CTesK 2.2 для Microsoft Visual C++® 6.0: Быстрое знакомство

Содержание

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, для программного обеспечения, реализованного 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 (*S*pecification *E*xtension 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: *“CTesK 2.2: Руководство пользователя ”* и *“CTesK 2.2: 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 ctesk@ispras.ru.

Пример тестируемой системы: Банковский кредитный счет

Предполагается, что на Вашем компьютере установлена среда разработки Microsoft Visual C++® 6.0 (в дальнейшем просто *среда разработки*) и инструмент CTesK 2.2, который интегрирован в нее. Если CTesK 2.2 не установлен или установлен без интеграции в среду разработки, установите его должным образом, следуя указаниям из документа "*CTesK 2.2: Инструкция по установке и использованию*".

В текущем документе (*"CTesK 2.2 для Microsoft Visual C++® 6.0: Быстрое знакомство"*) 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** дерева каталогов установки CTesK:

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

Допустимый размер кредита должен быть не меньше нуля и определяется макросом `MAXIMUM_CREDIT` в файле **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, функция возвращает `sum`, иначе — `0`.

Далее в документе описываются предварительные действия в среде разработки и демонстрируется как, используя CTesK, разработать тест для системы банковского кредитного счета, выполнить тестирование и проанализировать полученные результаты:

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

Предварительные действия

Запустите среду разработки и откройте файл проекта (workspace) **account.dsw** из каталога **examples\account** дерева каталогов установки CTesK. Для этого выберите пункт меню '**File\Open Workspace...**'. В появившемся окне '**Open Workspace**' зайдите в нужный каталог, выберите файл **account.dsw** и нажмите кнопку '**Open**'.

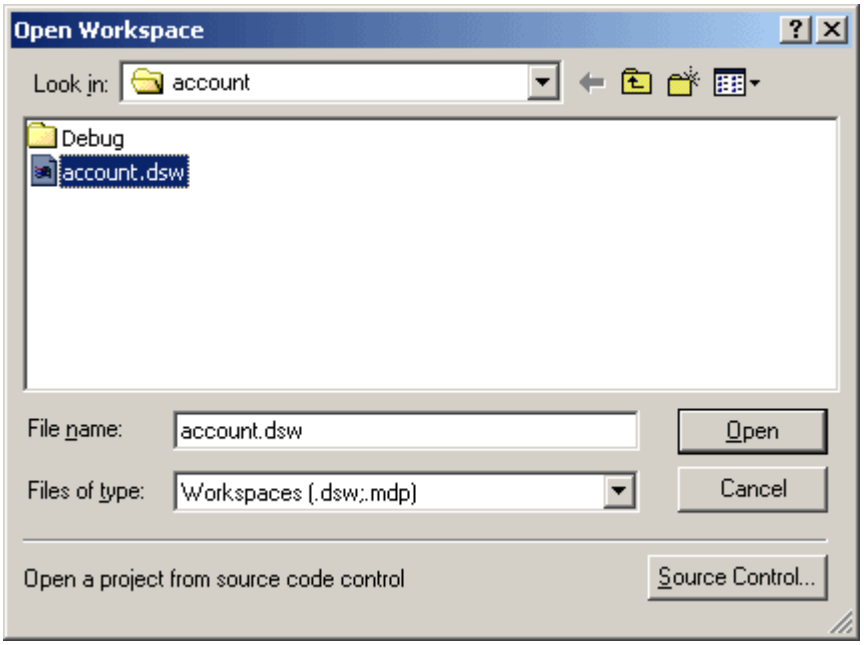

Рисунок 1. Открытие проекта account.

После выполнения указанных действий закладка '**FileView**' окна '**Workspace**', отображающая файлы проекта, должна выглядеть как показано на следующем рисунке.

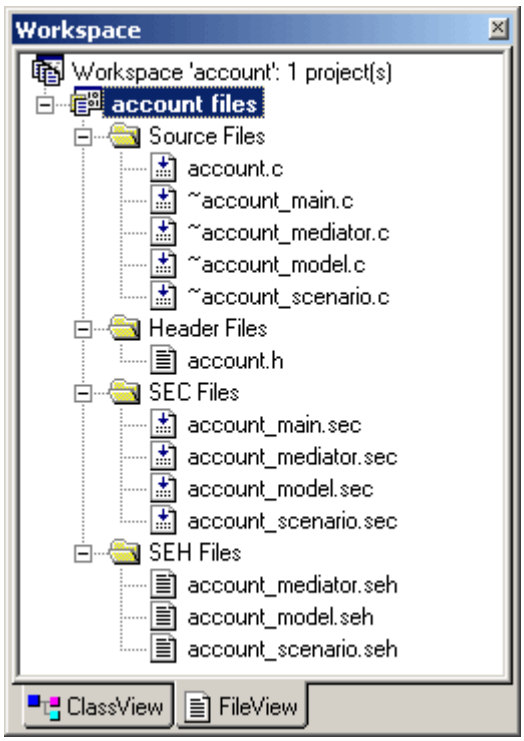

Рисунок 2. Файлы проекта account.

Все файлы проекта делятся на четыре группы:

- **Source files** — файлы, содержащие код на языке C. Это могут быть файлы, содержащие исходный код тестируемой системы, вспомогательные библиотеки и т.д. Также к этой группе относятся специальные файлы, сгенерированные CTesK, такие файлы начинаются с символа ~;
- **Header files** — заголовочные файлы, содержащие код на языке C;
- **SEC Files** — файлы, содержащие код на языке SeC. Такие файлы имеют расширение **.sec**. Они содержат компоненты теста: спецификации, медиаторы, тестовые сценарии;
- **SEH Files** — заголовочные файлы, содержащие код на языке SeC. Такие файлы имеют расширение **.seh**. Они служат для описания интерфейсов компонентов теста.

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 требованиям и реальным поведением интерфейсных функций тестируемой системы.

В CTesK формальные спецификации разрабатываются на специальном языке SeC[1], 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.

Спецификация системы кредитного счета находится в файлах **account_model.seh** и **account_model.sec**. Откройте файл **account_model.seh**. Для этого дважды нажмите левой кнопкой мыши на соответствующем узле в окне '**Workspace**'.

Исключая комментарии и некоторые вспомогательные макросы, файл **account_model.seh** выглядит следующим образом:

```
#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 начале файла включаются файлы **limits.h** и **account.h**, объявляется внешняя *переменная с инвариантом* и описываются объявления *типа с инвариантом* и *спецификационных функций*.

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

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

[1] Произносится `[sek]`.

```
#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 неменьше 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 файле **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 выполнение следующих условий: второй параметр должен быть больше нуля и в сумме с балансом, переданного счета, должен быть небольше 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 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 массиве[3]. 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 небольше 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 на входные значения параметров: указатель `acct` не должен быть равен нулю и сумма снимаемых денег `sum` должна быть положительной.

В блоке **`coverage`** `C` 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 C.

Для запуска трансляции в среде разработки, откройте файл **account_model.sec**, если он не был открыт, и выберите пункт меню '**Build\Compile account_model.sec**' или нажмите комбинацию клавиш **Ctrl+F7**.

В результате в каталоге **examples\account** сгенерируется файл ~**account_model.c**.

Медиатор для системы Банковского кредитного счета

В проекте **account** есть реализация системы Банковского кредитного счета, которую нужно протестировать, и ее спецификация. Для того чтобы дать тесту возможность проверить соответствие между реализацией и спецификацией, их нужно каким-то образом связать. В качестве связок используются специальные компоненты тестовой системы, называемые *медиаторами*.

Медиаторы оформляются в виде набора специальных функций, называемых *медиаторными функциями*.

В проекте **account** уже есть готовый медиатор, определенный в файле **account_mediator.sec** каталога **examples\account** дерева установки CTesK, поэтому вы можете пропустить описание того, как создать новый медиатор, и приступить к следующему разделу.

Для создания шаблона нового медиатора в среде разработки запустите мастер '**CTesK Mediator Wizard**', для этого нажмите кнопку 'C^{T}_{μ}', расположенную на панели инструментов.

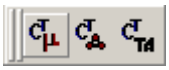

Рисунок 3. Кнопка запуска мастера создания шаблона нового медиатора.

На следующем шаге ('**SEH Files**') в появившемся окне мастера вы должны выбрать SEH файлы, содержащие объявления спецификационных функций, описывающих функциональность тестируемой системы, которую нужно протестировать.

Выделите в списке '**All project SEH files**' файл **account_model.seh** и нажмите кнопку '**Add >**'. В результате выделенный файл переместится в список '**Selected SEH files**' как показано на следующем рисунке. После этого нажмите кнопку '**Next >**'.

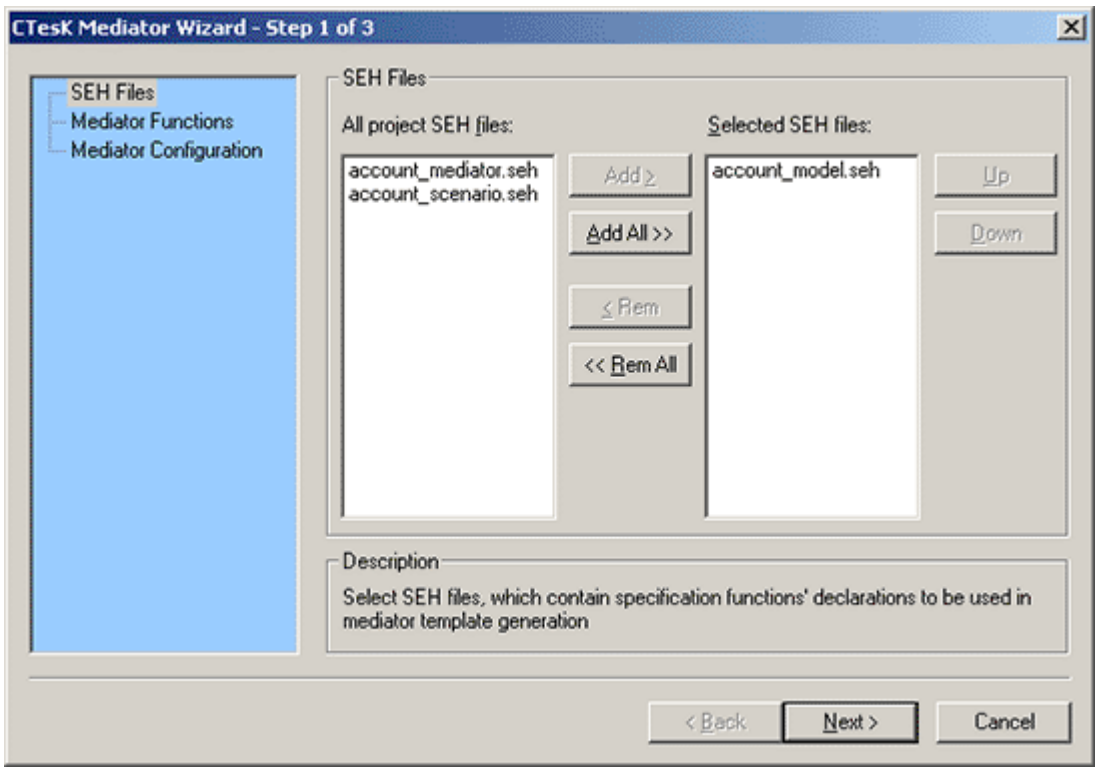

Рисунок 4. Выбор SEH файлов, содержащих объявления спецификационных функций.

На следующем шаге ('**Mediator Functions**') вы должны выбрать спецификационные функции, которые собираетесь протестировать через создаваемый медиатор и для каждой выбранной функции задать имя соответствующей ей медиаторной функции.

Оставьте панель без изменений и нажмите кнопку '**Next >**'.

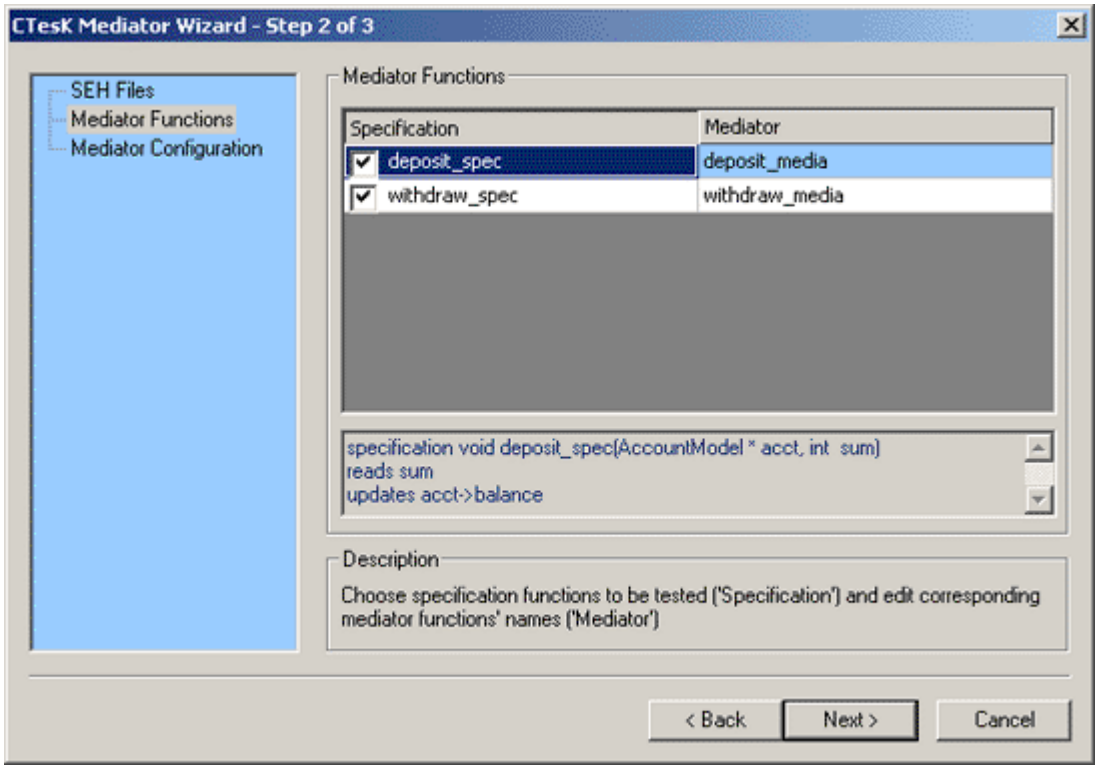

Рисунок 5. Выбор тестируемых спецификационных функций и задание имен соответствующих медиаторных функций.

На следующем шаге ('**Mediator Configuration**') вы должны выбрать тип реализации тестируемой системы и указать имя SEH и SEC файлов, в которых сохранится шаблон медиатора.

Оставьте выбранным тип '**Open state implementation**' (в этом случае синхронизация состояний реализации и спецификации будет оформлена в одной функции) и введите **my_account_mediator** в текстовое поле '**Mediator file names**'. После этого нажмите кнопку '**Finish**'.

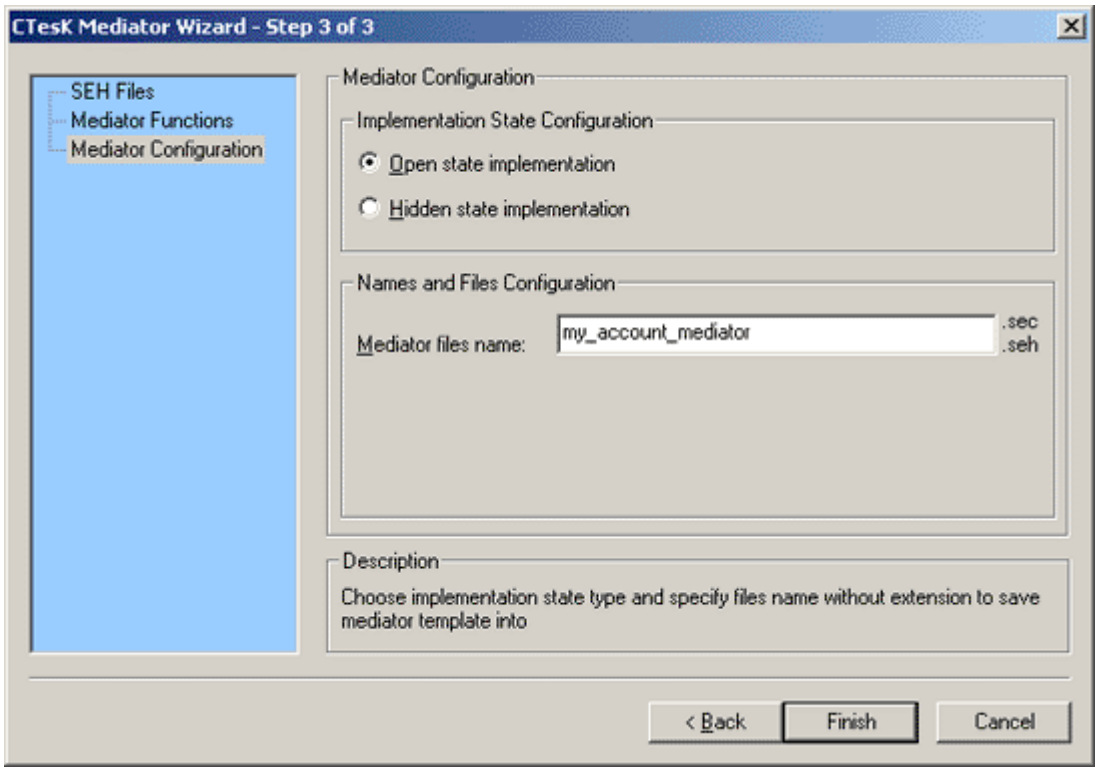

Рисунок 6. Выбор типа реализации и указание имени файлов для сохранения результата.

В результате будут созданы два файла **my_account_mediator.seh** и **my_account_mediator.sec**, содержащие шаблон медиатора. Файлы автоматически добавятся в текущий проект. В редакторе откроется файл **my_account_mediator.sec**, который требует дальнейшей доработки.

В файле **my_account_mediator.sec** определены шаблоны для следующих функций:

- функции синхронизации состояний реализации и спецификации `map_state_up_my_account_mediator`;
- двух медиаторных функций: `deposit_media` и `withdraw_media`: первая связывает спецификационную функцию `deposit_spec` с интерфейсной функцией реализации `deposit`, вторая — `withdraw_spec` с `withdraw`.

Чтобы получить работающий медиатор выполните следующие действия:

- определите функцию синхронизации состояний `map_state_up_my_account_mediator`;
- добавьте вызовы интерфейсных функций реализации в определения медиаторных функций `deposit_media` и `withdraw_media`.

В данном примере синхронизация состояний не нужна: указатель на состояние передается функциям реализации как один из аргументов. Оставьте функцию `map_state_up_my_account_mediator` без изменений:

```
static void map_state_up_my_account_mediator ()
{
  // TODO: Add state synchronization actions here
}
```

Рассмотрим шаблон медиаторной функции `deposit_media`.

Определение функции начинается с ключевого слова **`mediator`**, затем следует имя медиаторной функции, ключевое слово **`for`** и сигнатура спецификационной функции вместе с ограничениями доступа.

В теле медиаторной функции присутствует блок, помеченный ключевым словом **`call`**. Этот блок должен содержать реализацию функциональности, описанной в спецификационной функции, при помощи вызова соответствующей интерфейсной функции реализации.

Добавьте в блок **`call`** вызов интерфейсной функции реализации `deposit` с параметрами `acct` и `sum`:

```
mediator deposit_media for
specification void deposit_spec(AccountModel * acct, int  sum)
reads sum
updates acct->balance
{
  call
  {
    // TODO: Add implementation function call here
    deposit(acct, sum);
  }
  state
  {
    map_state_up_my_account_mediator ();
  }
}
```

Проделайте то же самое с шаблоном медиаторной функции `withdraw_media`: добавьте в блок **`call`** вызов интерфейсной функции реализации `withdraw` с параметрами `acct` и `sum`. Не забудьте написать **`return`**:

```
mediator withdraw_media for
specification int withdraw_spec(AccountModel * acct, int  sum)
reads sum
updates acct->balance
{
  call
  {
    // TODO: Add implementation function call here
    return withdraw(acct, sum);
  }
  state
  {
    map_state_up_my_account_mediator ();
  }
}
```

Убедитесь, что полученный файл успешно транслируется в код на языке C. Для трансляции выберите пункт меню '**Build\Compile my_account_mediator.sec**' или нажмите комбинацию клавиш **Ctrl+F7**.

В результате в каталоге **examples\account** сгенерируется файл **~my_account_mediator.c**.

Тестовый сценарий для системы Банковского кредитного счета

Спецификации являются формальным описанием функциональных требований к тестируемой системе. Из них генерируются компоненты, проверяющие правильность отдельных вызовов интерфейсных функций системы. Медиаторы связывают спецификации и тестируемую реализацию. Чтобы проверить поведение системы в различных ситуациях, на нее нужно оказать последовательность воздействий, вызывая различные интерфейсные функции с различными значениями параметров.

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

Тестовый сценарий состоит из двух основных частей:

- функция вычисления *обобщенного состояния*, используемого для уменьшения числа тестируемых ситуаций;
- *сценарные функции*, описывающие перебор аргументов вызовов спецификационных функций.

В проекте **account** уже есть готовый тестовый сценарий, определенный в файле **account_scenario.sec** каталога **examples\account** дерева установки CTesK, поэтому вы можете пропустить описание того, как создать новый тестовый сценарий, и приступить к следующему разделу.

Для создания шаблона нового тестового сценария в среде разработки запустите мастер '**CTesK Scenario Wizard**', для этого нажмите кнопку '$\mathbf{C^T_\Delta}$', расположенную на панели инструментов.

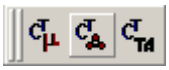

Рисунок 7. Кнопка запуска мастера создания шаблона нового тестового сценария.

На следующем шаге ('**SEH Files**') в появившемся окне мастера вы должны выбрать SEH файлы, содержащие объявления спецификационных функций, описывающих функциональность тестируемой системы, которую нужно протестировать, и объявления медиаторных функций, связывающих спецификацию и тестируемую реализацию.

Выделите в списке '**All project SEH files**' файлы **account_model.seh** и **my_account_mediator.seh** и нажмите кнопку '**Add >**'. В результате выделенные файлы переместятся в список '**Selected SEH files**' как показано на следующем рисунке. После этого нажмите кнопку '**Next >**'.

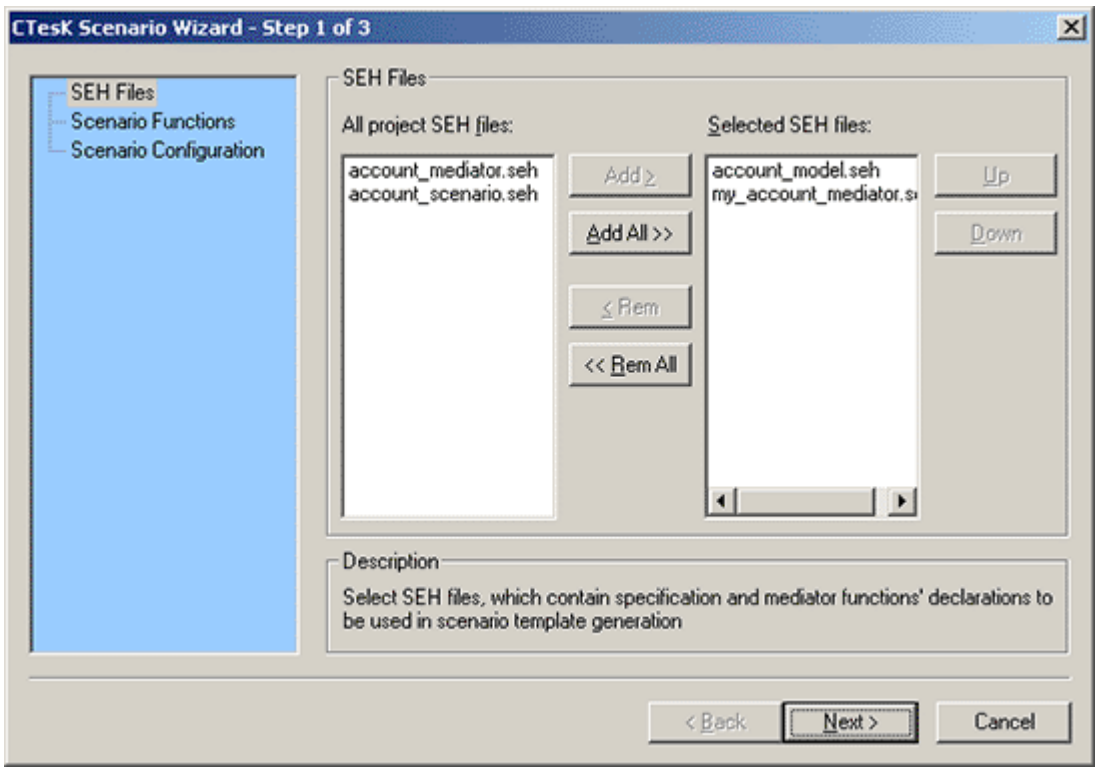

Рисунок 8. Выбор SEH файлов, содержащих объявления спецификационных и медиаторных функций.

На следующем шаге ('**Scenario Functions**') вы должны выбрать спецификационные функции, которые собираетесь протестировать, для каждой выбранной функции задать имя соответствующей ей сценарной функции, выбрать подходящую медиаторную функцию и включить, если это нужно, фильтрацию по тестовому покрытию.

Оставьте панель без изменений и нажмите кнопку '**Next >**'.

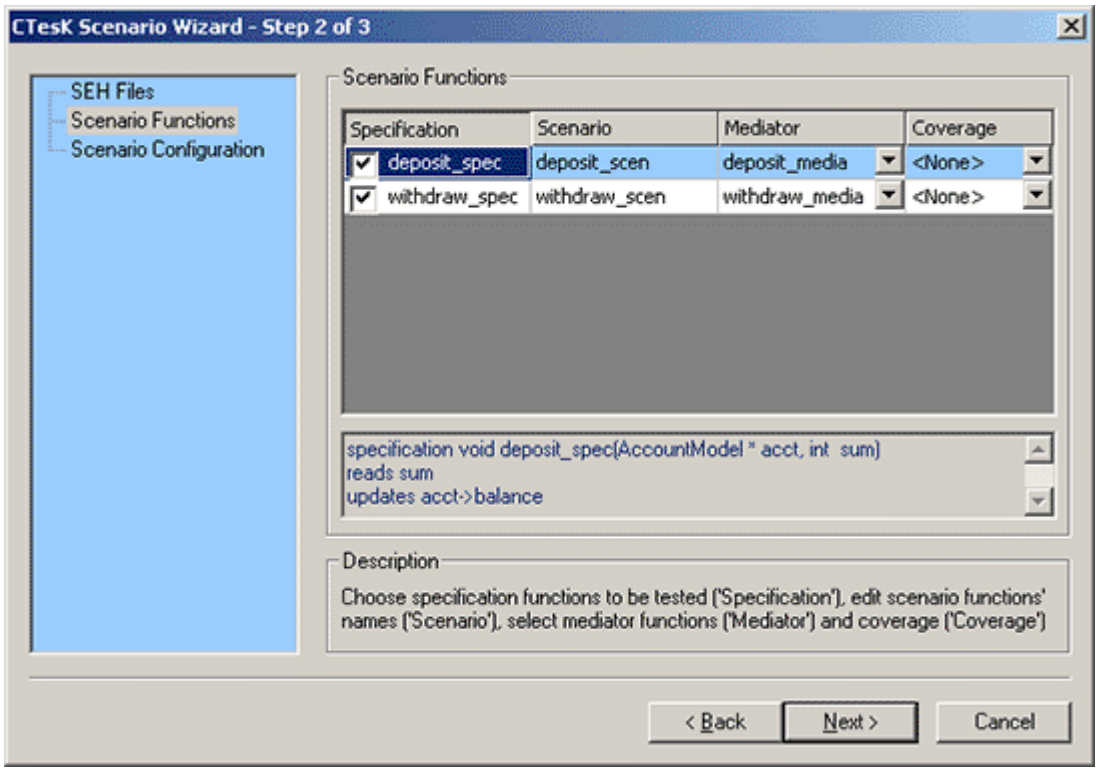

Рисунок 9. Выбор тестируемых спецификационных функций, задание имен соответствующих сценарных функций и установка других параметров.

На следующем шаге ('**Scenario Configuration**') вы должны указать тип и параметры построения обобщенного состояния тестового сценария и имена SEH и SEC файлов, в которых сохранится шаблон тестового сценария.

Мы в качестве обобщенного состояния будем использовать баланс счета. Так как баланс счета является целым числом, выберите элемент '**Integer**' в выпадающем списке '**Scenario state type**'. Введите **my_account_scenario** в текстовое поле '**Scenario files name**'. Поставьте галочку '**Into the separate files**' и введите **my_account_main** в текстовое поле '**Main files name**'. После этого нажмите кнопку '**Finish**'.

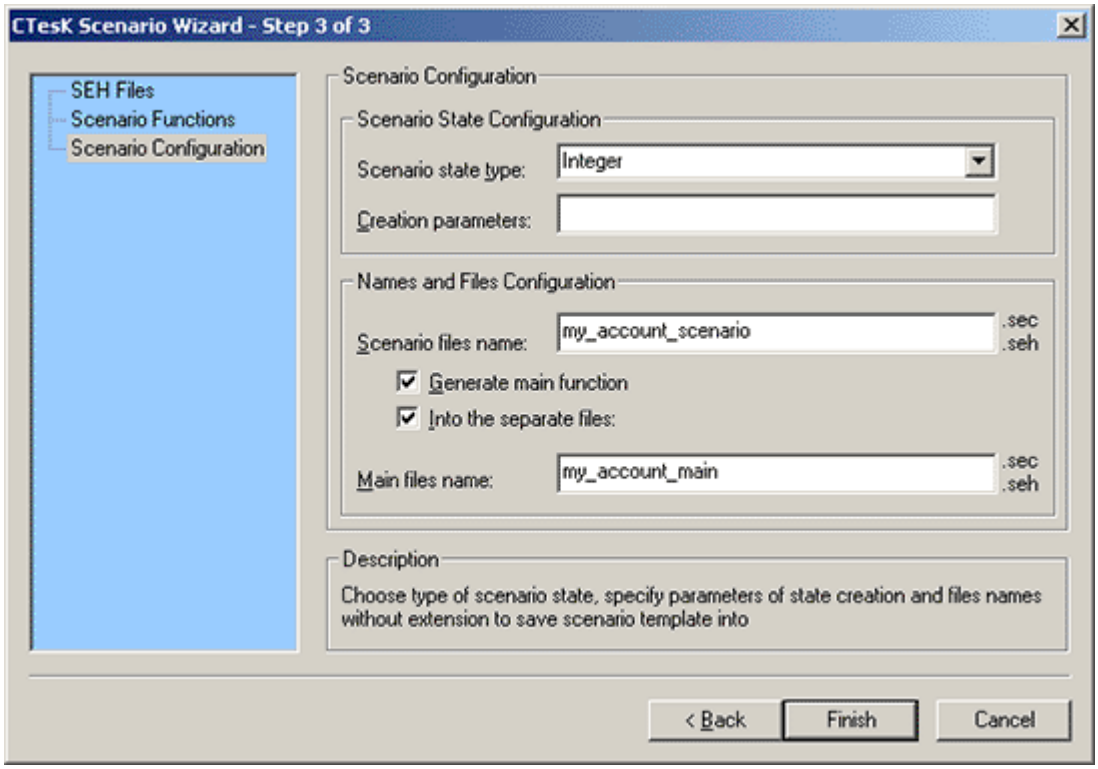

Рисунок 10. Задание типа и параметров построения обобщенного состояния тестового сценария и указание имен файлов для сохранения результатов.

В результате будут созданы четыре файла **my_account_scenario.seh**, **my_account_scenario.sec**, **my_account_main.seh** и **my_account_main.sec**, содержащие шаблон тестового сценария и функцию **main**, осуществляющую запуск теста. Файлы автоматически добавятся в текущий проект. В редакторе откроется файл **my_account_scenario.sec**, который требует дальнейшей доработки.

В файле **my_account_scenario.sec** определены шаблоны для следующих функций:

- функции инициализации тестового сценария `my_account_scenario_init`;
- функции завершения тестового сценария `my_account_scenario_finish`;
- функции вычисления обобщенного состояния `my_account_scenario_state`;
- двух сценарных функций `deposit_scen` и `withdraw_scen`: первая функция описывает перебор аргументов вызова спецификационной функции `deposit_spec`, вторая — `withdraw_spec`.

Чтобы получить работающий тестовый сценарий выполните следующие действия:

- определите спецификационное состояние тестового сценария, то есть набор данных, хранящих информацию о истории изменения счета;
- определите функцию инициализации тестового сценария `my_account_scenario_init`;

- определите функцию завершения тестового сценария `my_account_scenario_finish`;
- определите функцию вычисления обобщенного состояния `my_account_scenario_state`;
- добавьте перебор аргументов вызовов спецификационных функций в сценарные функции `deposit_scen` и `withdraw_scen`.

Спецификационные функции `deposit_spec` и `withdraw_spec` получают в качестве параметра указатель на переменную типа `AccountModel`, в которой хранится баланс счета. Мы будем при тестировании использовать один счет `acct`. Добавьте объявление переменной `acct` типа `AccountModel` в начало файла:

```
// TODO: Add specification state definition here
AccountModel acct;
```

После этого добавьте действия по инициализации состояния счета в определение функции `my_account_scenario_init`: вставьте присваивание нулевого значения полю `balance` переменной `acct`:

```
bool my_account_scenario_init (int argc, char **argv)
{
  // TODO: Add scenario initialization actions here
  acct.balance = 0;
  return true;
}
```

Оставьте без изменения функцию завершения тестового сценария `my_account_scenario_finish`:

```
void my_account_scenario_finish ()
{
  // TODO: Add scenario finalization actions here
}
```

Как отмечалось выше, в качестве обобщенного состояния тестового сценария мы будем использовать баланс счета, поэтому добавьте в функцию `my_account_scenario_state`, параметр построения обобщенного состояния `acct.balance`:

```
Object *my_account_scenario_state ()
{
  return create (&type_Integer /* TODO: Add scenario generalized
    state creation parameters here */, acct.balance);
}
```

Рассмотрим шаблоны сценарных функций `deposit_scen` и `withdraw_media`.

После типа возвращаемого значения, которое всегда должно быть **`bool`**, следует ключевое слово **`scenario`**, затем – имя сценарной функции. Сценарная функция не должна иметь параметров.

Добавьте в сценарную функцию `deposit_scen` перебор значений суммы вклада от `1` до `5` включительно, если баланс счета не превосходит `5`. Используйте конструкцию **`iterate`**:

```
/*
specification void deposit_spec(AccountModel * acct, int  sum)
reads sum
updates acct->balance
*/
bool scenario deposit_scen ()
{
  // TODO: Add cycles for parameters iteration here
  if (acct.balance <= 5)
  {
    iterate (int i = 1; i <= 5; i++;)
      deposit_spec (/* TODO: Add parameters values here */ &acct, i);
  }
  return true;
}
```

Добавьте в сценарную функцию `withdraw_scen` перебор значений снимаемой суммы от `1` до `5`:

```
/*
specification int withdraw_spec(AccountModel * acct, int  sum)
reads sum
updates acct->balance
*/
bool scenario withdraw_scen ()
{
  // TODO: Add cycles for parameters iteration here
  iterate (int i = 1; i <= 5; i++;)
    withdraw_spec (/* TODO: Add parameters values here */ &acct, i);
  return true;
}
```

Убедитесь, что полученный файл успешно транслируется в код на языке C. Для трансляции выберите пункт меню '**Build\Compile my_account_scenario.sec**' или нажмите комбинацию клавиш **Ctrl+F7**.

В результате в каталоге **examples\account** сгенерируется файл **~my_account_scenario.c**.

Выполнение теста для системы Банковского кредитного счета

Последний компонент теста для системы банковского кредитного счета находится в файле **my_account_main.sec**, который был сгенерирован мастером создания шабона нового тестового сценария (см. раздел *"Тестовый сценарий для системы Банковского кредитного счета"*). В этом файле содержится определение функции `main` тестовой программы.

```
#include "my_account_main.seh"

#include "account_model.seh"
#include "my_account_mediator.seh"
#include "my_account_scenario.seh"

void my_account_main (int argc, char **argv)
{
  set_mediator_deposit_spec (deposit_media);
  set_mediator_withdraw_spec (withdraw_media);

  my_account_scenario (argc, argv);
}

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

Включаемый заголовочный файл **my_account_scenario.seh** содержит декларацию сценарной переменной:

```
scenario dfsm my_account_scenario;
```

Функция `main` вызывает функцию `my_account_main`, передавая ей опции командой строки в качестве аргументов. В свою очередь функция `my_account_main` устанавливает медиаторные функции для спецификационных функций `deposit_spec` и `withdraw_spec`:

```
set_mediator_deposit_spec (deposit_media);
set_mediator_withdraw_spec (withdraw_media);
```

Для запуска трансляции в среде разработки откройте файл **my_account_main.sec**, если он не был открыт, выберите пункт меню '**Build\Compile my_account_main.sec**' или нажмите комбинацию клавиш **Ctrl+F7**.

В результате в каталоге **examples\account** сгенерируется файл **~my_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**, **~my_account_mediator.c**, **~my_account_scenario.c** и **~my_account_main.c**.

Удалите из проекта лишние файлы — **account_mediator.seh**, **account_mediator.sec**, **~account_mediator.c**, **account_scenario.seh**, **account_scenario.sec**, **~account_scenario.c**, **account_main.seh**, **account_main.sec** и **~account_main.c**. Для этого выделите данные файлы в окне '**Workspace**' (закладка '**FileView**') и нажмите клавишу '**Delete**'.

Эти файлы должны быть скомпилированы в объектные файлы, которые должны быть собраны в исполняемый файл.

Для сборки исполняемого файла в среде разработки, выберите пункт меню '**Build\Build account.exe**' или нажмите клавишу **F7**.

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

Опции тестовой программы передаются из функции `main` в тестовый сценарий. Обходчик типа `dfsm` обрабатывает следующие стандартные опции[4]:

-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 тестового сценария.

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

Для задания опций запуска теста в среде разработки, выберите пункт меню '**Project\Settings…**' или нажмите комбинацию клавиш **Alt+F7**, выберите закладку '**Debug**', введите в категории '**General**' в поле ввода аргументов программы '**Program arguments**' строку '**-t trace.xml**', и нажмите кнопку '**OK**'.

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

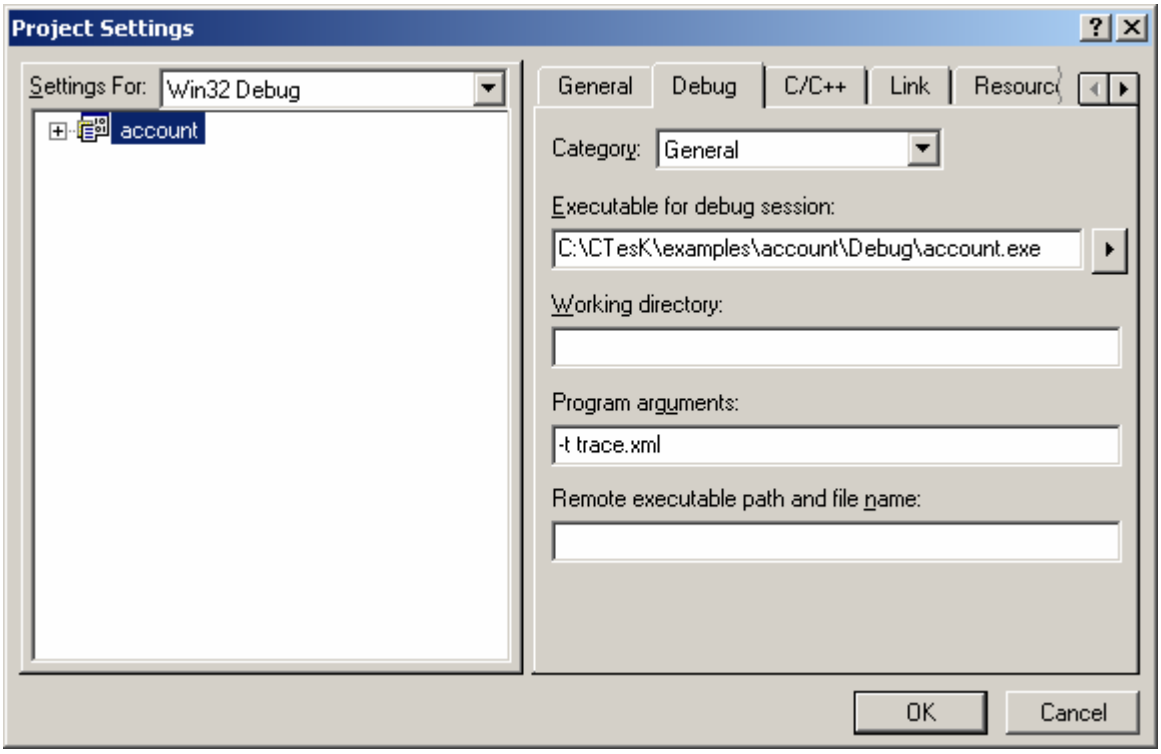

Рисунок 11. Задание опций запуска теста.

После этого, для запуска теста выберите пункт меню '**Build\Execute account.exe**' или нажмите комбинацию клавиш **Ctrl+F5**.

В результате выполнения теста в каталоге **examples\account** сгенерируется файл **trace.xml**.

Добавьте этот файл в проект. Для этого выберите пункт меню '**Project\Add To Project ►\Files...**'. В появившемся окне '**Insert Files into Project**' зайдите в нужный каталог, выберите тип файлов '**All Files (*.*)**' в выпадающем списке '**Files of type**', выберите файл **trace.xml** и нажмите кнопку '**OK**'.

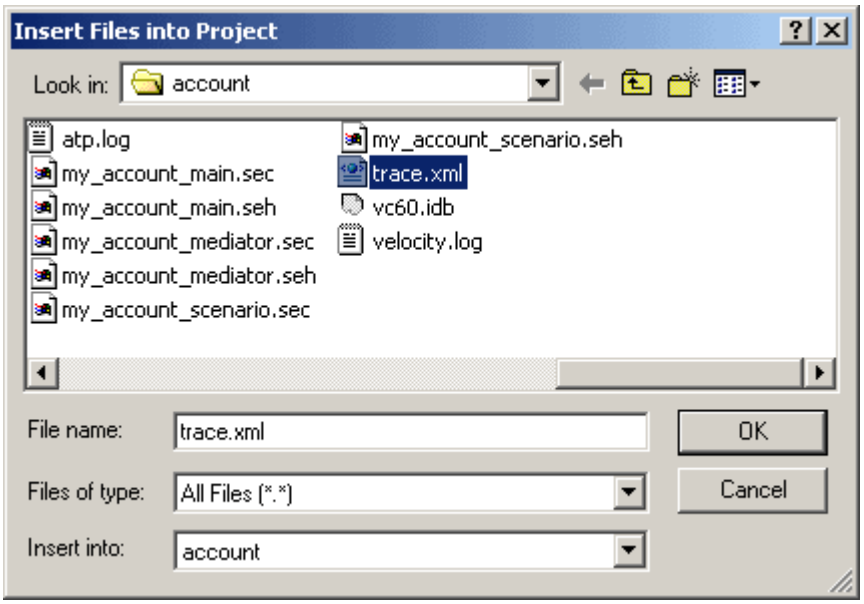

Рисунок 12. Добавление файла трассы в проект account.

Анализ результатов выполнения теста для системы Банковского кредитного счета

Генерация тестовых отчетов

Для создания отчетов в среде разработки откройте файл **trace.xml** и запустите анализатор трассы, для этого нажмите кнопку **C$^{T}_{TA}$**, расположенную на панели инструментов.

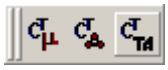

Рисунок 13. Кнопка запуска анализатора трассы.

В результате в каталоге **examples\account**, содержащем файл с трассой теста, создастся подкаталог, с тестовыми отчетами. Тестовые отчеты представляют набор взаимосвязанных HTML документов.

По окончании генерации автоматически запустится программа просмотра. В левом фрейме главной страницы располагается навигационный список отчетов.

Итоговый отчет

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

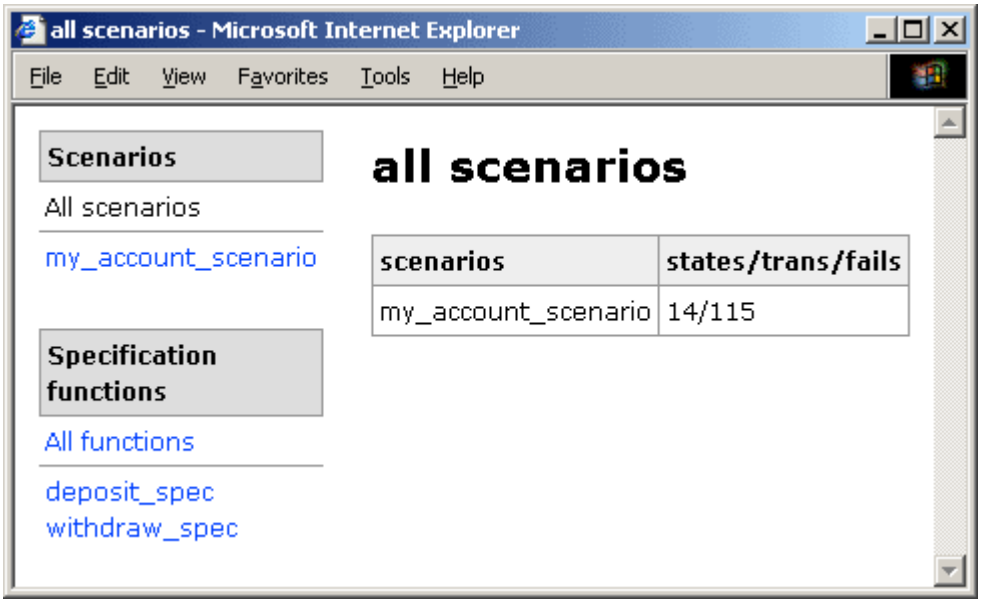

Рисунок 14. Итоговый отчет.

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.

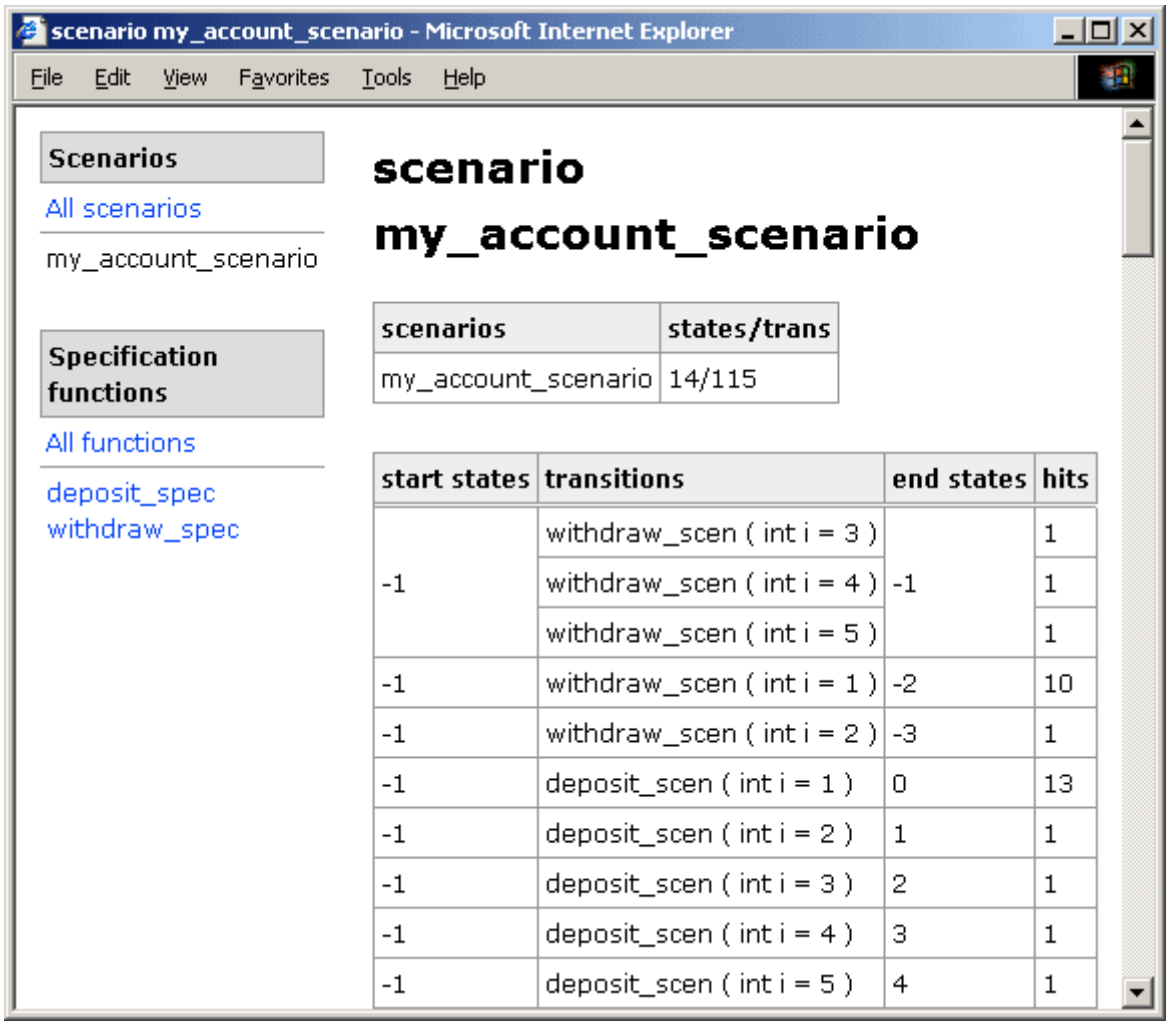

Рисунок 15. Подробный отчет сценария.

Так при выполнении теста по сценарию `my_account_scenario` 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.

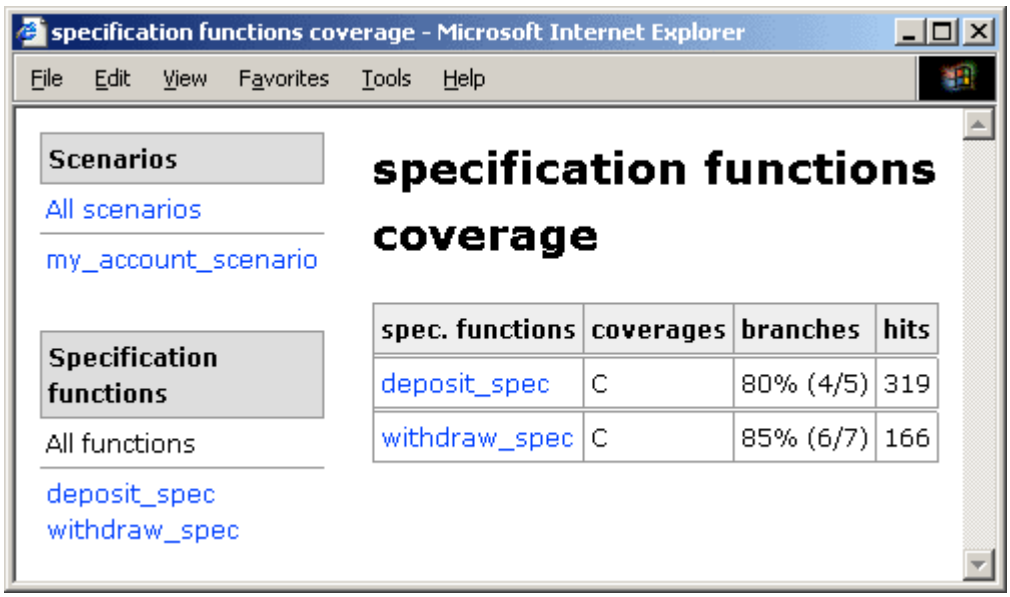

Рисунок 16. Итоговый отчет по покрытию функций.

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.

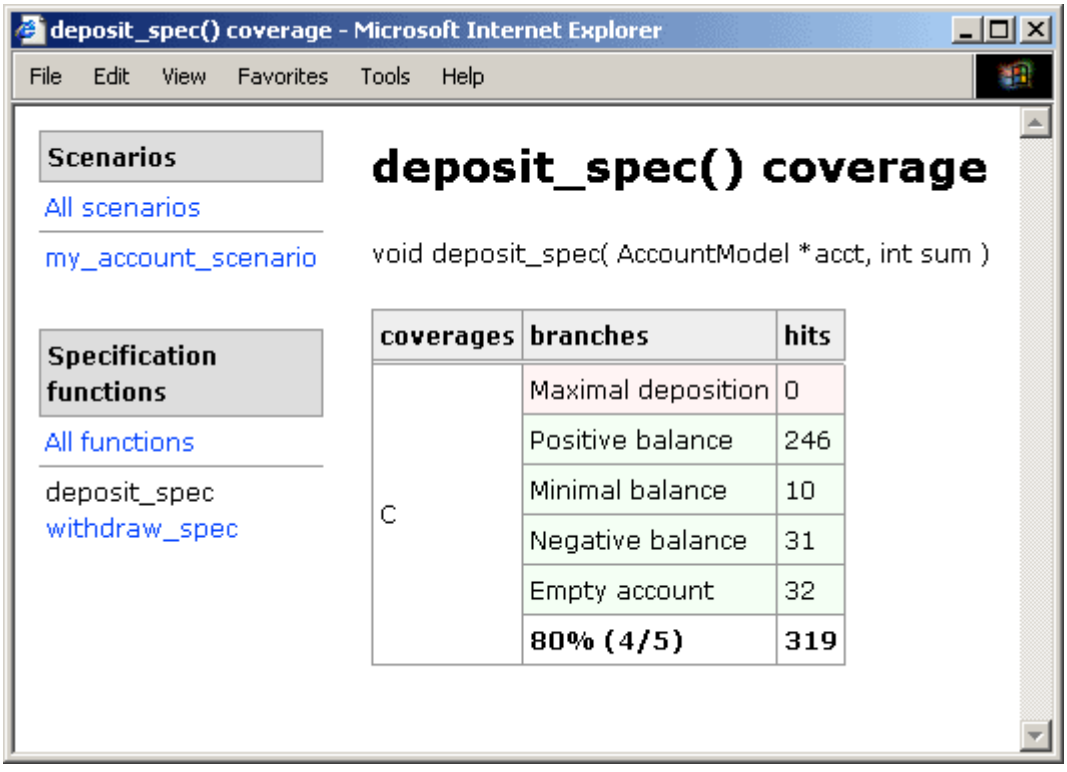

Рисунок 17. Подробный отчет по покрытию функции deposit_spec.

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.

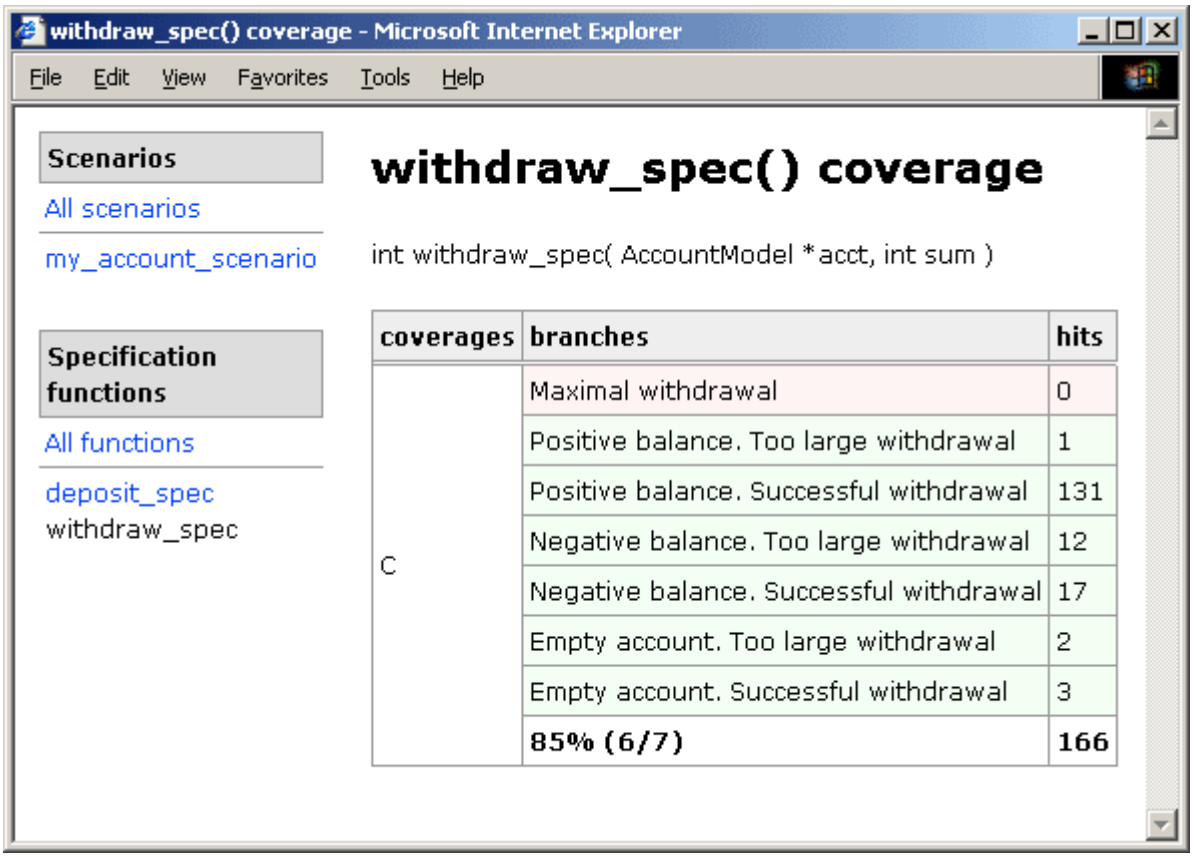

Рисунок 18. Подробный отчет по покрытию функции withdraw_spec.

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` определите в файле **my_account_scenario.sec** сценарные функции `deposit_max_scen` и `withdraw_max_scen`, которые поставляют значения параметров, при которых происходит вкладывание на счет максимально допустимой суммы и попытка снятия со счета суммы равной максимальному значению типа `int`:

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

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

Измените инициализацию сценарной переменной my_account_scenario. Добавьте сценарные функции `deposit_max_scen` и `withdraw_max_scen` в массив, инициализирующий поле `actions`:

```
scenario dfsm my_account_scenario =
{
  .init       = my_account_scenario_init,
  .finish     = my_account_scenario_finish,
  .getState   = my_account_scenario_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`, добавив условие на баланс:

```
/*
specification int withdraw_spec(AccountModel * acct, int  sum)
reads sum
updates acct->balance
*/
bool scenario withdraw_scen ()
{
  // TODO: Add cycles for parameters iteration here
  if (acct.balance <= 5)
  {
    iterate (int i = 1; i <= 5; i++;)
      withdraw_spec (/* TODO: Add parameters values here */ &acct, i);
  }
  return true;
}
```

Теперь при значениях баланса больших `5` будут вызваться только две новые функции.

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

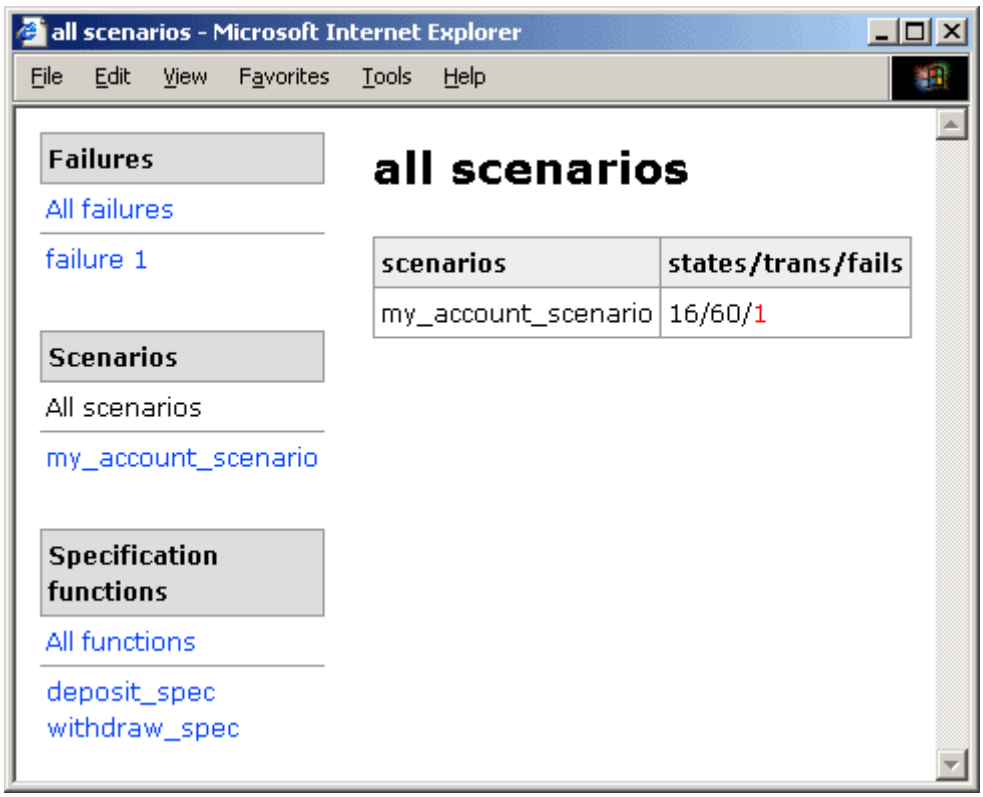

Рисунок 19. Итоговый отчет после изменения сценария.

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 **1**, выделенное красным цветом, — число обнаруженных нарушений в сценарии.

Из отчета о покрытии функций видно, что в функции `deposit_spec` покрылись все ветви, а в функции `withdraw_spec` покрылись четыре из семи ветвей, причем в одной из них обнаружилось нарушение.

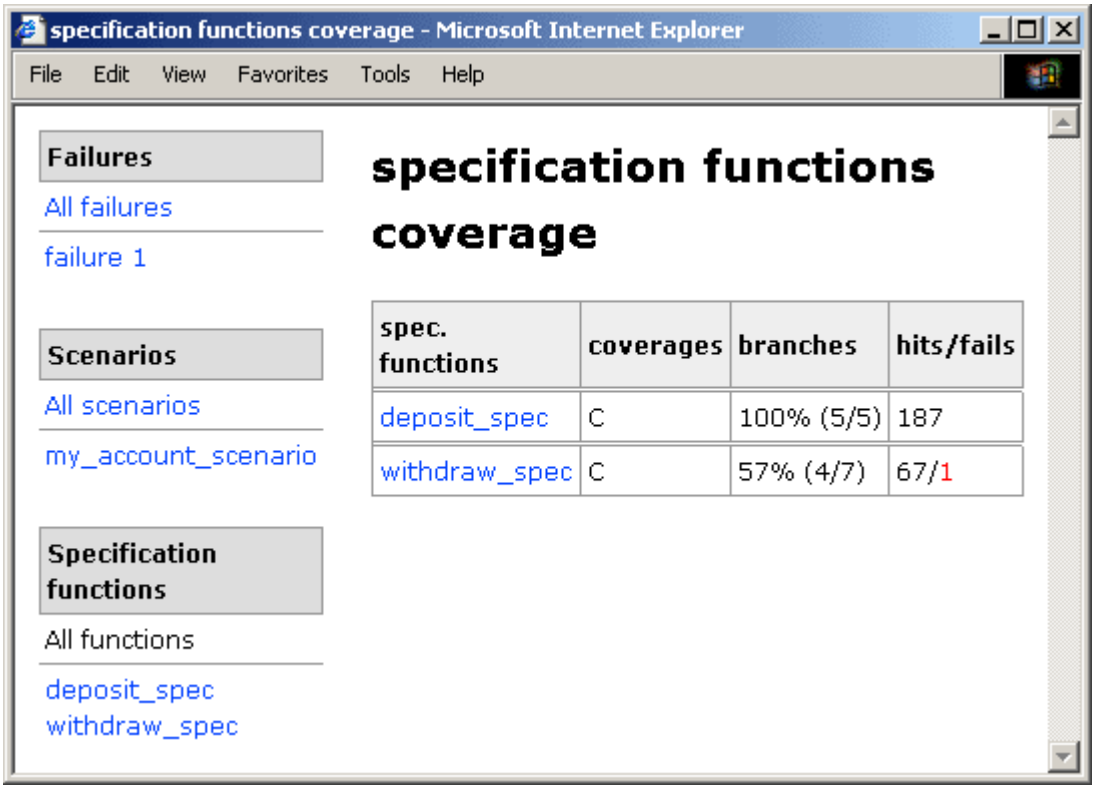

Рисунок 20. Покрытие функций после изменения сценария.

Уменьшение процента покрытия связано с обнаружением нарушения — по умолчанию при обнаружении нарушения работа теста заканчивается.

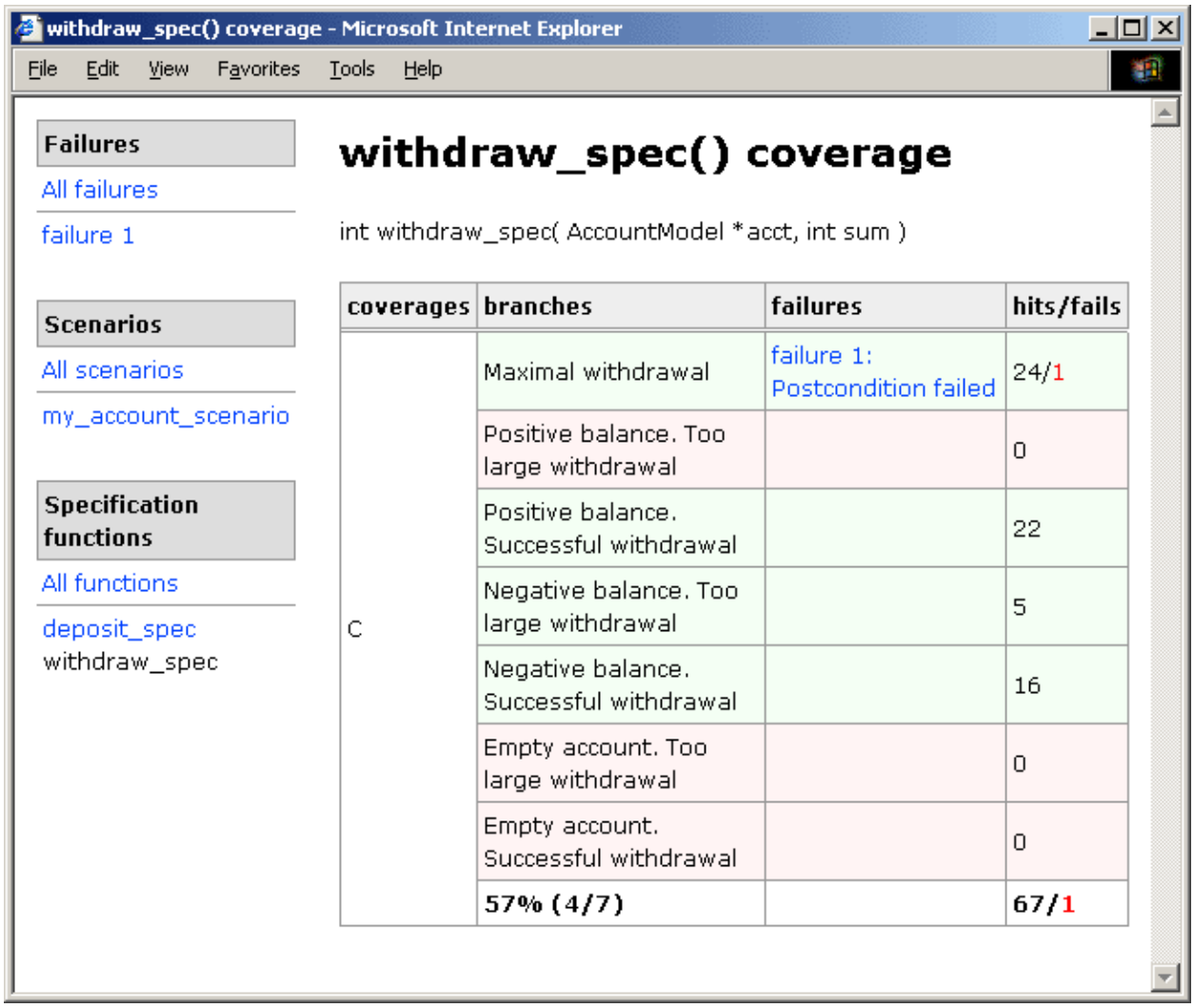

Рисунок 21. Отчет о покрытии функции withdraw_spec после изменения сценария.

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 обнаружено нарушение.

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_spec`.

Рисунок 22. Итоговый отчет об обнаруженных ошибках после изменения сценария.

Подробный отчет об обнаруженном нарушении

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

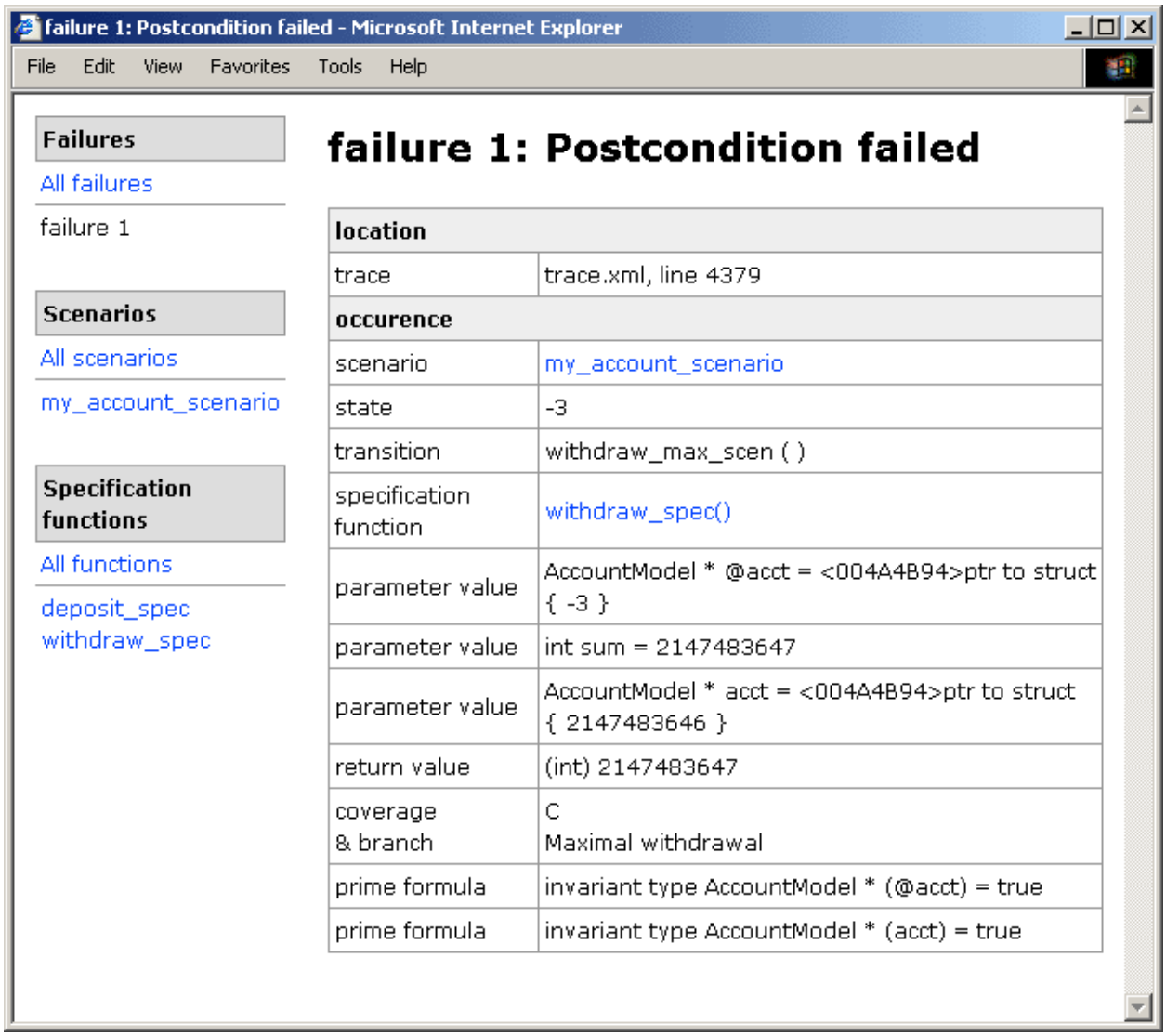

Рисунок 23. Подробный отчет об обнаруженном нарушении.

Отчет содержит подробное описание одного обнаруженного нарушения:

- **trace** — место обнаружения нарушения: в трассе, файл **trace.xml**, номер строки **4379**;

- **scenario** — сценарий, в котором обнаружено нарушение: **my_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** — значения параметров спецификационной функции при обнаружении нарушения: **@acct = <004A4B94>ptr to struct { -3 }**, **sum = 2147483647**, **acct = <004A4B94>ptr to struct { 2147483646 }**;
- **return value** — возвращаемое значение: **2147483647**;
- **coverage & branch** — ветви критериев покрытия, которым соответствуют значения параметров при обнаружении нарушения: **C Maximal withdrawal**;
- **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.

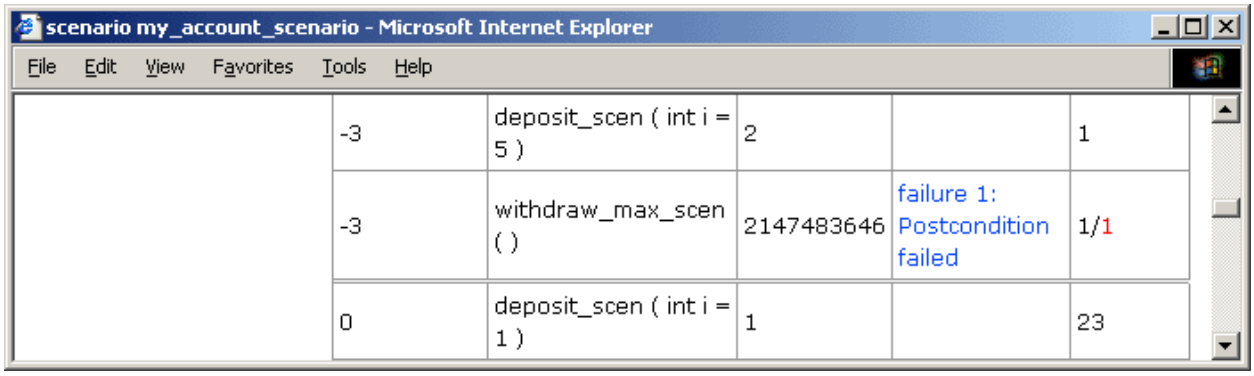

Рисунок 24. Нарушение в отчете сценария.

В этом отчете зафиксировано, что в состоянии при текущем балансе равным **-3** после вызова `withdraw_scen` получено конечное состояние с текущем 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** дерева 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 A: Использование CTesK на платформе Windows

Конфигурация проекта Microsoft Visual Studio 6.0

Проект для разработки тестов CTesK в среде разработки Visual Studio должен быть создан как '**Win32 Console Application**'.

Расширение файлов содержащих конструкции языка SeC должно быть **.sec** или **.seh**.

Панель инструментов CTesK

При запуске среды разработки Visual Studio должна появиться панель инструментов CTesK. Она содержит кнопки для запуска мастеров генерации шаблонов медиатров и сценариев и для запуска анализатора трассы.

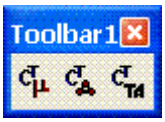

Рисунок 25. CTesK Toolbar.

В случае, если панель инструментов CTesK не появляется, посмотрите раздел "*Известные проблемы*" документа "*CTesK 2.2: Инструкция по установке и удалению*".

Сборка теста

Сборка теста может быть запущена, используя команду '**Build**' среды Visual Studio: выберите элемент '**Build <file>.exe**' в меню '**Build**' или нажмите кнопку **F7**.

Выполнение теста

Выполнение теста осуществляется с помощью команды '**Execute**': выберите элемент '**Execute <file>.exe**' в меню '**Build**' или нажмите кнопки **Ctrl+F5**. Опции трассировки и запуска теста, определенные при его разработке, устанавливаются в окне настроек проекта. Выберите корневой каталог проекта на закладке '**FileView**' окна '**Workspace**' и нажмите **Alt+F7** или выберите элемент '**Settings…**' в меню '**Project**'. В появившемся окне '**Project Settings**' в поле '**Program arguments**' категории '**General**' закладки '**Debug**' добавьте опции трассировки и запуска теста.

Трассировка теста настраивается 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.

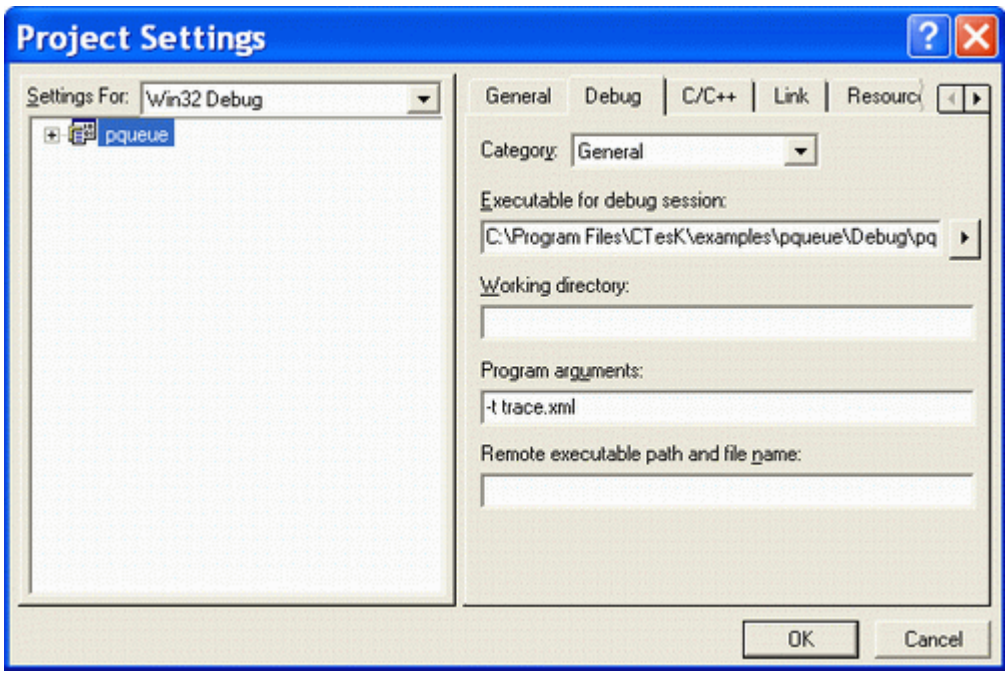

Рисунок 26. Program arguments.

Генерация тестового отчета

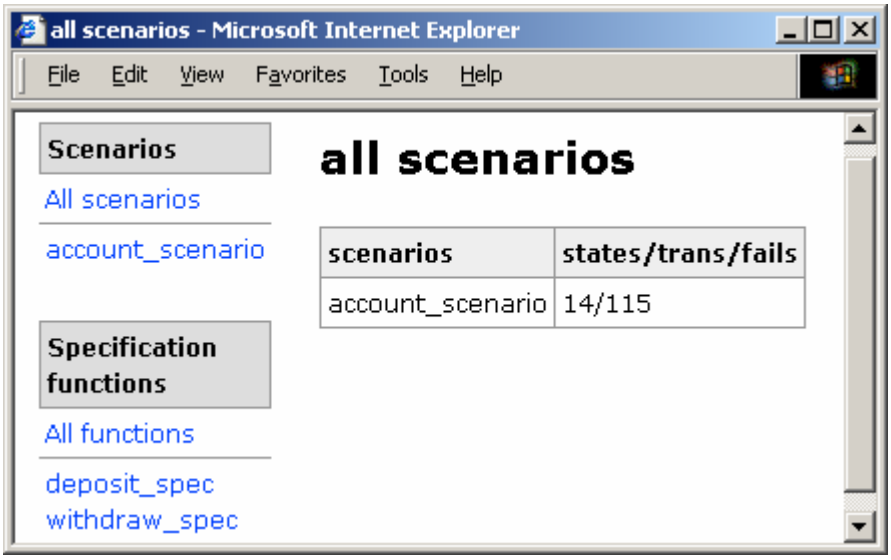

Рисунок 27. Стартовая страница HTML отчета.

Откройте файл содержащий трассу теста в Microsoft Visual Studio и нажмите кнопку 'C^{T}_{TA}' на панели инструментов CTesK. Сгенерируется HTML отчет и откроется Internet Explorer, содержащий его стартовую страницу.

Использование CTesK из командной строки

Для использования CTesK в режиме командной строки запускайте командные файлы **sec.bat** и **ctesk-rg.bat**. Они расположены в подкаталог **bin** установочного каталога CTesK. Для открытия командного интерпретатора на платформе Windows выберите '**Start Menu\(All)Programs\Accessories\Command Promt**'.

Транслятор может быть запущен с помощью команды:

> sec.bat <sec file> <c file> <sei file> [<preprocessor options>]

Где **<sec file>** входной спецификационный файл, **<c file>** выходной файл, **<sei file>** промежуточный файл препроцессорной обработки. Инструмент сгенерирует **<c file>**.

HTML отчет генерируется с помощью команды:

> ctesk-rg.bat -d <output folder> <trace files>

Где **<output folder>** — каталог, в котором будет сгенерирован отчет, **<trace files>** — список файлов, содержащих трассы выполнения тестов.

Использование CTesK со средой Cygwin

CTesK может быть использован со средой Cygwin (см. "*Приложение A: Использование CTesK с компилятором GCC*" документа "*CTesK 2.2 для GCC: Быстрое знакомство*"). Вы можете использовать скрипт **sec.sh** для трансляции SeC файлов в C файлы и **ctesk-rg.sh** для генерации HTML отчетов. Для сборки тестов рекомендуется использовать программу GNU Make.

Если вы не можете найти файлы **sec.sh** и **ctesk-rg.sh** в подкаталоге **bin** установочного каталога CTesK, пожалуйста, посмотрите раздел "*Известные проблемы*" документа "*CTesK 2.2: Инструкция по установке и удалению*".