Using CTesK 2.2 with GCC
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 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, before further reading this document, please, turn to "*CTesK 2.2: Installation Instructions*" and install CTesK 2.2.

The implementation of a bank credit account is located in the **account.c** file in the **examples/account**[1] directory 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.

[1] For Windows platforms slashes '/' in paths should be replaced by back-slashes '\'.

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[2], 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** directory 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:

2 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[3], 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 [4]. 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.

---

[3] 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.

[4] 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 C code.

To translate the **account_model.sec** file into C code on Unix platforms launch command shell, go to the **examples/account** folder in the CTesK tree and run the command
>sec.sh account_model.sec account_model.c account_model.sei

On Windows platforms one should run the following command in the **examples\account** directory of the CTesK tree
>sec.bat account_model.sec account_model.c account_model.sei

As a result of the translation the **account_model.c** file should be generated in **examples/account**.

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 mediators for the account example can be found in the **account_mediator.sec** file located in the **examples/account** directory of the CTesK tree.

The **account_mediator.sec** file starts with including the **account_mediator.seh** file located in **examples/account**.

```
#include "account_model.seh"

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

Besides the forward declarations of the mediators functions this header file also contains including the **account_model.seh** specification header file that in turn includes the **account.h** implementation header file. Thus mediators are able to deal with both the implementation and the specification.

The **account_mediator.sec** file contains the definition of two mediator functions.

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

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

The first one provides binding between the `deposit_spec` specification function and the `deposit` interface function of the implementation. The second binds the `withdarw_spec` specification function to the `withdraw` interface function of the implementation.

The signature of a mediator function should contain the **`mediator`** keyword, the mediator function name, the **`for`** keyword and the signature and access restrictions of the specification function to be bound.

The body of a mediator function for a specification function must contain a block statement marked with the **`call`** keyword.

The **`call`** block implements the functionality described in the corresponding specification function by means of calling the corresponding interface function of the implementation. It means that

- the parameters of the mediator function, which are the actual parameters of the specification function ones, should be transformed to the parameters of the interface implementation function;
- the obtained values of the parameters should be passed the interface implementation function;
- and the returned value and the output values of the parameters changed in the result of the call should be transformed to the value returned by the mediator function and the output values of the corresponding mediator function parameters.

To translate the **account_mediator.sec** file into C code on Unix platforms launch command shell, go to the **examples/account** folder in the CTesK tree and run the command
>sec.sh account_mediator.sec account_mediator.c account_mediator.sei

On Windows platforms one should run the following command in the **examples\account** directory of the CTesK tree
>sec.bat account_mediator.sec account_mediator.c account_mediator.sei

As a result of the translation the **account_mediator.c** file should be generated in **examples/account**.

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** directory of the CTesK tree.

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

AccountModel Acct;

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

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

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

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

As far as the implementation, specifications and mediators are used, the corresponding header files should be included into the scenario. Since the **account.h** file is included into the **account_model.seh** file, which in turn is included into the **account_mediator.seh** file, only the last header file is included into the **account_scenario.sec** file.

The scenario is developed to test the account system with the only instance of the account structure. The instance is implemented as a global variable of the scenario.

```
AccountModel acct;
```

Next the test initialization function follows.

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

The `account_init` function initializes the test scenario state setting the `balance` field to `0`, sets the mediators of the specification functions of the scenario and returns `true`. The functions `set_mediator_deposit_spec` and `set_mediator_withdraw_spec` are used to set the mediator functions for the corresponding specification functions.

A function that sets a mediator of the specification function is implicitly defined when defining the specification function. It has a name `set_mediator_`*`<specification function name>`*.

The `account_state` function defines a set of states which are considered as different in the test scenario. In the account test scenario states are different when values of account balance are different.

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

The behavior of the system under test should be tested in various situations. In other words the test should call the interface functions in different scenario states. In the scenario of the account example test situations are distinguished by the current value of the balance. Therefore the scenario state is defined as an integer value of `acct.balance`. CTesK requires a scenario state type to be a specification type. In the account example the scenario state type is the library integer specification type that is defined in the **atl/integer.h** header file.

In each reachable scenario state the test should check behavior of each function in each functional branch available in the state. In CTesK a set of simple test actions that will be performed by the test engine in each reachable scenario state is defined in *scenario functions*. In the account example there are two scenario functions.

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

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

The `deposit_scen` function is intended for testing of the `deposit` function. The `withdraw_scen` function is intended for testing of the `withdraw` function. They are named `deposit_scen` and `withdraw_scen` respectively.

The **`iterate(;;;)`** statement is used to enumerate the actions. Its syntax is similar to **`for(;;)`** statement except the last additional field that is a filtration condition. In the body of an **`iterate`** statement actions to be performed are defined. Iterate statements can be nested one within another.

Both scenario functions have similar structure. They iterate an integer value and call the corresponding specification function passing the pointer to the account structure and the iterated value as its arguments. The only difference is a stop condition appeared in the `deposit_scen` scenario function. It is intended to prevent test engine from infinite or simply too big number of `deposit_spec` function calls — without stop condition the balance will grow until `INT_MAX`. The stop condition states that the `deposit_spec` function should be called only if the balance value is less than or equal to `5`. The `withdraw_spec` function does not require a stop condition because it has natural limitation — the maximal possible credit value.

Scenario functions can perform additional checks of behavior of the system under test, which are based on additional knowledge of the scenario about an environment state. The result of this checking should be returned.

In the account example all requirements are described in the specification, hence there is no necessity for additional checks. Therefore the scenario functions return the `true` value.

Finally the scenario itself is defined.

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

In SeC test scenario is defined by the declaration of the global variable marked with the keyword **`scenario`**. The type of the global variable corresponds to the type of test engine used by the scenario. In the scenario of the account example test engine of the `dfsm` type is used[5]:

This type is a structure with the following fields:

- `init` is a pointer to the function of the type `bool (*PtrInit)(int, char**)`;
- `state` is a pointer to the function of the type `Object* (*PtrGetState)(void)`, `Object` is a special specification library type, the reference of any specification type can be used as the reference of the `Object` type without an explicit cast;
- `actions` is an array of pointers to *scenario functions* ended by `NULL`;
- `finish` is a pointer to the function of the type `void (*PtrFinish)(void)`.

The `init` field is initialized by the `account_init` function.

The `getState` is initialized by the `account_state` function.

The `actions` field is initialized by the array containing two scenario functions — `deposit_scen` и `withdraw_scen`.

The `finish` field should be initialized with the pointer to the function that finalizes the test. The scenario of the account example does not need finalization actions. Therefore the `finish` field is not initialized.

To translate the **account_scenario.sec** file into C code on Unix platforms launch command shell, go to the **examples/account** folder in the CTesK tree and run the command
>sec.sh account_scenario.sec account_scenario.c account_scenario.sei

On Windows platforms one should run the following command in the **examples\account** directory of the CTesK tree
>sec.bat account_scenario.sec account_scenario.c account_scenario.sei

As a result of the translation the **account_scenario.c** file should be generated in **examples/account**.

[5] The ndfsm test engine type is also implemented in the CTesK 2.2.

Running test of
Bank credit account example

The last component of the account example can be found in the **account_main.sec** file located in the **examples/account** directory of the CTesK tree. It contains definition of the main function that obtains command line arguments of the test and launches the test scenario with these arguments. The scenario is declared in the **account_scenario.seh** header file.

```
#include "account_scenario.seh"

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

The header file **account_scenario.seh** contains the declaration of the extern scenario `account_scenario`.

```
extern scenario dfsm account_scenario;
```

The `main` function starts the scenario with the command line options as its arguments.

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

To translate the **account_main.sec** file into C code on Unix platforms launch command shell, go to the **examples/account** folder in the CTesK tree and run the command
>sec.sh account_main.sec account_main.c account_main.sei

On Windows platforms one should run the following command in the **examples\account** directory of the CTesK tree
>sec.bat account_main.sec account_main.c account_main.sei

As a result of the translation the **account_main.c** file should be generated in **examples/account**.

The next step is compiling all C files to object files and linking them into the executable file. This task is performed by standard for C compiler used way. The only additional requirement is to link the CTesK static libraries to the executable file. These libraries are **libatl.a**, **libts.a**, **libtracer.a** and **libutil.a**. They are placed in the **lib/<platform>** directory of the CTesK tree.

To build the executable file by means of GCC one should run in the **examples/account** directory of the CTesK tree the GNUmake
>make

As a result, the executable file **account** (or **account.exe**) should be built.

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[6]:

-t <file-name> — trace will be directed to the file '**<file-name>**'

-tc — trace will be directed to the console

[6] 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**'.

-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.utt** file.

On Unix platforms go to the directory containing the executable file (**examples/account**) and run the command:
>account –t trace.utt

On Windows platform go to the folder containing the executable file (**examples\account**) and run the command:
>account.exe –t trace.utt

As a result of the test execution the **trace.utt** file should be generated.

Test result analysis of Bank credit account example

Text test report generation

To generate report on UNIX platforms launch command shell, go to the directory containing the trace file (**examples/account**) and run the command
>ctesk-rg.sh –d trace.report trace.utt

To generate report on Windows platforms launch command shell, go to the folder containing the trace file (**examples\account**) and run the command
>ctesk-rg.bat –d trace.report trace.utt

As a result, the directory **trace.report** should be generated.

Open in the directory **trace.report** the file **index.html** to see the start page of the report set. The reports' navigation bar is placed on the left side of the start page.

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.

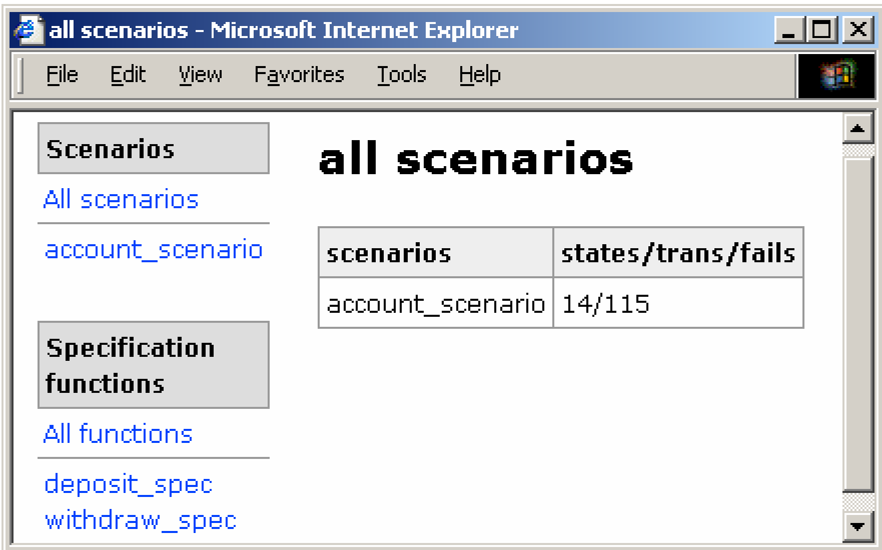
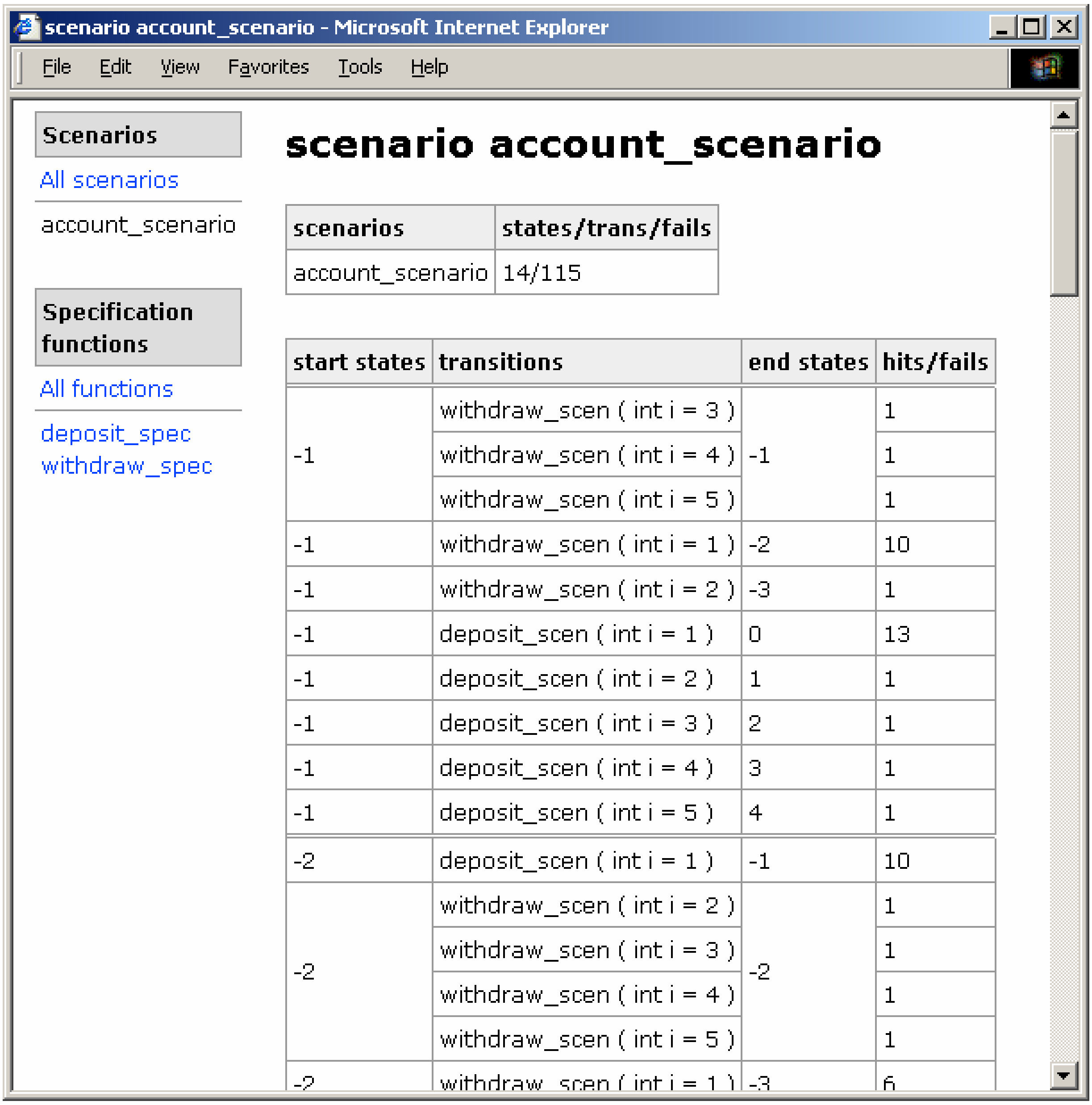


Figure 1. The summarized test report.

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.


scenario account_scenario - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Scenarios

All scenarios

account_scenario

Specification functions

All functions

deposit_spec

withdraw_spec

scenario account_scenario

scenarios	states/trans/fails
account_scenario	14/115

start states	transitions	end states	hits/fails
-1	withdraw_scen (int i = 3)	-1	1
	withdraw_scen (int i = 4)		1
	withdraw_scen (int i = 5)		1
-1	withdraw_scen (int i = 1)	-2	10
-1	withdraw_scen (int i = 2)	-3	1
-1	deposit_scen (int i = 1)	0	13
-1	deposit_scen (int i = 2)	1	1
-1	deposit_scen (int i = 3)	2	1
-1	deposit_scen (int i = 4)	3	1
-1	deposit_scen (int i = 5)	4	1
-2	deposit_scen (int i = 1)	-1	10
-2	withdraw_scen (int i = 2)	-2	1
	withdraw_scen (int i = 3)		1
	withdraw_scen (int i = 4)		1
	withdraw_scen (int i = 5)		1
-2	withdraw_scen (int i = 1)	-3	6

Figure 2. 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.

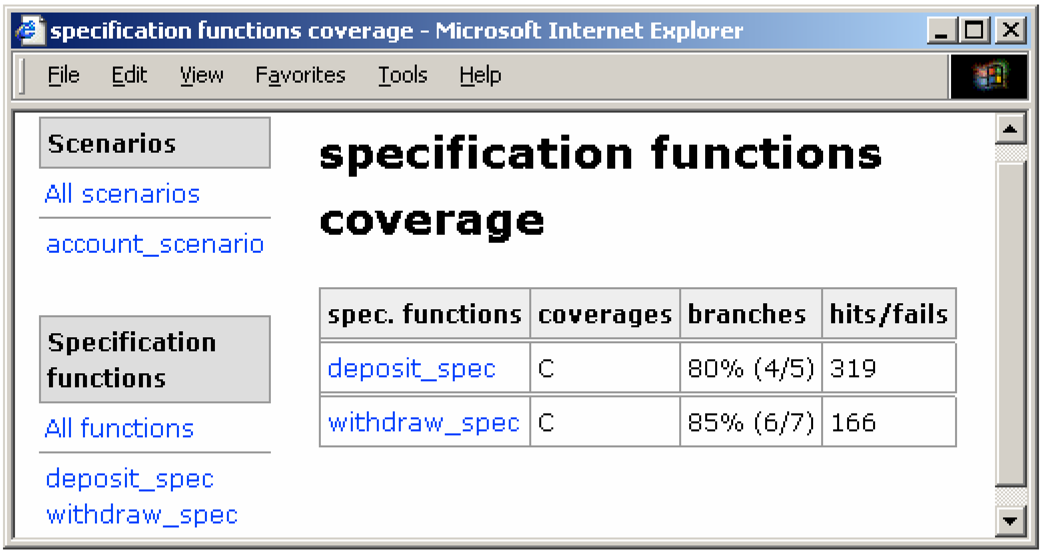


Figure 3. The summarized function coverage report.

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.

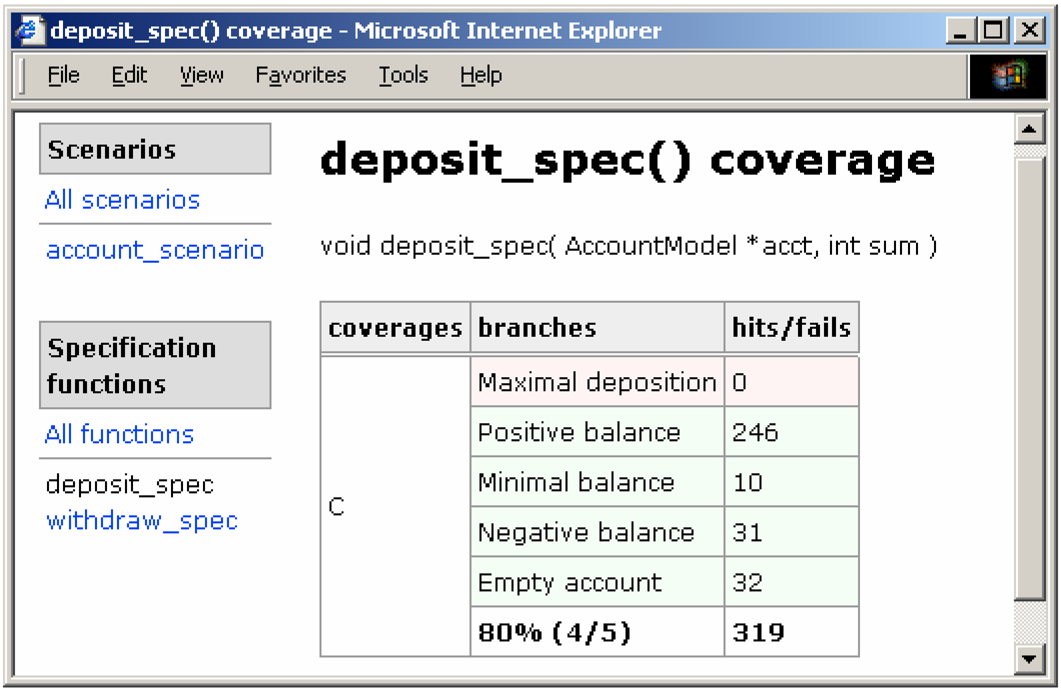
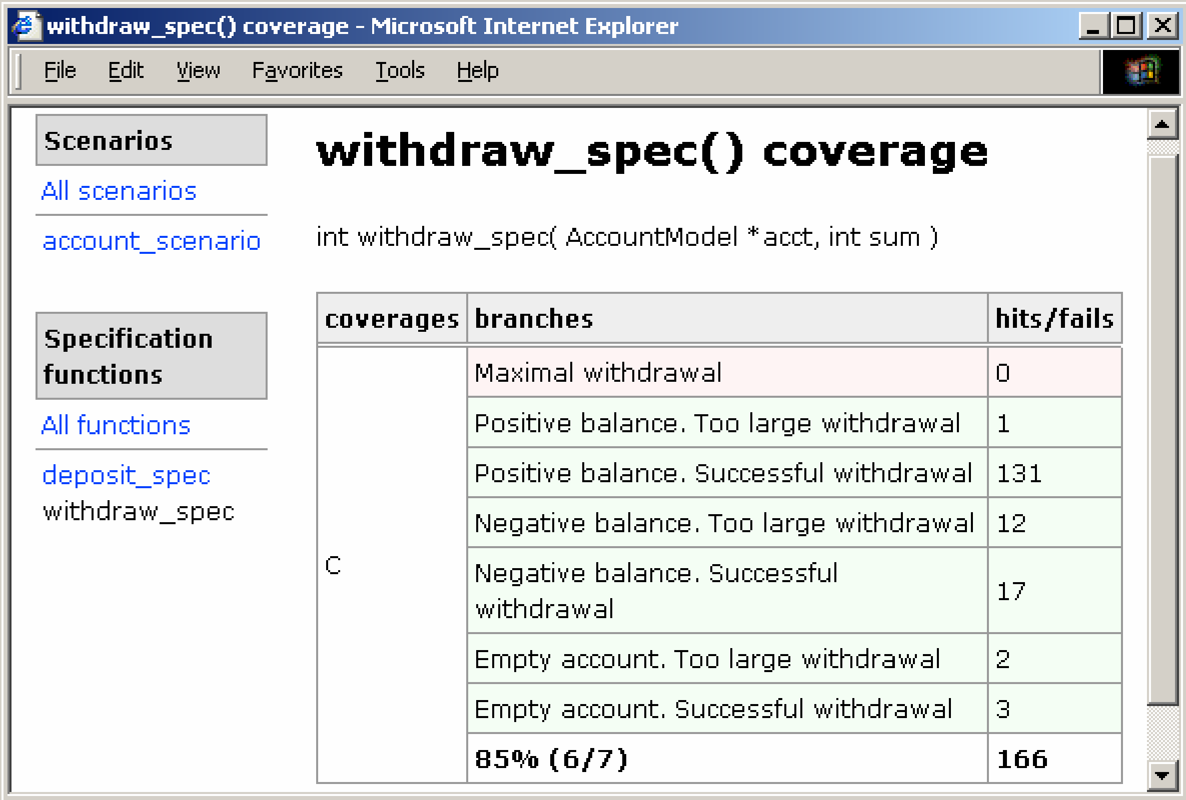


Figure 4. The detailed function coverage report of `deposit_spec` function

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**.



withdraw_spec() coverage

int withdraw_spec(AccountModel *acct, int sum)

coverages	branches	hits/fails
C	Maximal withdrawal	0
	Positive balance. Too large withdrawal	1
	Positive balance. Successful withdrawal	131
	Negative balance. Too large withdrawal	12
	Negative balance. Successful withdrawal	17
	Empty account. Too large withdrawal	2
	Empty account. Successful withdrawal	3
	85% (6/7)	**166**

Figure 5 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:

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

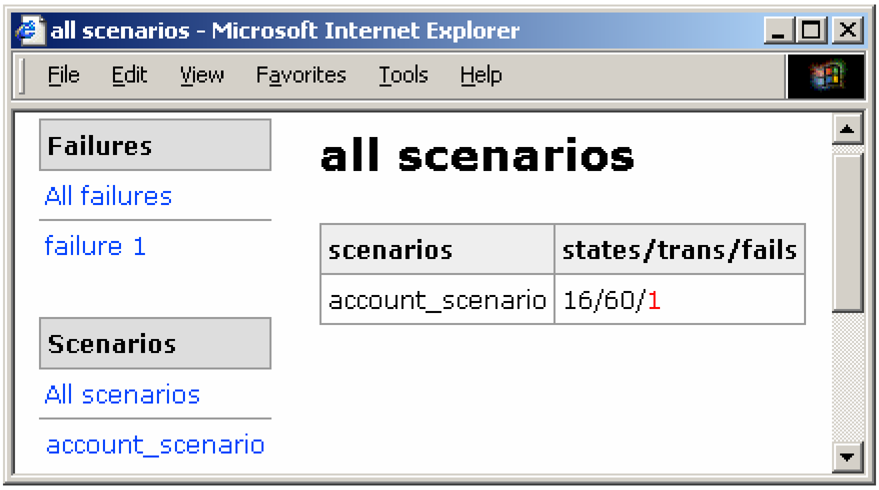


Figure 6 The summarized test report containing the failures

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.

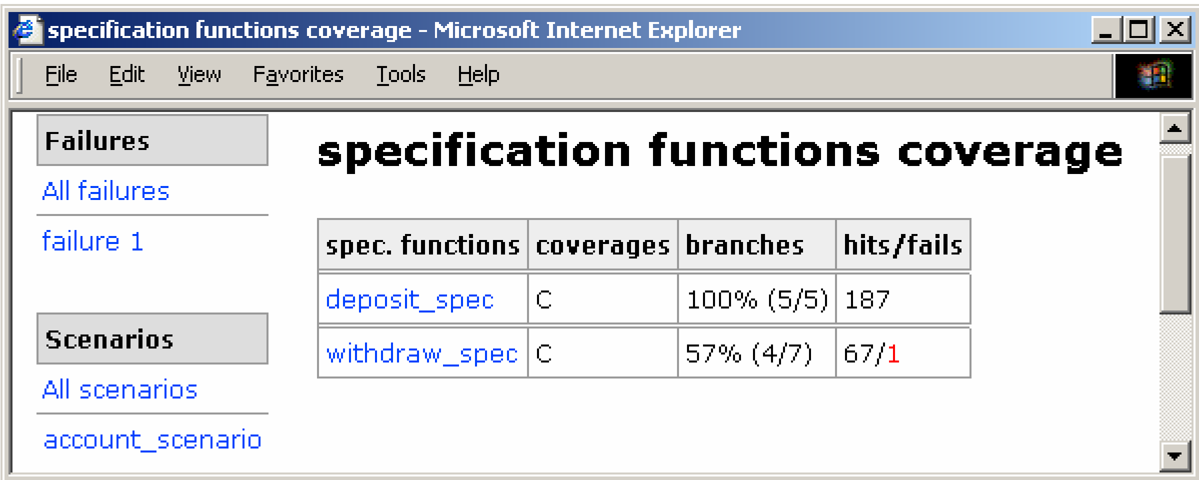


Figure 7. The summarized function coverage report after scenario changes.

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.

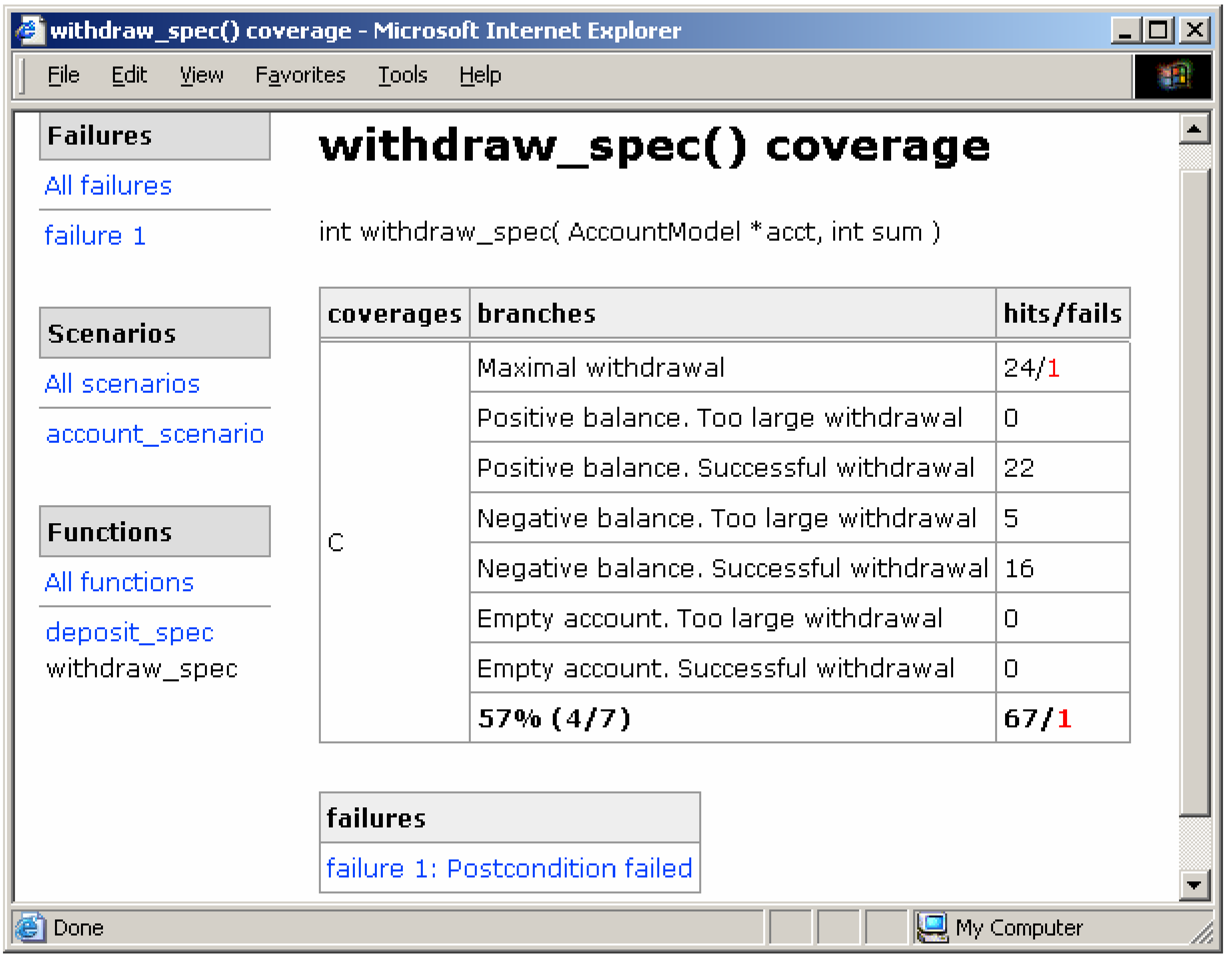


Figure 8. The detailed function coverage report of `withdraw_spec` function after scenario changes.

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.

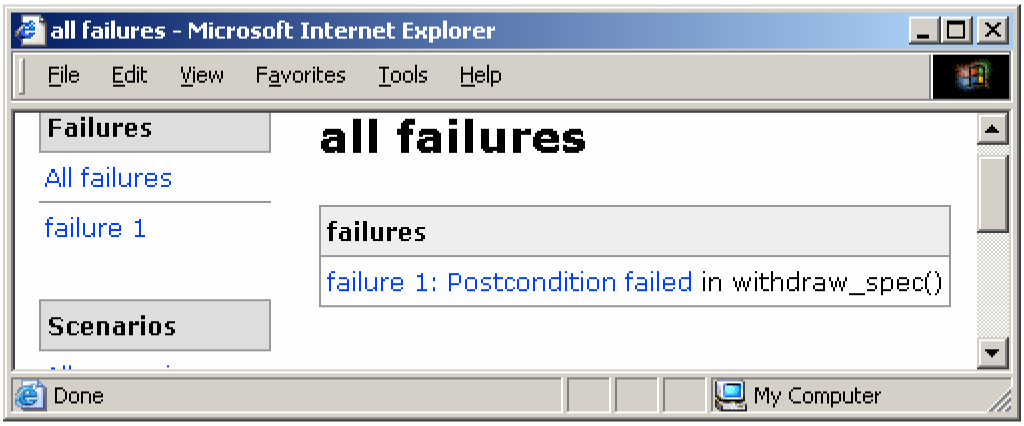


Figure 9. The summarized failure report.

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.

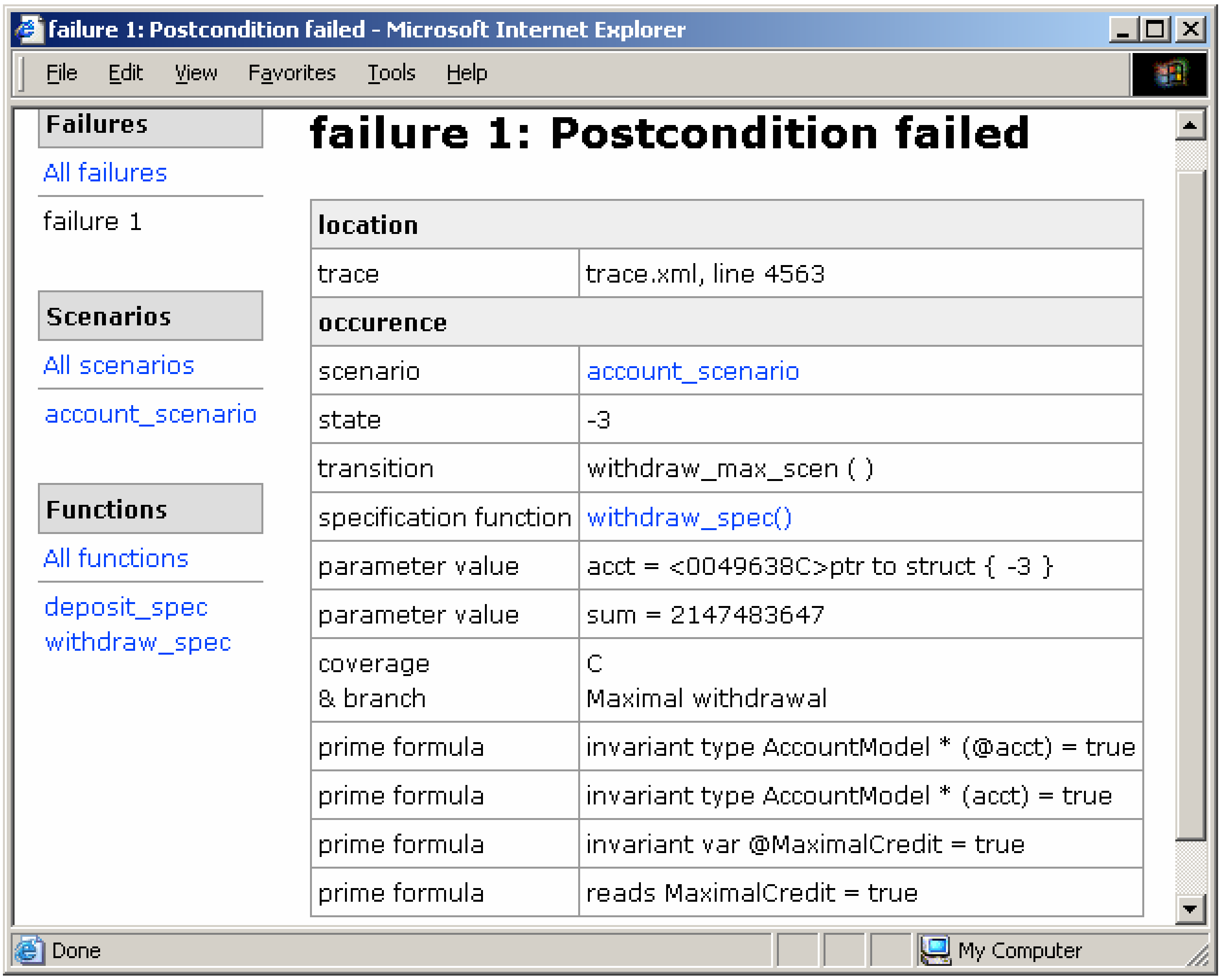


Figure 10. The detailed failure report for the erroneous implementation

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.

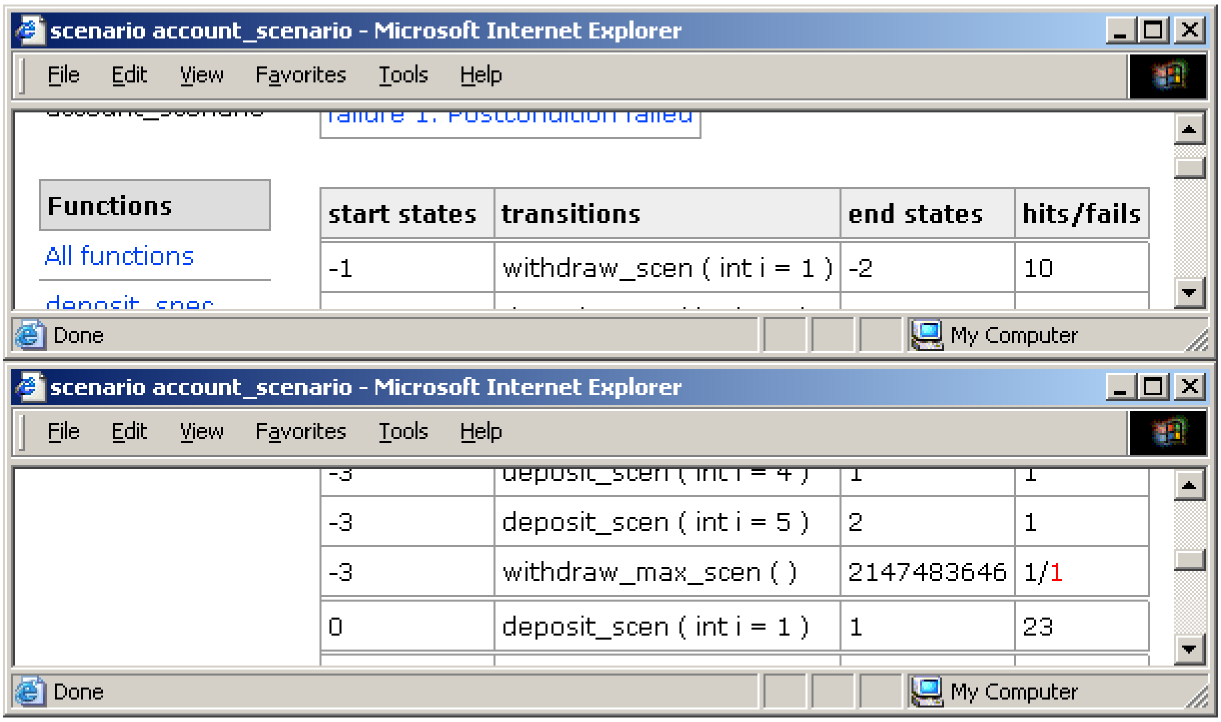


Figure 11. A failure in the detailed scenario report.

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 with GCC compiler

SeC translator is located in the **bin** directory of the CTesK installation directory[7]. Translator can be launched in the following form:

> **> sec.sh [options] <sec file> <c file>**

It translates the SeC file **<sec file>** into the C file **<c file>**.

The option **--sei <sei file>** sets an intermediate file to use. This option could be omitted. All other options are passed to preprocessor.

Using GNU Make to Build Test

To build a test developed with the CTesK, GNU Make program can be used. To simplify a make file creation, its template contained in the CTesK distribution can be used. It is located in the **examples/example.make** file.

To use it, a new make file (Makefile or GNUmakefile) should be created in the test's directory.

In this file, the following variables should be defined:

sec_sources

> This variable should contain a whitespace-separated list of **.sec** files that are developed for the test.

c_sources

> This variable should contain a whitespace-separated list of **.c** files that should be linked with the test.

example

> This variable should contain a name of the executable test file.

Then makefile located in the CTesK distribution should be included using the **include** directive:

> **include $(CTESK_HOME)/examples/example.make**

After that GNU Make program should be run using **make** or **gmake** program (depending on Linux distribution).

The **XINCLUDE** and **XLIB** variables allow to specify additional include files paths (**-I<path>**) and additional libraries and their locations (**-l<lib>** and **-L<path>**).

Examples of make files can be found in **examples/account**, **examples/pqueue**, and **examples/stack** directories.

Test Execution

Run in the directory containing an executable file of the test the command

[7] When using CTesK on the Window platform with Cygwin environment, if you cannot find **sec.sh** in **bin** directory of the CTesK installation directory please refer to the section "*Known installation issues*" of the document "*CTesK 2.2: Installation Instructions*".

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

<test file> is an executable test. **<trace options>** are test trace configuration options. **<options>** are options defined by user during a test development. 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**.

Test Report Generation

CTesK report generator **ctesk-rg.sh** is located in the **bin** directory of the CTesK installation directory[8]. To view the test execution report in human-readable form, run the following command

>ctesk-rg.sh -d <trace directory> <trace file>

As a result, the HTML report should be generated in the **<trace directory>** directory.

Open the file **index.html** in this directory to see the start page of the HTML report.

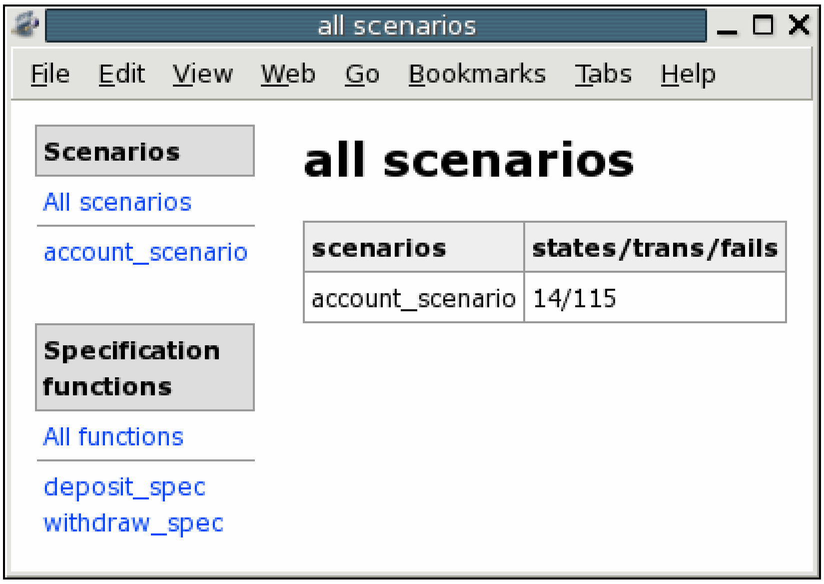


Figure 12. Start page of HTML test report

[8] When using CTesK on the Window platform with Cygwin environment, if you cannot find **ctesk-rg.sh** in **bin** directory of the CTesK installation directory please refer to the section "*Known installation issues*" of the document "*CTesK 2.2: Installation Instructions*".