--HH-MM-SS.utt**, где **YY-MM-DD--HH-MM-SS** текущая дата и время.

Генерация тестового отчета

Генератор тестовых отчетов CTesK расположен в поддиректории **bin** установочной директории CTesK. Для генерации отчета о выполнении теста в удобном для чтения виде, запустите следующую команду

>**ctestk-rg.sh -d <trace directory> <trace file>**

В результате будет сгенерирован HTML отчет в директории **<trace directory>**.

Откройте файл **index.html** в этой директории. Вы увидите стартовую страницу HTML отчета.

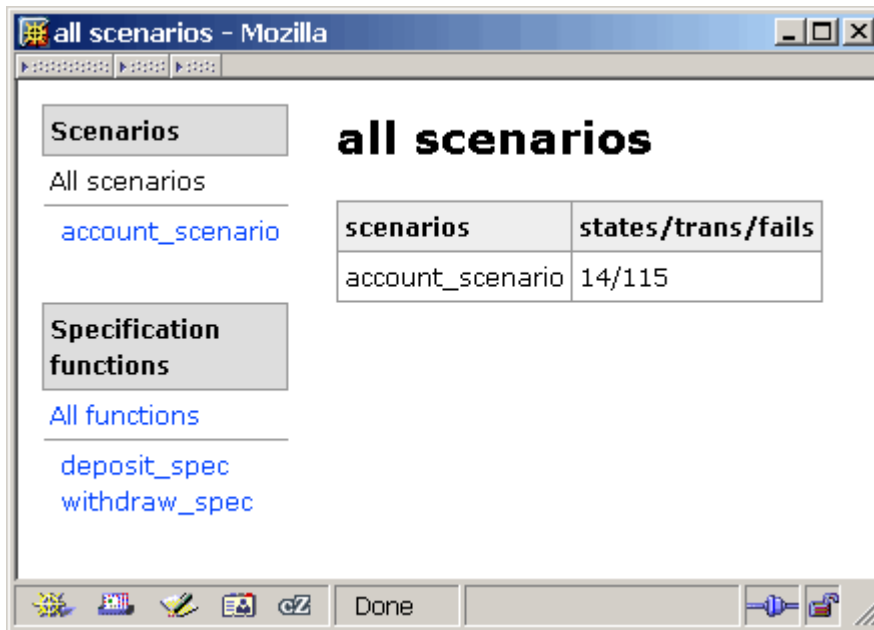


Рисунок 12. Стартовая страница HTML отчета.