Using CTesK 2.2 with Microsoft Visual C++® 6.0

Getting Started

Moscow 2006

Contents

Introduction

This document is intended to introduce the basic concepts of CTesK. It provides an example of test development with the help of the tool. It contains a quick overview of the SeC language concepts and syntax to help getting started with test developing in CTesK environment.

CTesK is an implementation of UniTesK test development method, which amplifies industrial test development with a variety of cutting-edge technologies based on formal methods including specification based testing.

UniTesK supports automated development for *functional testing*. Functional testing provides checking whether software behavior is proper or not. In other words functional testing checks conformance of the software to its *functional requirements*.

Any software provides some interface through which it communicates with an environment. Functional requirements do not describe the way how the system should be implemented. They define what externally observable effects the system must produce when interacting with the environment by means of an interface of the system. System behavior conforms to its functional requirements if any effect that is being observed complies with the functional requirements.

Functional test development automation is possible only if functional requirements are specified in a strong formal way. "Formal" means written in a computer readable form that has a unique interpretation. It is not bound to difficult mathematics or theoretical considerations. The difference between informal and formal specifications of functional requirements is likewise difference between natural and programming languages rather than between programming and mathematics languages.

To implement UniTesK method for C software, CTesK uses *SeC* (pronounced as [sek]) language — specially developed *S*pecification *E*xtension of *C* programming language. SeC extends ANSI C with notation for preconditions, postconditions and coverage criteria as well as defining mediators and test scenarios. The main goal is to allow test developers to define and generate components that can be easily composed into a wide range of complete and effective tests, and yet to perform intensive reuse of specifications and scenarios.

CTesK toolkit includes the *translator* of SeC to C, the *library of test system support*, the *specification type library* and the *test report generator*.

The *translator* of SeC to C allows the generation of test components from specifications, mediators and test scenarios. The *library of test system support* provides the *test engine* that implements in C the algorithms of building test sequences and support for tracing test execution. The *specification type library* supports the types integrated with standard functions of creation, initialization, copying, comparison and destroying data of these types. Also the specification type library includes a set of already defined specification types. The *test report generator* provides ability of automatic analyzing test trace and generation of various informative reports from it.

Format conventions

Italic font emphasizes the terms for the main concepts or clauses containing important ideas.

"*Double quoted italic font"* emphasizes references to other documents from the CTesK documentation set.

```
Source code examples are presented in preformatted paragraphs.
```

`Monospaced font` emphasizes code elements dispersed in the text. SeC keywords are emphasized with **`monospace bold font`**.

Bold font marks file names and commands.

Other documents

More information on CTesK and related test development method can be found in other documents included in CTesK 2.2 documentation set: *"CTesK 2.2: User Guide"* and *"CTesK 2.2: SeC Language Reference"*. Also UniTesK web site http://www.unitesk.com/ contains information about UniTesK itself, CTesK and others tools supporting UniTesK.

For additional information and questions, please contact e-mail address support@unitesk.com.

An example of system under test: Bank credit account

The document presents test development process using CTesK tool in Microsoft Visual C++® 6.0 on the example of test development for a system that implements the functionality of a bank credit account: money deposition and money withdrawal. The account provides an option for a credit with preset maximum credit.

If you have not installed CTesK 2.2 in your system or have installed it without an option of the integration into IDE, before further reading this document, please, turn to "*CTesK Installation and Usage Instructions"* and install CTesK 2.2 with Microsoft Visual C++® 6.0 integration option.

The implementation of a bank credit account is located in the **account.c** file in the **examples\account** folder of the CTesK tree. An account itself is implemented as a structure `Account` defined in **examples\account\account.h**.

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

The implementation to be tested is located in the file **examples\account\account.c**.

The interface of the system consists of two functions:

- `void deposit(Account *acct, int sum)` — deposits a positive amount `sum` to the account by increasing the balance of the account with the given amount;
- `int withdraw(Account *acct, int sum)` — withdraws a positive amount `sum` from the account and returns the actually withdrawn amount, by which the account balance is decreased. If the difference between the current balance and the amount `sum` is out of the permissible credit, the method does not change balance of the account and returns `0`. A maximal value of the credit is defined by macro `MAXIMUM_CREDIT` in **account.h**. It should not be negative.

Our objective is to develop test for this system employing CTesK tool, run the test and analyze the obtained results.

The next sections describe the development of the test. It includes the following steps:

- Development of a specification of the system under test
- Development of mediators
- Development of a test scenario
- Test execution and analysis of test results.

IDE Run-up

Run Microsoft Visual C++® 6.0 and open the worspace file **account.dsw** located in the **examples\account** folder of the CTesK tree: select the menu item '**File\Open Workspace...**', in the pop-up window '**Open Workspace**' open needed folder, select the file **account.dsw** and click the button '**Open**'.

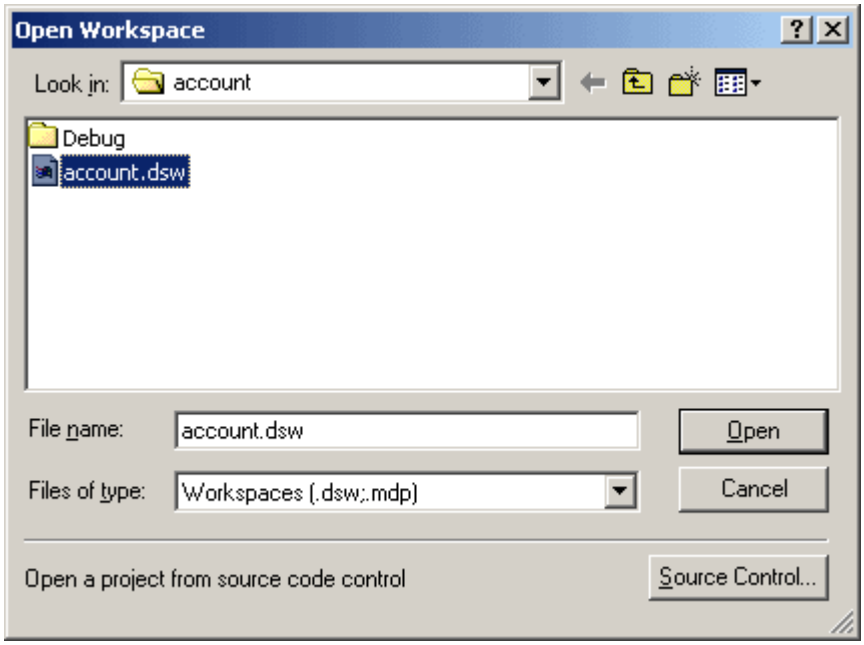

Figure 1. Opening the account project.

After it the tab '**FileView**' of the window '**Workspace**' representing the file tree of the project should look as on the Figure 2.

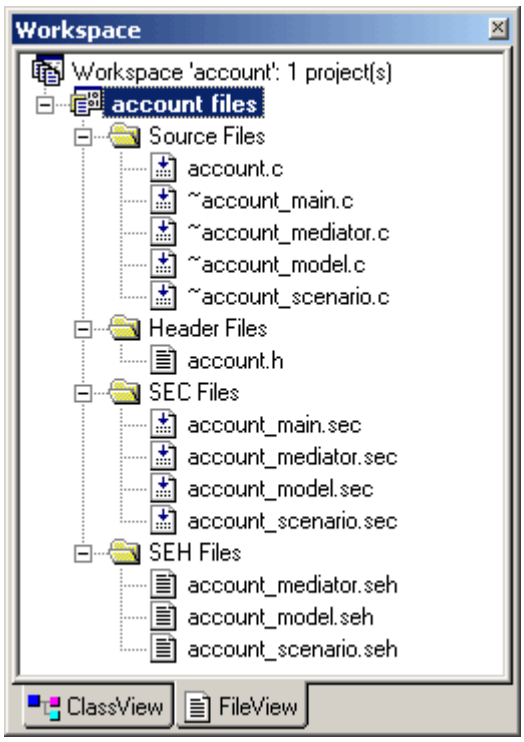

Figure 2. Files of the account project.

Files are divided into four groups:

- **Source files** — files containing the code on C language. They could contain an implementation under test, auxiluiry librararies and so on. Also files generated by CTesK with names begining the symbol '~' belong to this group;
- **Header files** — header files on C language;
- **SEC Files** — files containing the code on SeC language, and having the extension **.sec**. They contain test components: specifications, mediators and test scenarios;
- **SEH Files** — header files on SeC language having the extension **.seh**. They contain declarations of interfaces of test components.

Specification of Bank credit account example

The UniTesK test development method supported by the CTesK tool assumes that the functional requirements should be represented in a clear, unambiguous and computer readable form, which is called *formal specifications*. Due to formal representation it is possible to use specifications to generate programs that verify the compliance of the real behavior of interface functions with the requirements stipulated for them.

In CTesK formal specifications are written in a special language named SeC[1], which is an extension of C programming language. SeC allows describing of the *functional requirements* that determine the functionality of interface functions, i.e. what the system must do when call of its interface function takes place.

Specifications in SeC language have C-like syntax. Files with the SeC code have **.sec** or **.seh** extensions.

The specification of the account example can be found in the **account_model.sec** file located in the **examples\account** folder of the CTesK tree.

The account specification starts with including the **account_model.seh** header file located in **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
;
```

The **account_model.seh** contains an including files **limits.h** and **account.h** and the declarations of extern *variable with invariant*, *type with invariant* and *specification functions*.

The **limits.h** file is included to allow using the `INT_MAX` constant.

The **account.h** file is included to allow using the `MAXIMUM_CREDIT` constant and the `Account` structure type defined in it:

[1] Pronounced as [sek]

```
#define MAXIMUM_CREDIT 3

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

The `Account` type represents an account as a structure with a single field of `int` type. When `balance` is negative its absolute value should not exceed the limit defined by macro `MAXIMUM_CREDIT`. To describe this requirement in the **account_model.seh** file the `AccountModel` type is declared as the `typedef` of `Account` *type with invariant*. The invariant is defined in **account_model.sec**:

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

It returns `true`, if a value of the `balance` field of verified structure meets the requirement, and `false` otherwise. The invariant should be held before and after calling interface functions, which use data of the type having constraints described in the invariant. Thus an invariant encapsulates common parts of constraint specifications of these interface functions.

Because the set of valid values of the `AccountModel` type does not coincide with the value set of the `Account` structure, the `AccountModel` type is a subtype of the `Account` type.

The `MaximalCredit` *variable with invariant* is declared in the **account_model.sec** file. In the same file its invarint is defined.

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

The invariant of the `MaximalCredit` variable describes the requirement to the maximal value of the credit — it should not be negative. The variable invariant should be held before and after any calls of any interface functions.

Further constraints on the system behavior are described in special functions marked with the keyword **`specification`**. They are called *specification functions*.

The `deposit_spec` specification function is correspondent to the `deposit` interface function.

This interface function does not return any result and has two parameters. The first one is a non-null pointer to the `Account` structure that represents an account to which money should be deposited. The second parameter of `int` type is an amount of money to deposit. The function should read the second parameter and update the `balance` field of the structure pointed by the first parameter: after the call the `balance` field should be increased exactly by the number passed in the second parameter. Besides, the value of the second parameter should be more than zero and should not be too large to cause overflow in the `balance` field after increasing.

In SeC these requirements are described in the following specification function `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; }
}
```

The definition of specification function begins with the signature:

```
specification void deposit_spec (AccountModel *acct, int sum)
```

The signature of any specification function should contain a keyword **specification**. Besides the name the signature of the `deposit_spec` specification function differs from the signature of the `deposit` implementation function only in the type of the first parameter. It is a pointer to the `AccountModel` type, which is a subtype of the `Account` type.

After the signature *access restrictions* follow.

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

They show that when the `deposit` function is called

- the system's behavior depends on the value of the `MaximalCredit` variable[2], and its value should not be changed after the call;
- the system's behavior depends on the value of balance field of the structure referenced by `acct` parameter, and its value can be changed after the call.

In addition access restriction of the `balance` field defines an *alias* of the mentioned field. The *alias* is used in the body of the specification function to simplify and clarify expressions.

The keyword **reads** specifies "read only" access restriction, i.e. the values of parameters and variables under the **reads** access restriction should not be changed as a result of the call of the described function.

In SeC like as in C, any change of values of the arguments passed by value cannot be visible outside the function, i.e. these parameters always have the **reads** access restriction.

Unlike C, parameters passed trough pointers are interpreted more strictly. If a pointer is not of the `void*` type it is considered as a pointer to a single value of pointed type, not an array of values [3]. Pointers to the `void` type are interpreted as just values of an address, paying no attention to content of memory referred by these pointers.

At run-time CTesK checks that values referenced by pointers with the **reads** access restriction are not changed right after each call of the corresponding interface function. Also CTesK provides automatic run-time checking for the values of **reads** parameters and variables against their invariants, if any, before each call of the corresponding interface function.

The keyword **updates** specifies "read-write" access restriction, i.e. the values of parameters and variables under the **updates** access restriction may be changed as a result of the call of the described function. In SeC like as in C, the externally visible value of the argument may be changed only if it is passed through a pointer. But in SeC, it must be considered strongly as a pointer to a single value of the corresponding type.

CTesK provides an automatic run-time checking for values of **updates** parameters and variables against their invariants, if any, before and after each call of the corresponding interface function. By default, all parameters of a specification function have "read-write" access restriction.

---

[2] In accordance to the requirements the behavior of the system under test is defined only when a maximal credit is no t negative, this requirement is described in the `MaximalCredit` variable invariant. For automatic checking the variable invariant before pre- and postcondition checking the variable access restriction should be described in the corresponding specification functions.

[3] To specify the function with the dynamic arrays as arguments one should use containers of *specification types*. For details see "*CTesK 2.2: Users' Guide*" and "*CTesK 2.2: SeC Language Reference*"

The body of the `deposit_spec` specification function describes the behavior of the system under test when calling the `deposit` interface function. The body contains a description of functional requirements in the form of *pre-* and *postconditions* and *coverage criteria*.

When calling the `deposit` interface function the pointer to an account structure should be non-null, an amount of money to be deposited should be positive and the sum of the current balance and the second parameter should not exceed the maximum value of the `int` type. In SeC it is described in the precondition.

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

It is a block statement marked with the **`pre`** keyword. The precondition returns `true` if input values of parameters are valid and `false` otherwise. Thus precondition specifies definition domain of the function. If input parameters' values do not belong under it, the behavior of the function is undefined.

Precondition should have no side effects. No more than one precondition can be defined in a specification function. It should be located before coverage criteria and postcondition. If there are no constraints on input values the precondition may be omitted.

After the `deposit` interface function call the account balance should be equal to the account balance prior to the function call increased by the `sum` amount. In SeC these requirements are described in the postcondition.

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

It is a block statement marked with the **`post`** keyword. The postcondition returns `true` if input and output values of the function call conform to the functional requirements, and `false` otherwise. Thus it verifies that the function behaves correctly.

A special unary operator `@` is used in the postcondition to get access to the input value of the alias of the `balance` field. That is, in the postcondition '`balance`' denotes the output value of the alias of the `balance` field, and '`@balance`' denotes the input value.

This operator is applicable to expressions inside the **`post`** block statement only. The keyword **`post`** defines the point where the corresponding implementation function is called. In the body of a specification function expressions located before the **`post`** keyword are evaluated before the implementation function call. Expressions located after the **`post`** keyword are evaluated after the call except for expressions under `@` operator that are evaluated before the implementation function call.

Postcondition should have no side effects. The specification function should have exactly one postcondition. It should follow precondition and coverage criteria, if any.

According to the requirements the `deposit` function has the uniform behavior on the whole function definition domain. It is rather reasonable assumption that the behavior of any implementation of the `deposit` function does not depend of the absolute value of the current balance and an amount of money to be deposited. But it may depend of the sign of the current balance. Also the function behavior should be tested when the parameters' values are on the boundaries of sets of their allowable values. Therefore coverage criterion of the `deposit` specification function distinguishes five different test situations.

```
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" };
}
```

The function behavior should be tested in each situation defined in the coverage criterion.

The coverage criterion is a *named* block statement marked with the **coverage** keyword. It defines the partition of the function behavior into branches — *functional branches*. Each branch is defined by the `return` operator with a construct similar to the structure variable initialization construct in C. It should contain an identifier as the first field— *branch identifier*, and a string literal as the second field — *branch name*.

The partition defined by the **coverage** block should be complete and unambiguous, i.e. each allowable set of input parameters' values should correspond to a single functional branch.

In a specification function several coverage criteria with different names can be defined. The **coverage** blocks should be located after precondition and before postcondition. They should have no side effects. If no coverage blocks are defined, it is equivalent to a coverage criterion with a single functional branch.

The `withdraw_spec` specification function is correspondent to the `withdraw` interface function.

```
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;
  }
}
```

The `withdraw` interface function returns a value of the `int` type and has two parameters. The first one is a non-null pointer to the `Account` structure from which money should be withdrawn. The second parameter is a number of `int` type that is an amount of money to withdraw. The function should read the second parameter and update the `balance` field of the structure pointed by the first parameter. If the requested withdrawal does not lead to the maximum credit overcome, then the `balance` field after the call should be decreased exactly by the number passed in the second parameter. Otherwise the `balance` field should not be changed. The function should return the withdrawn sum in the case of successful withdrawal or `0` otherwise.

The precondition states that the `acct` pointer should be non-null and the `sum` amount to withdraw should be positive.

There are two main use cases of the withdraw function — when the withdrawal of the amount given is possible and when it is impossible. In the **coverage** `C` block the functionality is

partitioned into the seven branches. This criterion specifies that each use case should be tested with the current balance values from different subsets of its definition domain — especially on the domain boundaries.

The postcondition divided into two cases: when the withdrawal of the amount given is possible and when the withdrawal of the amount given is impossible. In the first case the postcondition tells that the balance should be reduced by the `sum` value and the function should return `sum`. In the second case the postcondition tells that the balance should not be changed and the function should return `0`. A function identifier is used to refer to the result returned by the function, in the given example it is `withdraw_spec`.

In order to obtain the components that check the calls of the specified interface functions, the specification should be translated into the C code.

To translate the specification file in Microsoft Visual C++® 6.0 IDE, open the **account_model.sec** file and choose the menu item '**Build\Compile account_model.sec**' or press the key combination **Ctrl+F7**.

As a result of the translation the **~account_model.c** file should be generated in **examples\account** and added to the project folder '**SEC Files**'.

Mediators of Bank credit account example

An implementation of the bank credit account and its specification should be bound to enable the test to check their conformance to each other.

In UniTesK method, special components called *mediators* are used for this purpose.

In SeC mediators are implemented by special *mediator functions* marked with the keyword **`mediator`**.

The project **account** contains implemented mediator defined in the file **account_mediator.sec** located in the folder **examples\account** of the CTesK tree. You can skip instructions for developing mediators and go to the next section.

To create a template of new mediator in Microsoft Visual C++® 6.0 the wizard '**CTesK Mediator Wizard**' should be run. To launch it click the button '$\mathbf{C^T_\mu}$' located on the CTesK tool panel.

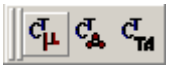

Figure 3. The button launching CTesK mediator wizard.

In the pop-up window on the first step ('**SEH Files**') of the wizard should SeC header files containing declarations of specification functions describing the functionality to be tested.

In the list '**All project SEH files**' select the file **account_model.seh** and click the button '**Add >**'. As a result selected file moves into the list '**Selected SEH files**' as shown on the Figure 4. Then click the button '**Next >**'.

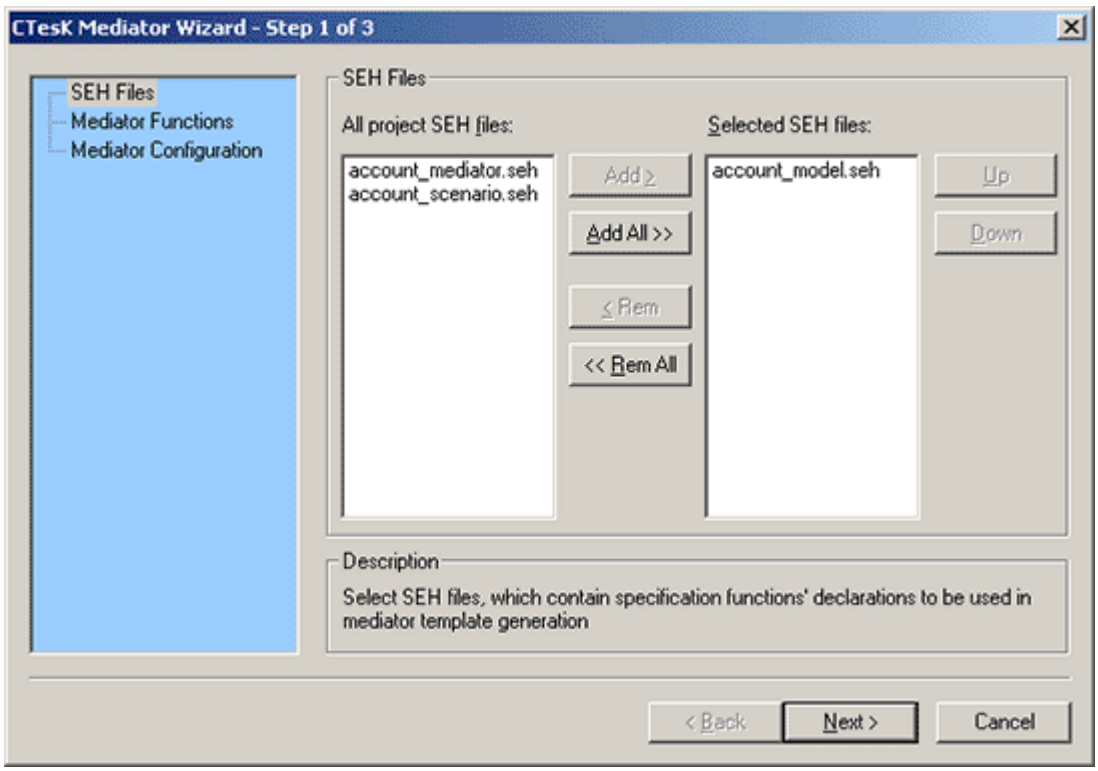

Figure 4. Selecting SEH files containing specification function declarations.

On the next step ('**Mediator Functions**') you should select specification functions, for which mediators should be developed, and define names of mediator functions.

Do not change default setting and click the button '**Next >**'.

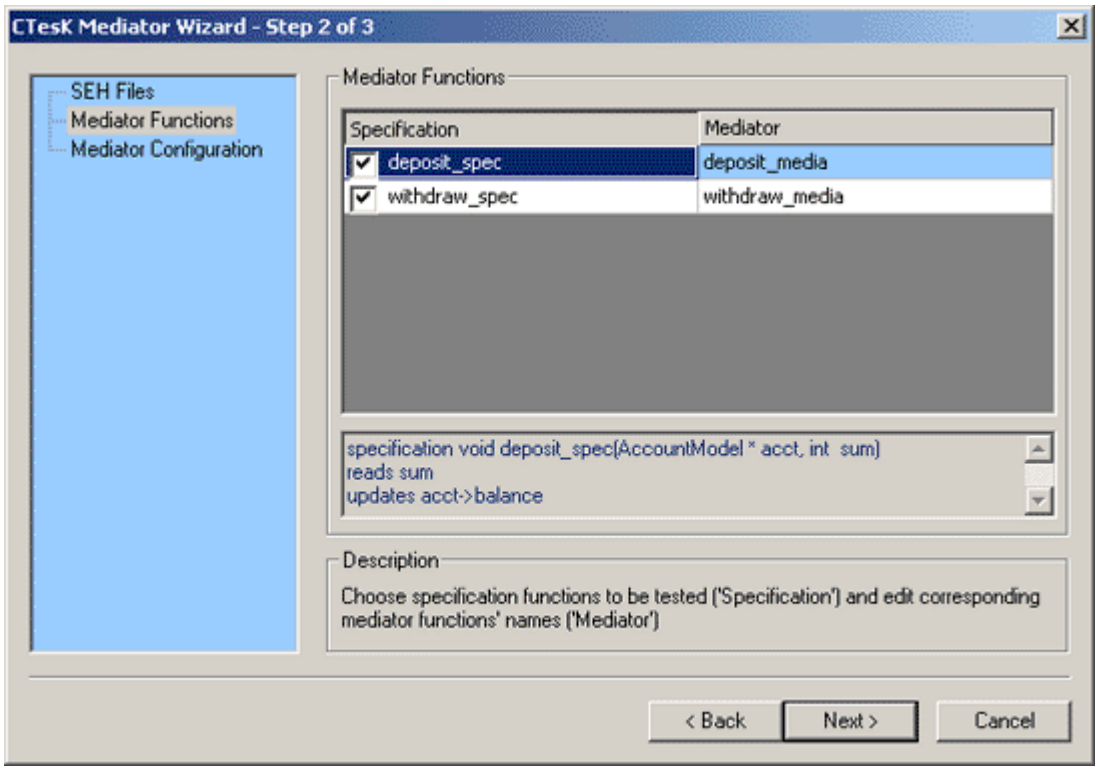

Figure 5. Selecting specifiation functions to be tested and names of their mediator functions.

On the next step ('**Mediator Configuration**') you should select a type of the implementation under test and define name for SeC files for generated mediator template.

Do not change default implementation type '**Open state implementation**' (in this case synchronization of the specification and implementation states is implemented in one function) and type '**my_account_mediator**' in the text field '**Mediator file names**'. To generate mediator template click the button '**Finish**'.

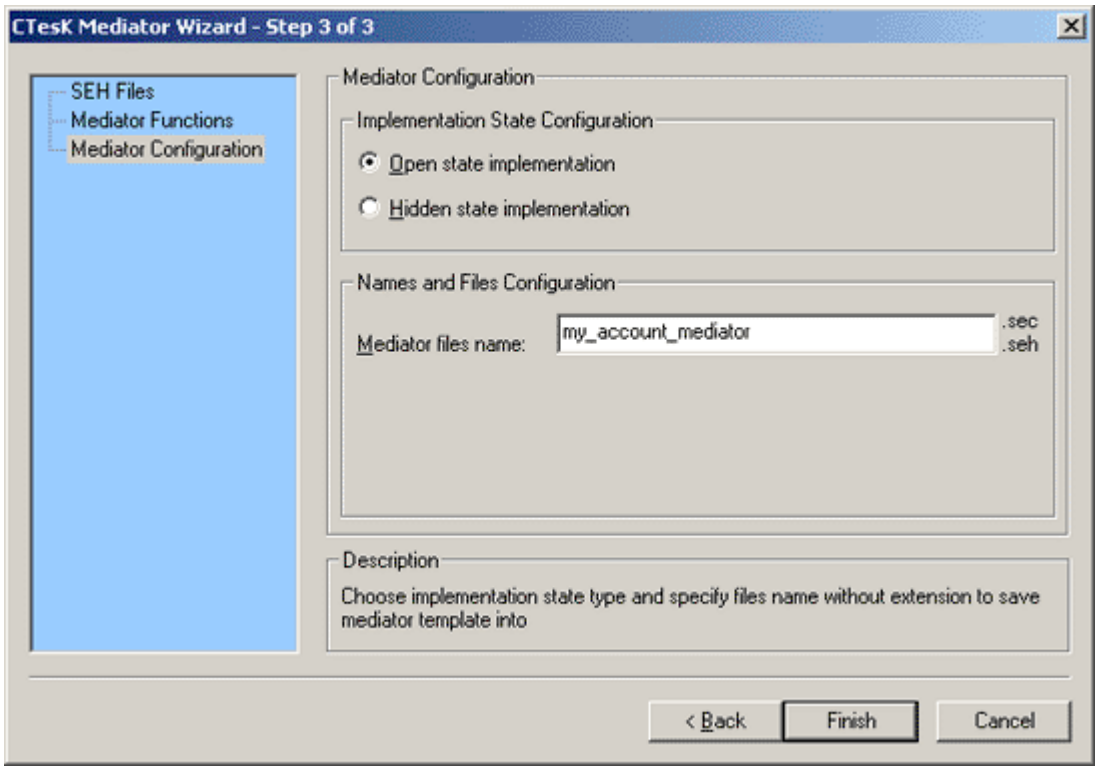

Figure 6. Selecting implementation type and mediator name defining.

As a result two files are generated **my_account_mediator.seh** and **my_account_mediator.sec** containing mediator template. The files are automatically added to the current project. In the editor window of Microsoft Visual C++® 6.0 the file **my_account_mediator.sec** is opened, which should be added by the code implementing meditor functionality.

In the file **my_account_mediator.sec** templates of the following functions are defined:

- the function `map_state_up_my_account_mediator` synchronizing the specification and implementation states;
- two mediator functions: `deposit_media` and `withdraw_media`. The first one binds the specification function `deposit_spec` c with the implementation interface function `deposit`, and the second — `withdraw_spec` with `withdraw`.

To complete mediator development it should be done the following:

- the state synchronization function should be implemented;
- in the blocks of the mediators marked with the keyword **`call`** the functionality described in the correspondent specification function by means of the call of the correspondent implementation interface function should be implemented.

In the example there are no special actions for state synchronization because the pointer to the specification state is paased to the implementation functions among of their arguments. Do not change the function `map_state_up_my_account_mediator`:

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

Let's consider the template of the mediator function `deposit_media`.

The function definition begins with the keyword **`mediator`**, then the mediator function name, the keyword **`for`** and the signature of the specification function with access restrictions follow.

In the block of the mediator function body marked with the keyword **`call`** an implementation of the functionality described in the specification function by means of the call of the corresponding implementation interface function should be defined.

Add in the call block the call of the function `deposit` with the same arguments as the specification function ones (`acct` and `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 ();
  }
}
```

Likewise in the call block of the template of the mediator function `withdraw_media` add the call of the function `withdraw` with `acct` and `sum` as arguments. Because the specification function has return value the call block of its mediator should return the result of the call:

```
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 ();
  }
}
```

Make sure that developed file is correctly translated into the C code. To translate it select the menu item '**Build\Compile my_account_mediator.sec**' or press the key combination **Ctrl+F7**.

As a result in the folder **examples\account** the file **~my_account_mediator.c** is generated and added into the project.

Test scenario of Bank credit account example

Specifications provide a formal description of the functionality of a system under test. Components that check individual calls of the specified interface functions are generated basing on them. Mediators provide binding between the specification and the implementation under test. They allow testing different implementations of the same functionality using the same specifications.

To check the behavior of the system under test in various conditions, relevant sequence of calls to interface functions should be built. In CTesK test sequences are built automatically by *test engine*. Test engine should be given by a short description of the test called *test scenario*.

The scenario of the account example can be found in the **account_scenario.sec** located in the **examples\account** folder of the CTesK tree.

Test scenario consists of two main parts:

- a *function of building generalized state*, which is used to reduce the number of test situations;
- *scenario functions*, describing iterations of arguments of specification functions.

The project **account** contains implemented the test scenario defined in the file **account_scenario.sec** located in the folder **examples\account** of the CTesK tree. You can skip instructions for developing test scenario and go to the next section.

To create a template of new test scenario in Microsoft Visual C++® 6.0 the wizard '**CTesK Scenario Wizard**' should be run. To launch it click the button 'C^T_Δ' located on the CTesK tool panel.

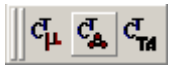

Figure 7. The launching CTesK scenario wizard.

In the pop-up window on the first step ('**SEH Files**') of the wizard should SeC header files containing declarations of specification functions describing the functionality to be tested and their mediator functions.

In the list '**All project SEH files**' select the files **account_model.seh** and **my_account_mediator.seh**, and click the button '**Add >**'. As a result selected file moves into the list '**Selected SEH files**' as shown on the Figure 8. Then click the button '**Next >**'.

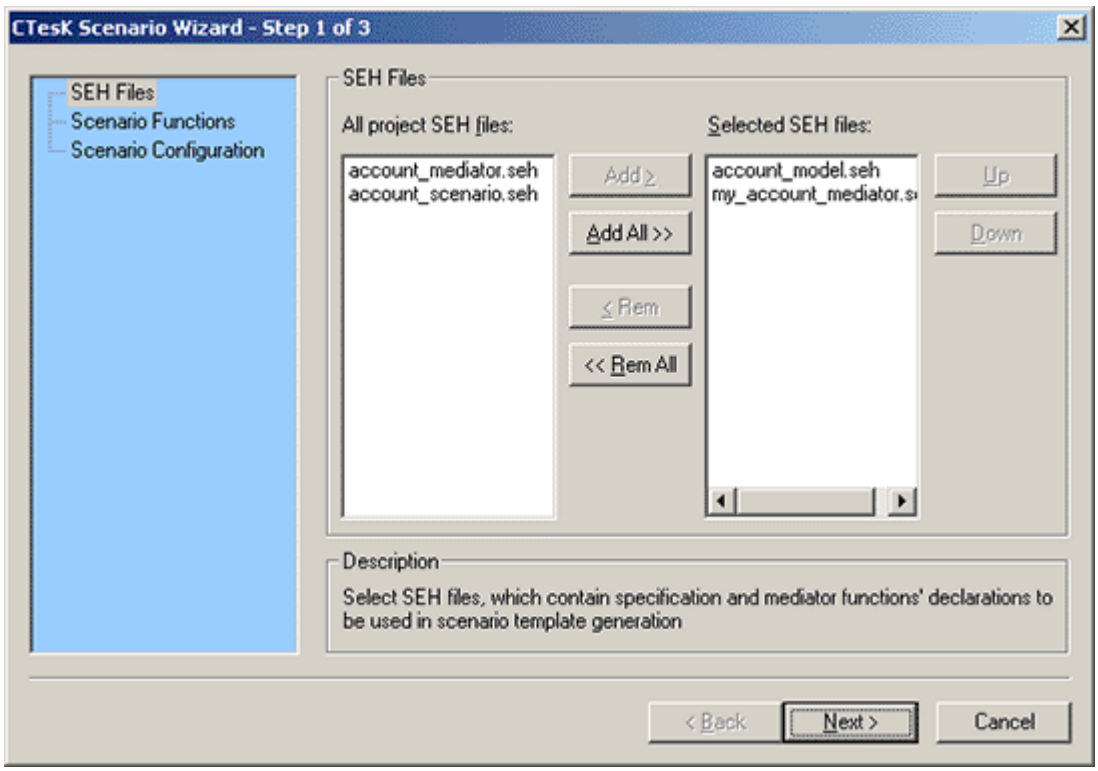

Figure 8. Selecting SEH files containing specification and mediator function declarations.

On the next step ('**Scenario Functions**') you should select specification functions, for that templates of scenario functions should be created, define their names, select corresponding mediator functions, and turn on (if it is needed) the filtration by coverage for specification function arguments.

Do not change default setting and click the button '**Next >**'.

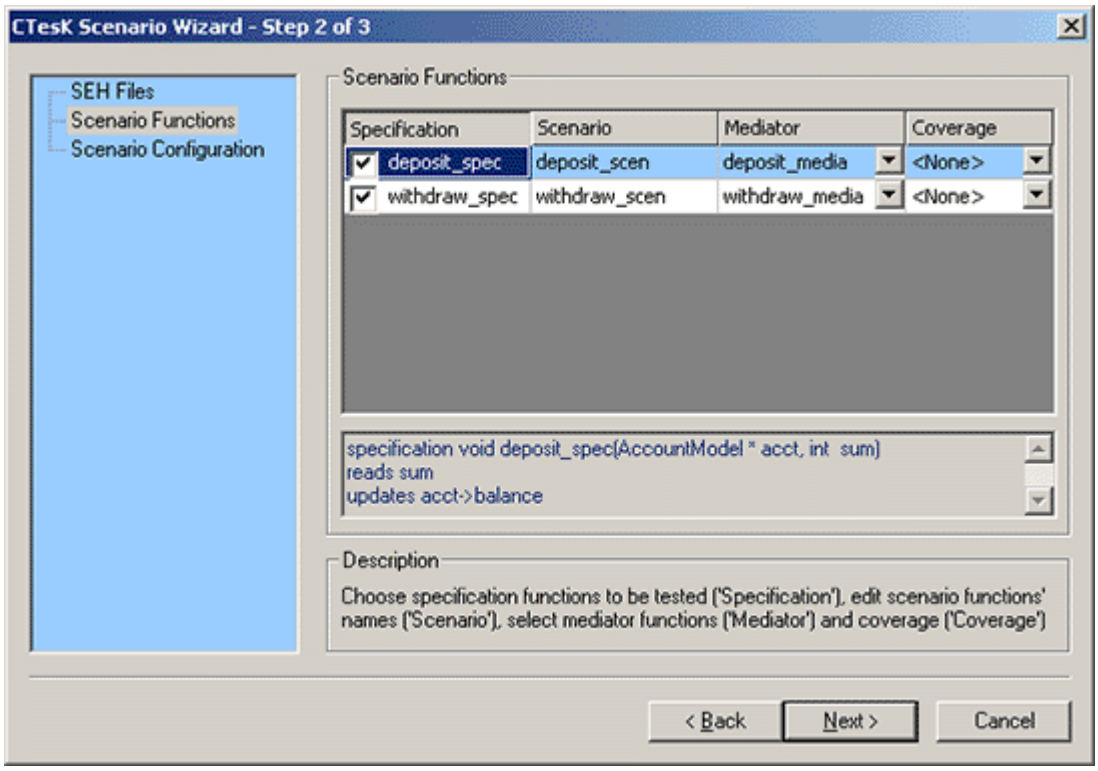

Figure 9. Selecting specification functions to be tested, defining scenario function names, and setting other options.

On the next step ('**Scenario Configuration**') you should define a type and creation arguments of generalized state of the test scenario, and names of SeC files to be generated.

As the generalized state we use the account balance. Because it is an integer select the item '**Integer**' in thy drop-down list '**Scenario state type**'. Type **my_account_scenario** in the text field '**Scenario files name**'. Turn on the option '**Into the separate files**' and type **my_account_main** in the text field '**Main files name**'. Then click the button '**Finish**'.

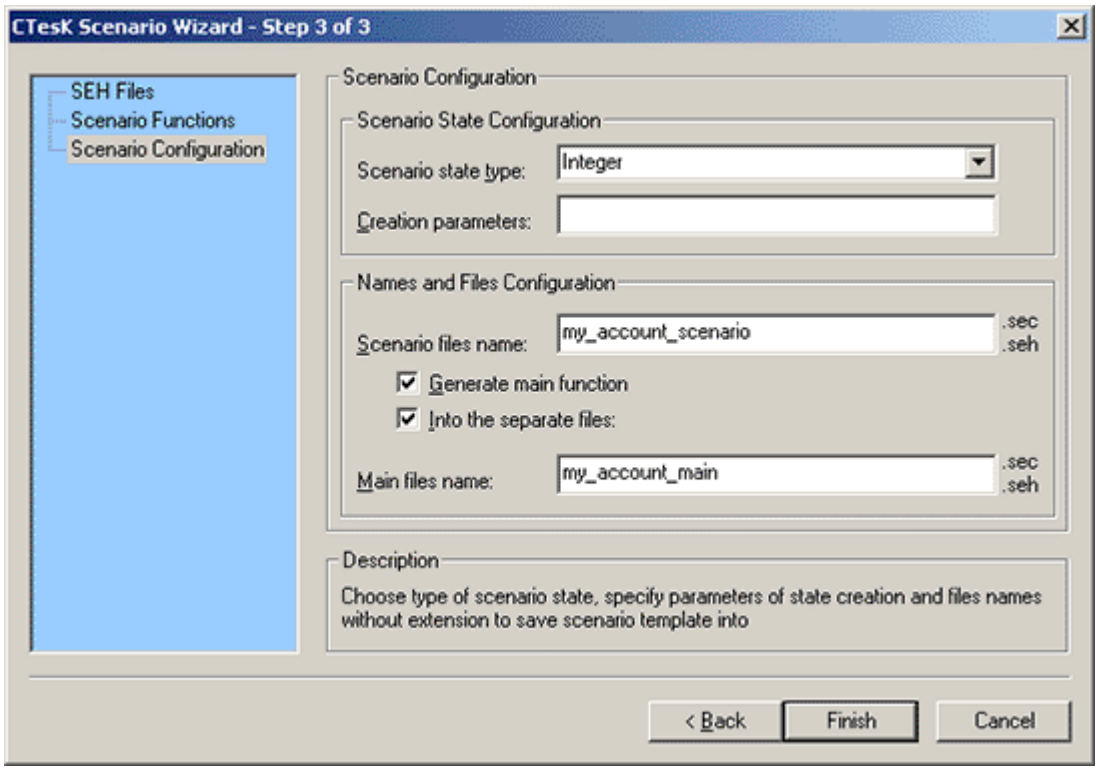

Figure 10. Defining type and creation arguments of generalized test scenario state, and names of files to be generated.

Four files **my_account_scenario.seh**, **my_account_scenario.sec**, **my_account_main.seh** and **my_account_main.sec** should be generated and automatically added into the current project. They contain the template of the test scenario and the function `main` launching the test. In the editor window the file **my_account_scenario.sec** is opened, which should be added by the code implementing test scenario functions.

In the file **my_account_scenario.sec** are defined templates of the following functions:

- an initialization function of the test scenario `my_account_scenario_init`;
- a finalization function of the test scenario `my_account_scenario_finish`;
- a function of building generalization state `my_account_scenario_state`;
- two scenario functions `deposit_scen` and `withdraw_scen`, that describing argument iteration for specification functions `deposit_spec`, вторая and `withdraw_spec` correspondingly.

To complete test scenario development should be done the following:

- a set of specification data containing the history of changes of the account balance should be defined;
- an initialization function of the test scenario `my_account_scenario_init` should be defined;

- a finalization function of the test scenario `my_account_scenario_finish` should be defined;
- a function of building generalization state `my_account_scenario_state` should be defined;
- iterations of arguments of specification functions ahould be added into scenario functions `deposit_scen` and `withdraw_scen`.

Specification functions `deposit_spec` and `withdraw_spec` have as their argument a pointer to a variable of the type `AccountModel` containing the account balance. During testing the only account `acct` will be used. Add the declaration of the variable `acct` of the type `AccountModel` in the file begining:

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

Then define account initialization actions into the initialization function `my_account_scenario_init` (assining zero into the field `balance` of the variable `acct`):

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

Do not change the finzlization function `my_account_scenario_finish`:

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

For building the generalized state of the test scenario based on the account balance add the function `my_account_scenario_state` the field `acct.balance` as an argument of the function `create`:

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

Let's consider the templates of the scenario functionы `deposit_scen` and `withdraw_media`.

Any scenario function should have return type **`bool`**, should be marked with the keyword **`scenario`** and should have no arguments.

Using the operate **`iterate`** add to the scenario fucntion `deposit_scen` an iteration of deposition value from `1` to `5`, for the account having the balance not more `5`:

```
/*
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;
}
```

To the scenario fucntion `withdraw_scen` add an iteration of withdrawal value from `1` to `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;
}
```

Make sure that developed file is correctly translated into the C code. To translate it select the menu item '**Build\Compile my_account_scenario.sec**' or press the key combination **Ctrl+F7**.

As a result in the folder **examples\account** the file **~my_account_scenario.c** is generated and added into the project.

Running test of Bank credit account example

The last component of the account example is located in the **account_main.sec** file generated by the scenario wizard (see the previos section "*Test scenario of Bank credit account example*"). It contains definition of the function `main` of the test program, that obtains command line arguments of the test and launches the test scenario with these arguments.

```
#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;
}
```

The header file **my_account_scenario.seh** contains the declaration of the scenario variable:

```
scenario dfsm my_account_scenario;
```

The function `main` calls with its arguments the function `my_account_main`, which sets the mediator functions for the specification functions `deposit_spec` and `withdraw_spec`:

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

, and starts the scenario. The last looks as calling the function with the same name as th scenario variable (`my_account_scenario`) and having the same arguments as the function `main`:

```
  my_account_scenario (argc, argv);
```

To translate the file **my_account_main.sec** open it in IDE, and select the menu item '**Build\Compile my_account_main.sec**' or press the key combination **Ctrl+F7**.

As a result in the folder **examples\account** the file **~my_account_main.c** is generated and added into the project.

Now there are all needed files for building the executable test file: the implementation source file **account.c**, and generated files — **~account_model.c**, **~my_account_mediator.c**, **~my_account_scenario.c** and **~my_account_main.c**.

Remove from the project needless files: **account_mediator.seh**, **account_mediator.sec**, **~account_mediator.c**, **account_scenario.seh**, **account_scenario.sec**, **~account_scenario.c**,

account_main.seh, **account_main.sec** and **~account_main.c**. To do it select they in the window '**Workspace**' (on the tab '**FileView**') and press the key '**Delete**'.

To build the executable test file select the menu item '**Build\Build account.exe**' or pres the key **F7**.

As a result in the folder **examples\account** the executable file **account.exe** should be created.

The command line options of the test are passed by the `main` function to the test scenario. The standard options of test scenario are the following[4]:

-t <file-name>	—	trace will be directed to the file '**<file-name>**'
-tc	—	trace will be directed to the console
-tt	—	trace will be directed to the file '**<scenario-name>--YY-MM-DD--HH-MM-SS.utt**'
-nt	—	no trace will be created

Test scenario processes standard arguments and passes the rest to an initialization function of the test scenario.

Let's run the executable file with parameters directing trace to the **trace.xml** file.

To define of the test program options select the menu item '**Project\Settings…**' or press **Alt+F7**, select the tab '**Debug**', in the text field '**Program arguments**' of the category '**General**' type a line '**-t trace.xml**', and press the button '**OK**'.

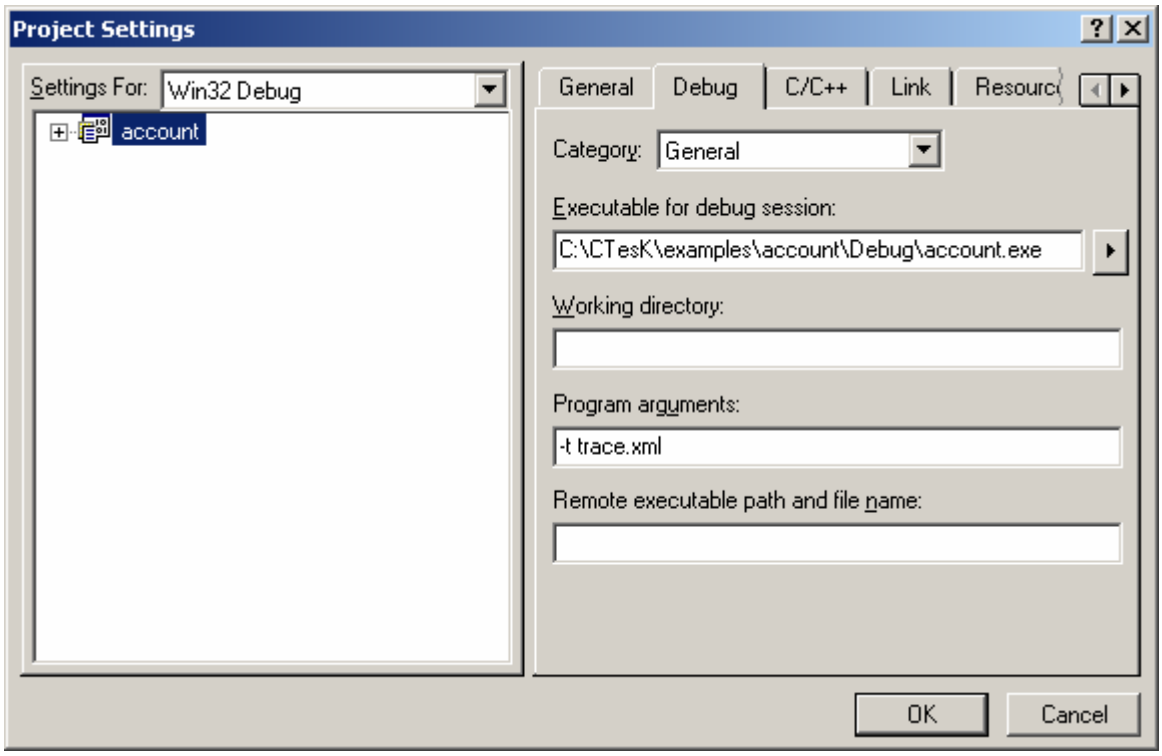

Figure 11. Setting the test execution options.

To run the test select the menu item '**Build\Execute account.exe**' or press the key combination **Ctrl+F5**.

As a result of the test execution the **trace.xml** file should be generated in the folder **examples\account**.

[4] By default the –tt option is used, i.e. trace will be directed to the file '**<scenario-name>--YY-MM-DD--HH-MM-SS.utt**'.

Add the file **trace.xml** to the project. To do it select the menu item '**Project\Add To Project ►\Files...**'. In the pop-up window '**Insert Files into Project**' open needed folder, in the drop-down list '**Files of type**' select the file type '**All Files (*.*)**', select the file **trace.xml** and press the key '**OK**'.

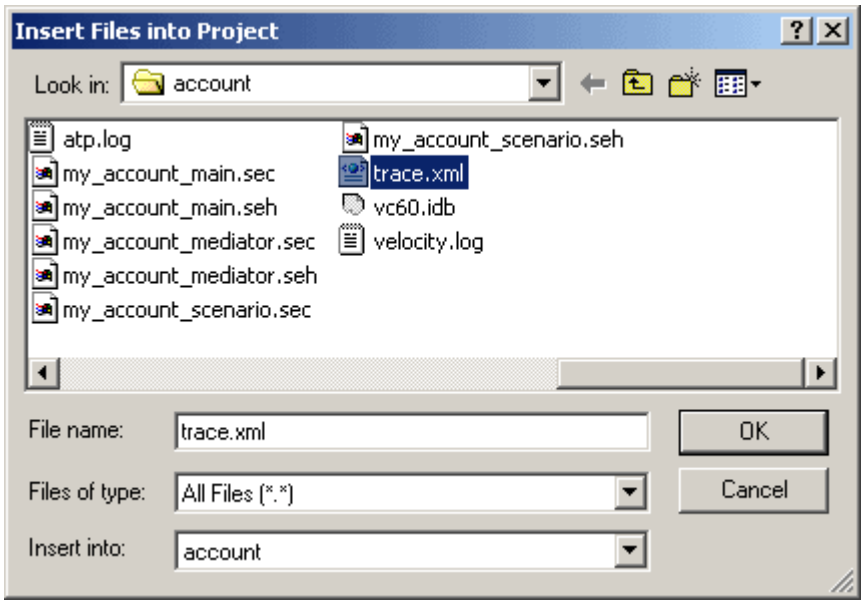

Figure 12. Adding the trace file into the project.

Test result analysis of Bank credit account example

Text test report generation

To generate test reports open the trace file **trace.xml** in Microsoft Visual C++® 6.0 and launch CTesK trace analyzer by clicking the button 'C^{T}_{TA}' located in the CTesK tool panel.

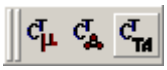

Figure 13. The button launching CTesK trace analyzer.

As a result in the folder **examples\account** the subfolder containing generated test reports should be created. Test reports are a set of linked HTML documents.

After report generation the program for report browsing should be automatically start. In the left frame of the start page is placed a navigation report list.

Summarized scenario report

The start page contains *a summarized test report*. It shows how many states and transitions were visited and how many fails were detected for each scenario.

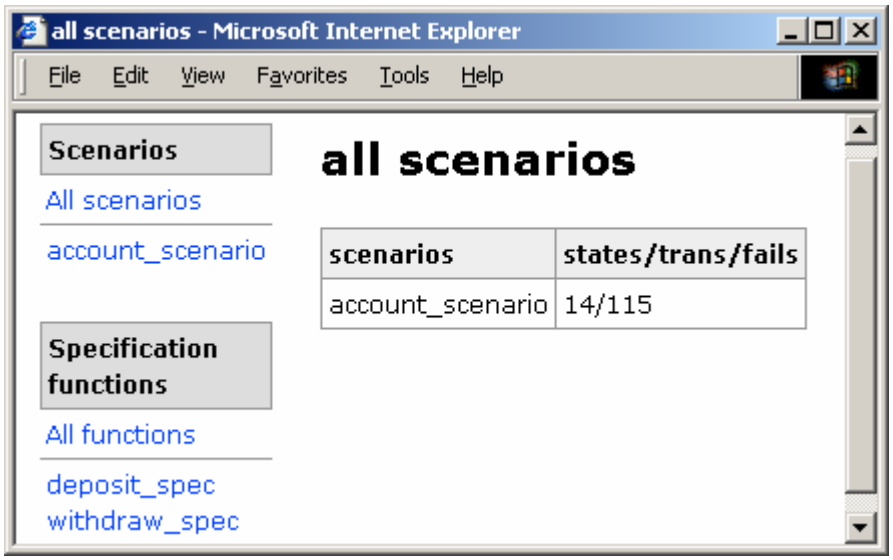

Figure 14. The summarized test report.

In the account example the only scenario is available. It has visited **14** states and **115** transitions. No fails were detected.

Detailed scenario report

A detailed scenario report can be opened by the scenario name link. It describes all states and transitions visited during the test scenario execution. The first three columns of the table describe transitions. The last one shows the total number of hits and the number of failures detected on the transition given.

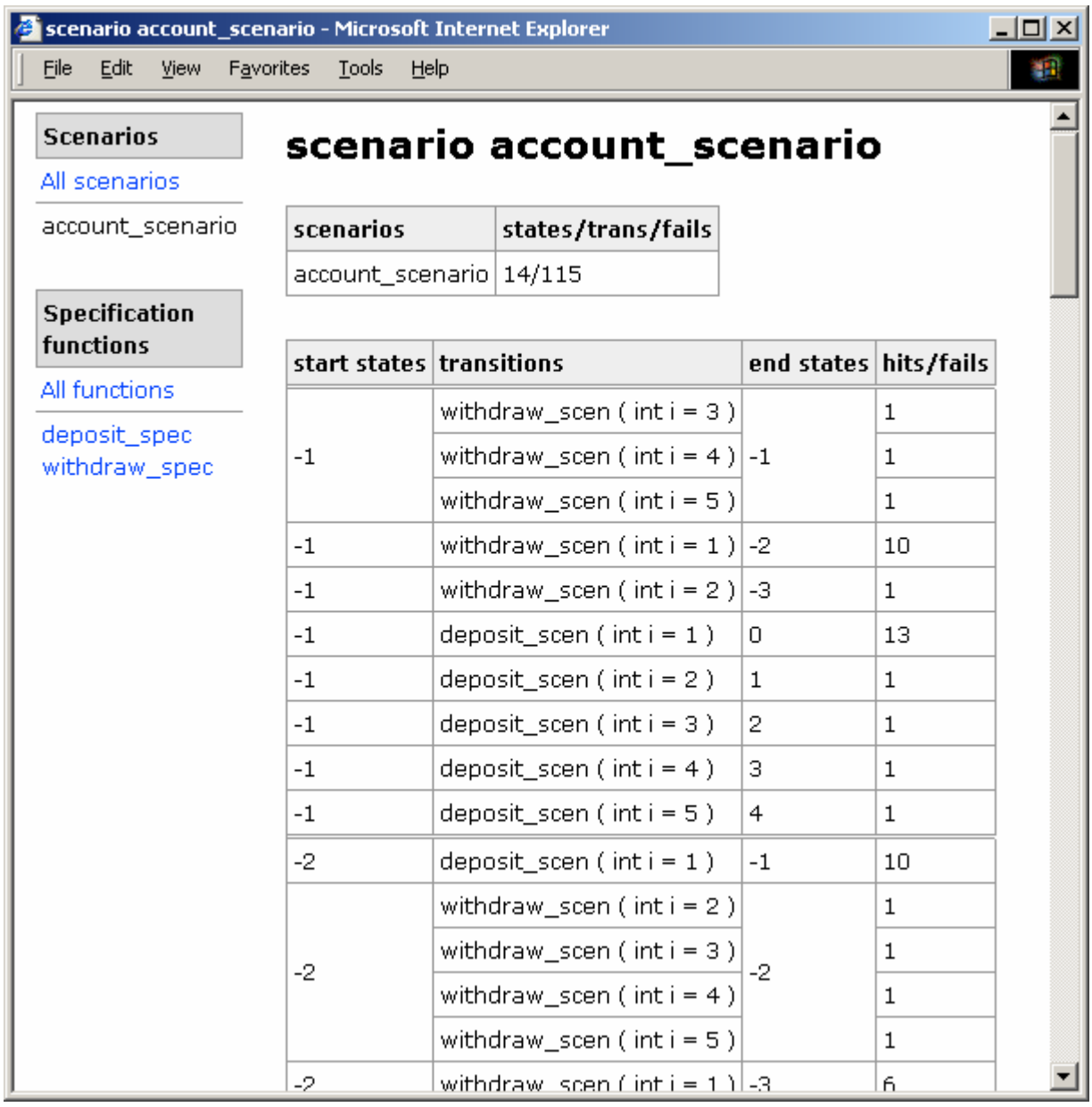

Figure 15. The detailed scenario report.

For instance, there are ten transitions started from the test scenario state **–1** in the account example. The test scenario state was defined as the current value of the balance in the. Therefore this state corresponds to the balance value **–1**.

The transition marked **deposit_scen(int i = 1)** leads to the state **0**. The mark shows the transition is performed by call of the scenario function `deposit_scen` with the value of the iterated variable `i` equal to `1`. This transition was performed **13** times and no failures were detected.

Summarized function coverage report

A summarized function coverage report can be opened by the **All functions** link. It shows a percentage of branch coverage for each tested function.

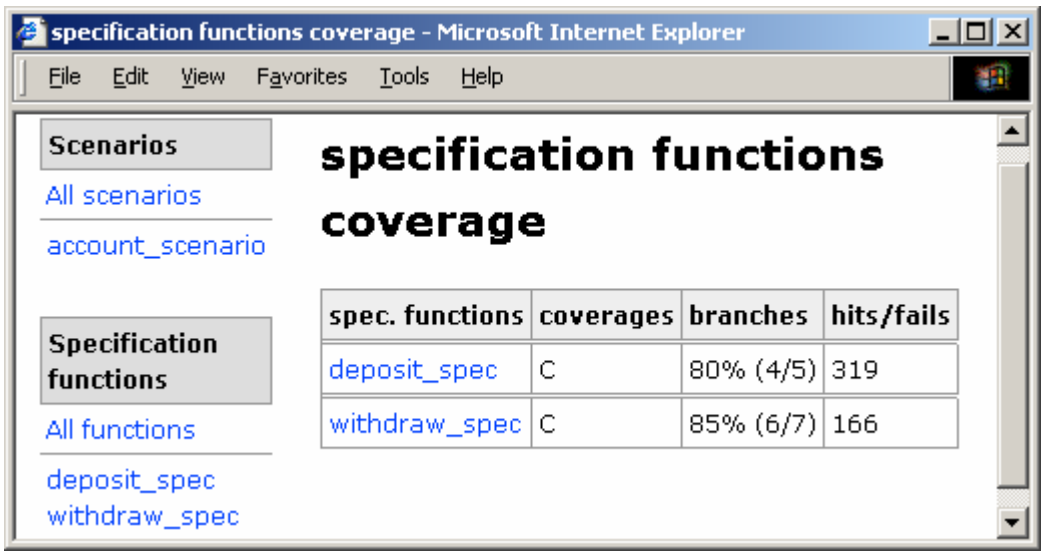

Figure 16. The summarized function coverage report.

There are two specification functions in the account example. The both of them have one coverage called **C**. The `account_scenario` scenario has covered four of five branches of the `deposit_spec` function and six of seven branches of the `withdraw_spec` function.

Detailed function coverage report

A detailed function coverage report can be opened by the function name link. It includes information about a number of hits and fails in each branch of the function given.

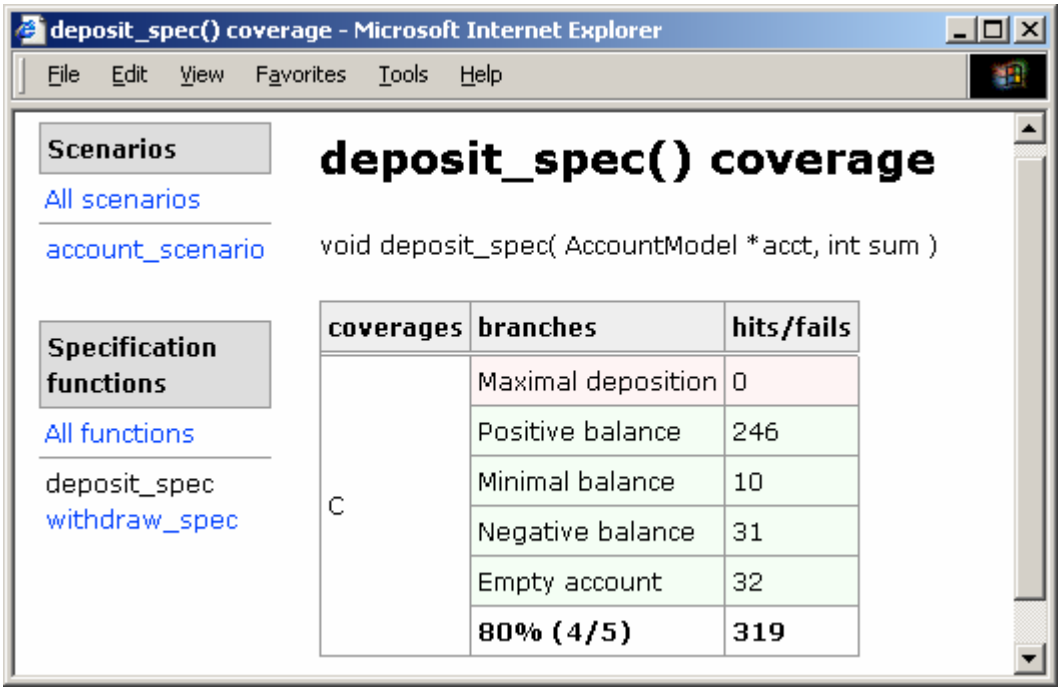

Figure 17. The detailed function coverage report of `deposit_spec` function

The report of the `deposit_spec` function shows that the `deposit_spec` function was called with arguments corresponding to **Positive balance**, **Minimal balance**, **Negative balance** and **Empty account** branches — **246**, **10**, **31** and **32** times respectively. No calls were performed with the arguments corresponding to **Maximal deposition**.

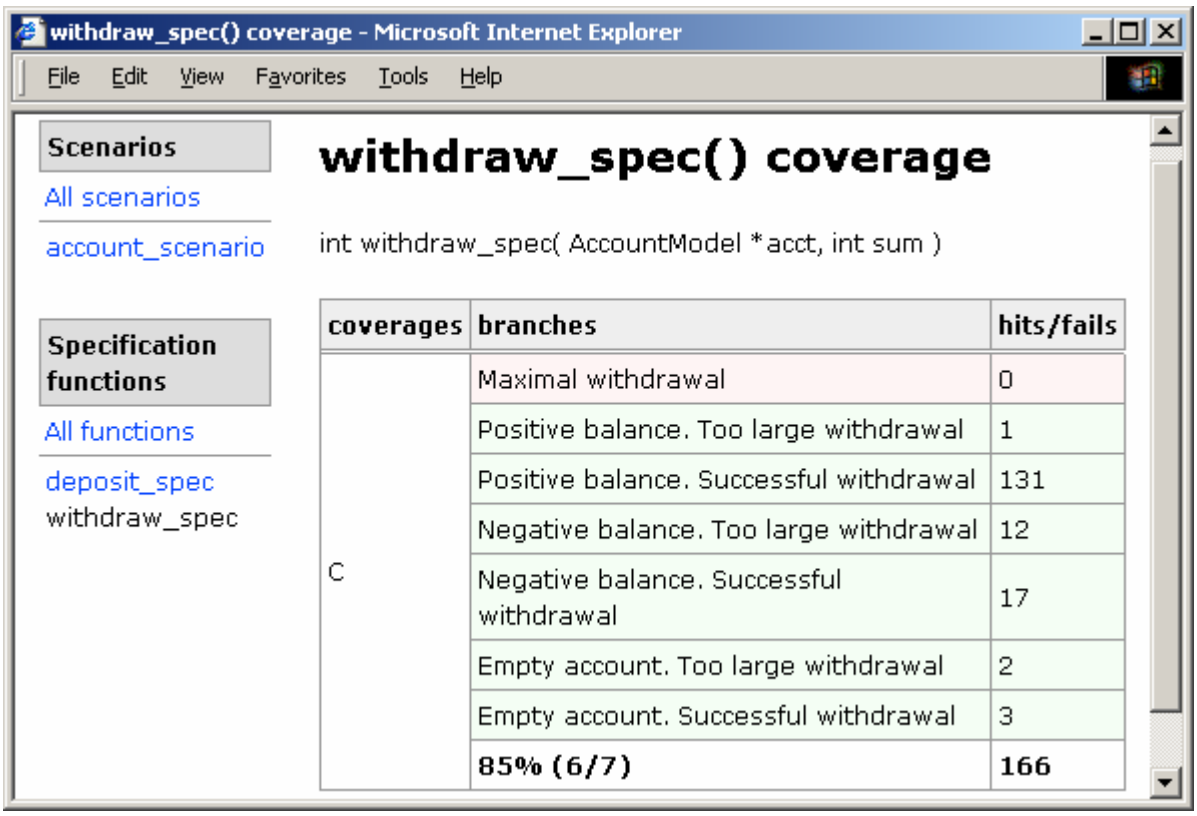

Figure 18 The detailed function coverage report of `withdraw_spec` function.

The report of the `withdraw_spec` function shows the `withdraw_spec` function was called with arguments corresponding to all branches besides **Maximal withdrawal** branch.

To ensure complete coverage of the branches of the `deposit_spec` and `withdraw_spec` function two new scenario functions should be defined in the scenario. They should provide the parameter values to maximal deposition and maximal withdrawal.

```
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
              }
};
```

The condition `if (0 < acct.balance && acct.balance < INT_MAX)` in the `deposit_max_scen` function is required to prevent the overflow during evaluation the expression `INT_MAX - acct.balance` and precondition violation when depositing zero sum.

But now the number of test states equals to the sum of `INT_MAX` and `MaximalCredit`. To prevent unacceptable growth of the number of test states the `withdraw_scen` function should be changed:

```
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;
}
```

That is if account balance is more `5` only two new functions will be called.

Rebuild the test, run it and generate reports.

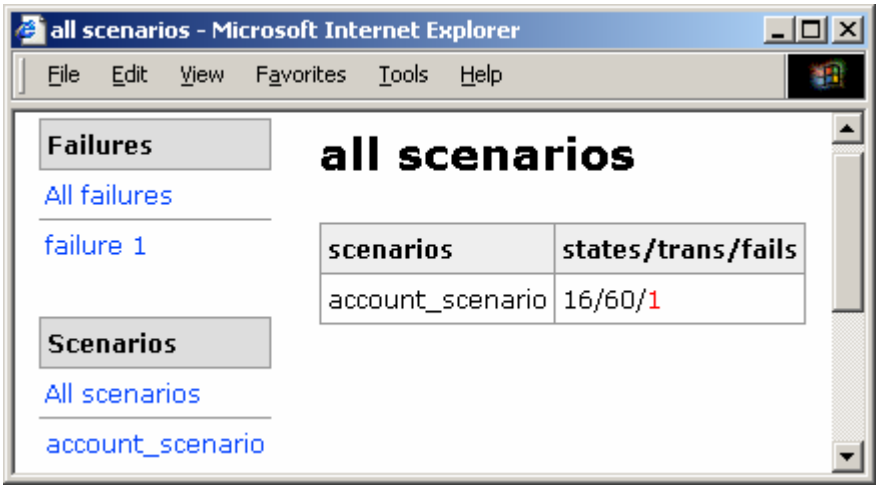

Figure 19 The summarized test report containing the failures

Now on the start report page there are the numbers of failures found in each scenario marked with red color and new links to *summarized failure report* and *detailed failure reports* on the navigation bar. Besides there is a number **1** marked with red color.

The summarized function coverage report shows that all branches of the `deposit_spec` function is covered, but among seven branches of the `withdraw_spec` function only four ones is covered, and a failure is found in one of covered branches.

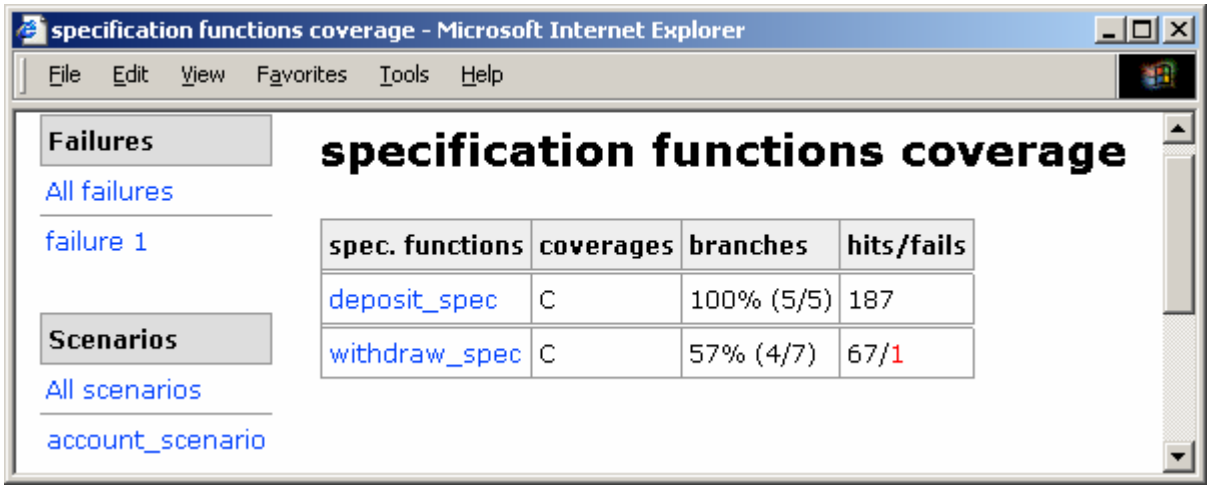

Figure 20. The summarized function coverage report after scenario changes.

Decreasing coverage is caused by failure occuring — by default test running is stoped when occuring failure.

The detailed coverage report of the `withdraw_spec` function shows, that after scenario changes the **Maximal withdrawal** branch is covered, and in this branch a failure is found.

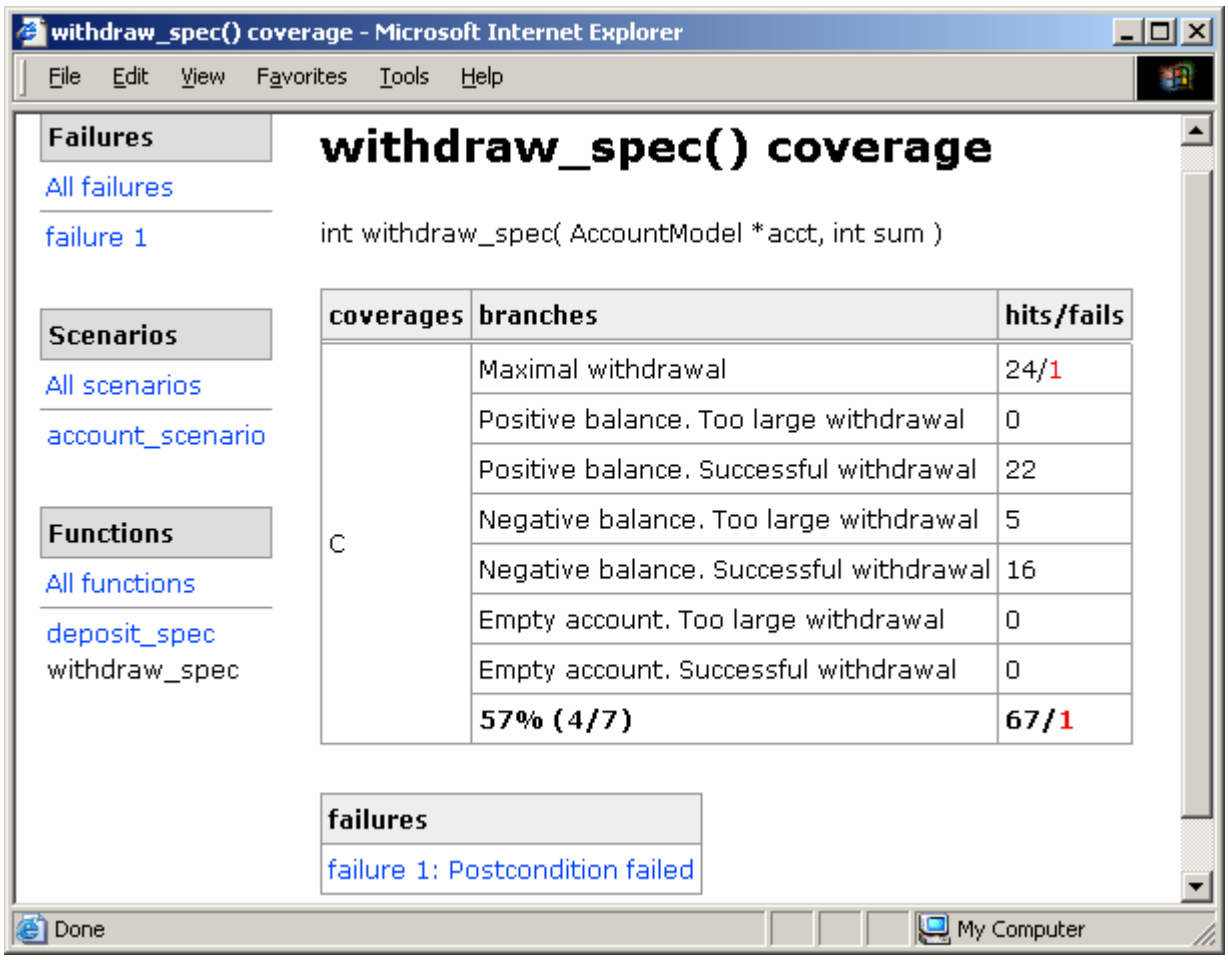

Figure 21. The detailed function coverage report of `withdraw_spec` function after scenario changes.

Summarized failure report

A summarized failure report can be opened by the **All failures** link. It contains a list of detected failures with a short description containing a kind of failures and a place where it has become apparent.

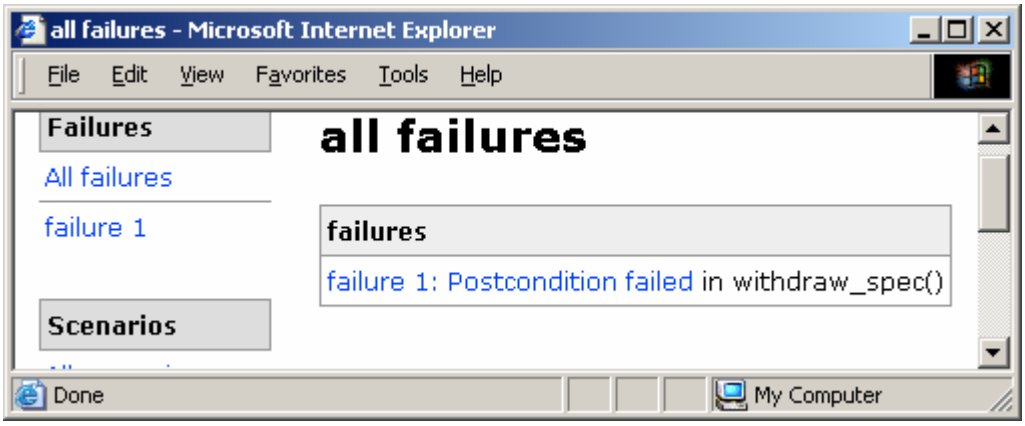

Figure 22. The summarized failure report.

The report shows one failure — the violation of the postcondition of the `withdraw_spec` function.

Detailed failure report

A detailed failure report can be opened by the **failure <*failure number*>** link.

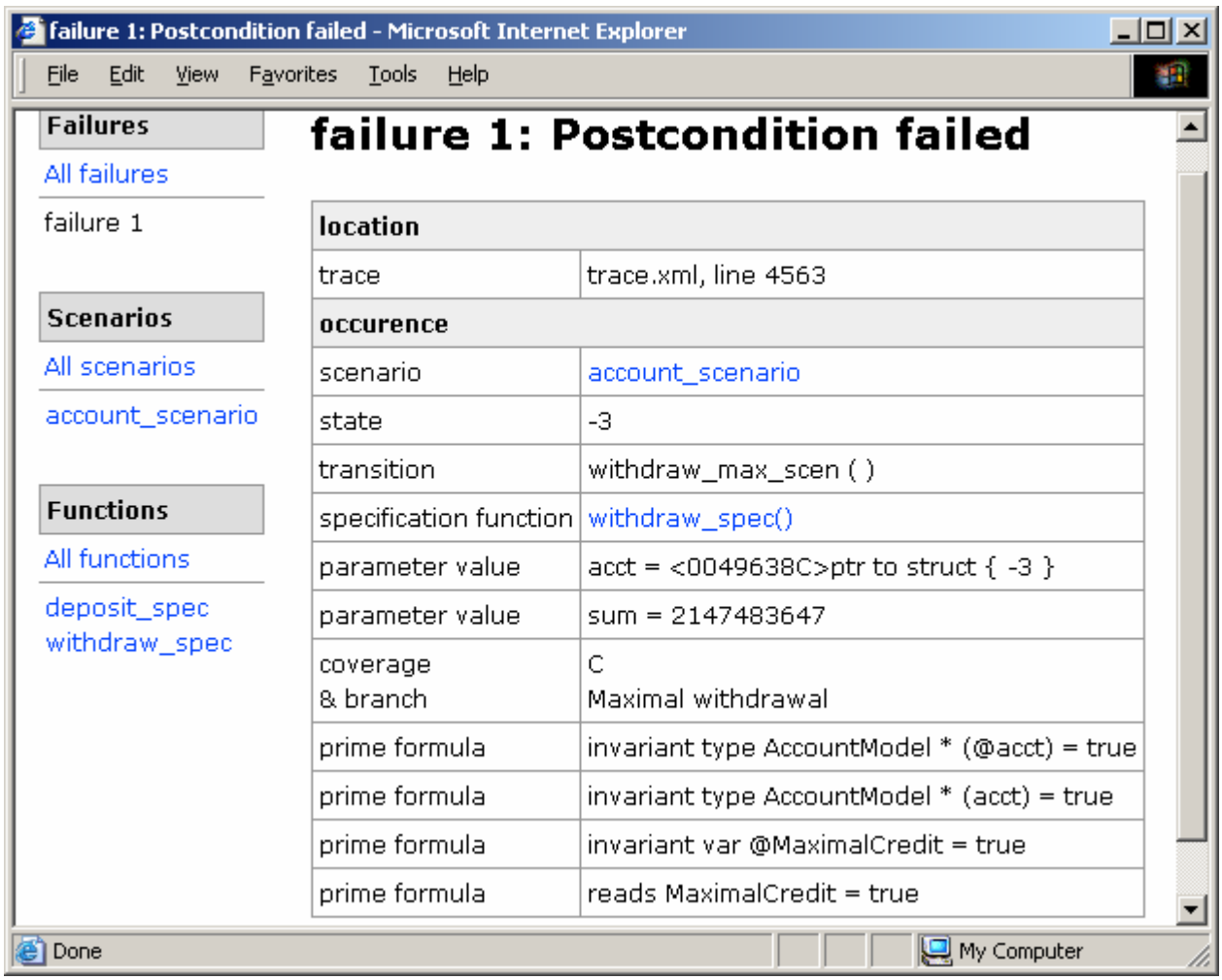

Figure 23. The detailed failure report for the erroneous implementation

It contains a detailed description of the failure:

- **location** — the location of the failure description in the trace file: **4563** line;
- **scenario** — the test scenario detecting the failure: **account_scenario**;
- **state** — the test scenario state preceding the failure occurrence: **-3**;
- **transition** — the scenario function and the values of its iterated variables corresponding to the failure occurrence: **withdraw_max_scen()**;
- **specification function** — the specification function detecting the failure: **withdraw_spec()**;
- **parameter value** — the values of the arguments of the specification function detecting the failure: **acct = <0049638C>ptr to struct { -3 }**;
- **coverage & branch** — the branches of the specification function coverages corresponding to the failure occurrence: **C**, **Maximal withdrawal**;
- **prime formula** — the values of prime formulae corresponding to the failure: all invariants and **`reads`** access restrictions are `true`.

An information concerning a failure could be also found in the detailed scenario report.

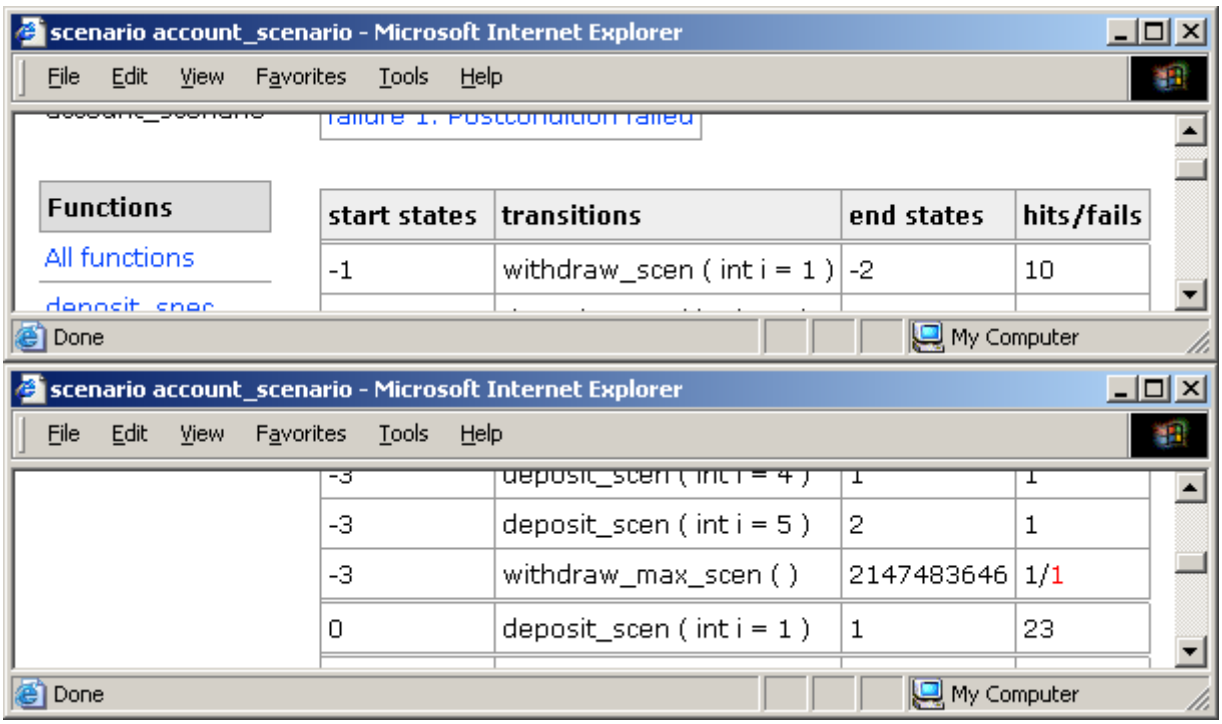

Figure 24. A failure in the detailed scenario report.

The reports show that in the state **-3** and the withdrawn amount **2147483647** the `withdraw_scen` function returns **2147483647** and the balance value after the call is **2147483646**. Although the `withdraw_scen` function postcondition states that in this case the balance should not be changed and the return value should be zero:

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

The implementation can be found in the **account.c** file located in **examples\account** of the CTesK tree. The implementation of the `withdraw` function is:

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

That is, if `acct->balance` is negative and `sum` more than `INT_MAX + acct >balance + 1` the overflow occurs in the expressions `acct->balance - sum` and `acct->balance -= sum`. The fixed code is:

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

In this implementation the overflow is not occured, and the function work meets the requirements.

Please, rebuild the test with the fixed implementation, run it and regenerate reports. Reports should show no failures and 100% of coverage of the both functions.

Appendix A: Using CTesK on Windows

Microsoft Visual Studio 6.0 Project Configuration

CTesK test development project in the Visual Studio is created as '**Win32 Console Application**'.

An extension of files containing the SeC language constructs should be **.sec** or **.seh**.

CTesK Toolbar

When you start Visual Studio CTesK toolbar should appear. It contains buttons to launch '**Mediator Wizard**', '**Scenario Wizard**' and '**Trace Analyzer**'.

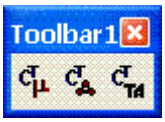

Figure 25. CTesK Toolbar.

If CTesK toolbar is not appeared please refer to the section "*Known installation issues*" of the document "*CTesK 2.2: Installation Instructions*".

Test Building

The test building can be started using the Visual Studio '**Build**' command: choose the '**Build <file>.exe**' item in the '**Build**' menu or press **F7** key.

Test Execution

The test execution is performed using the '**Execute**' command: choose the '**Execute <file>.exe**' item in the '**Build**' menu or **Ctrl+F5** keys. The test trace configuration options and user-defined options can be set using the project settings window. To open this window select the root folder of the project on the tab '**FileView**' in '**Workspace**' window, choose '**Settings…**' item in '**Project**' menu or type **Alt+F7**. '**Projects Settings**' window should appear. Set trace configuration options and user-defined options in '**Program arguments**' field of the '**General**' category of the '**Debug**' tab.

The test tracing is affected by the following options:

-t <trace file> — trace will be directed to the file **<trace file>**;

-tc — trace will be directed to the console;

-tt — trace will be directed to the file **<scenario name>--YY-MM-DD--HH-MM-SS.utt**.

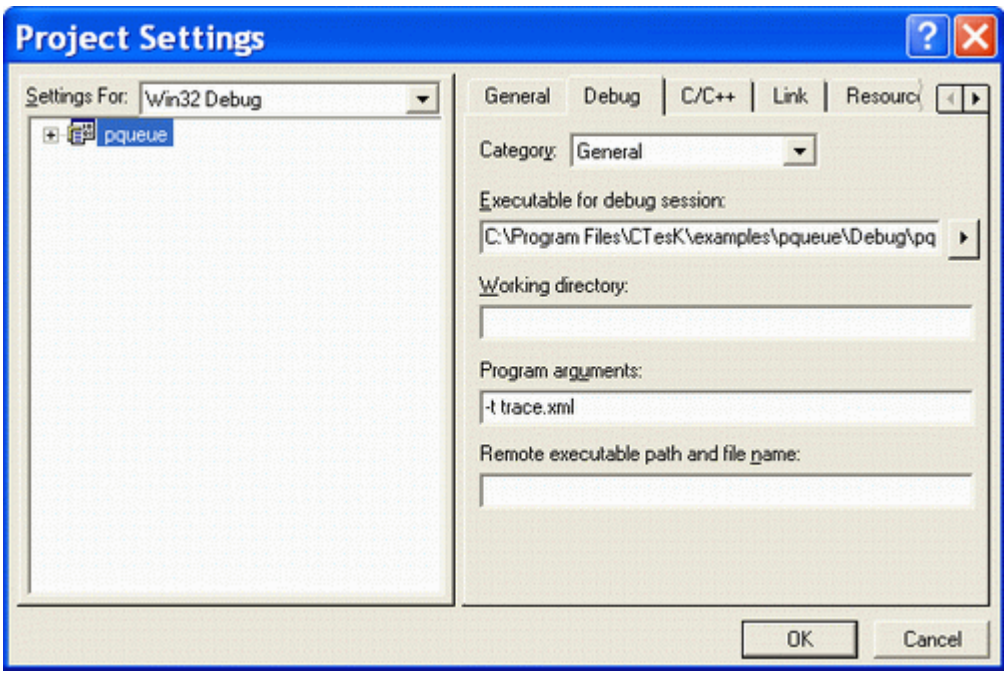

Figure 26. Program arguments.

Test Report Generation

Open the file containing the test trace in Microsoft Visual Studio and press the 'C^{T}_{TA}' button of the '**CTesK Toolbar**' menu. HTML test report should be generated and Internet Explorer contained the first page of the report window should be launched.

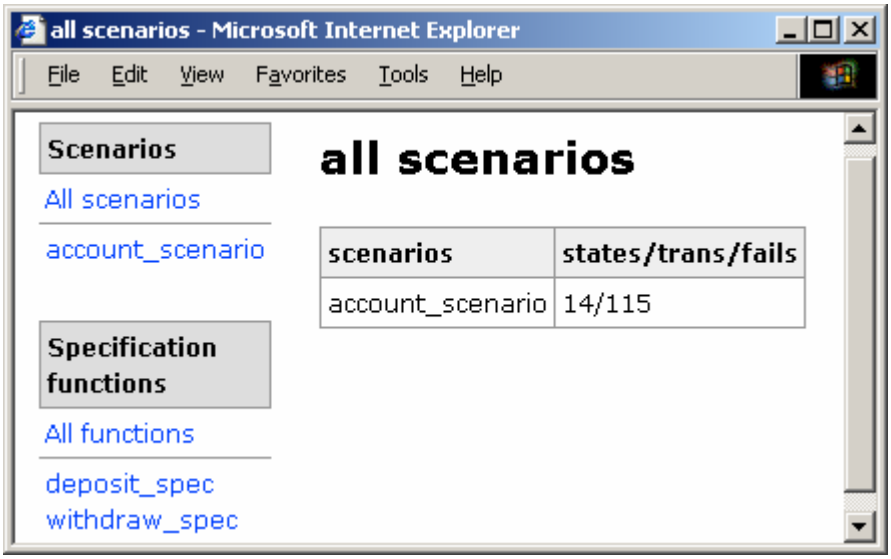

Figure 27. Start page of HTML test report.

Using CTesK in command line

To use CTesK in the command line mode execute **sec.bat** and **ctesk-rg.bat** command files. They are located in the **bin** folder of the CTesK installation folder. To open Command prompt under Windows choose '**Start Menu\(All)Programs\Accessories\Command Promt**'.

Translator can be launched in the following form:

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

Here **<sec file>** is an input specification file, **<c file>** is an output file, and **<sei file>** is an intermediate file for preprocessor output. The tool will generate **<c file>**.

The HTML test report is generated using the following command:

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

Here **<output folder>** is the folder where report files will be placed, **<trace files>** is the list of files containing test execution traces.

Using CTesK with Cygwin

CTesK could be used with Cygwin environment (see the section "*Appendix A: Using CTesK with GCC compiler*" of the document "*Using CTesK 2.2 with GCC: Getting started*"). You can use **sec.sh** to translate SeC files into C files and **ctesk-rg.sh** to generate HTML reports. It is recommended to use GNU Make program to build tests.

If you cannot find **sec.sh** and **ctesk-rg.sh** in **bin** folder of the CTesK installation folder please refer to the section "*Known installation issues*" of the document "*CTesK 2.2: Installation Instructions*".