CTesK 2.2 Community Edition: User's Guide

Moscow
2006

Contents

Introduction

What is CTesK

CTesK toolkit is intended for automated test development for systems that provide API interface in C. Software testing with the help of CTesK tool is based on UniTesK technology.

UniTesK technology

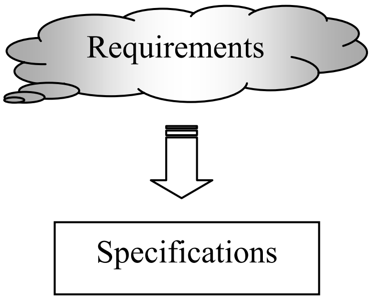
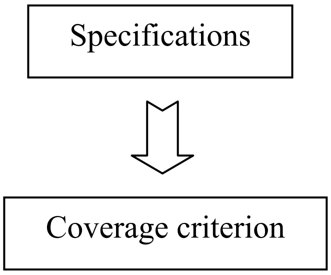
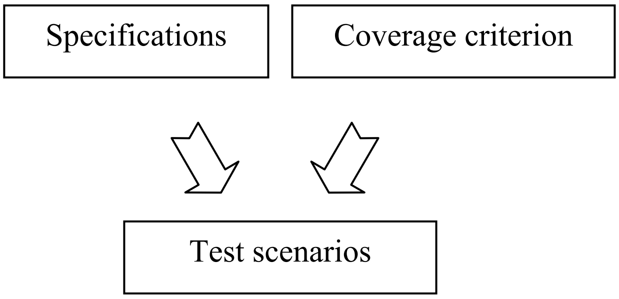
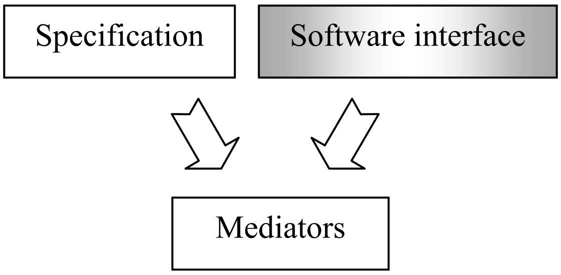
Quality control is an important challenge that faces software engineers. Testing is the best-known and most widespread method of quality evaluation and improvement. Functionality and complexity of modern software grow rapidly, while its life time increases permanently. Under such conditions labor-intensiveness of traditional approaches to testing grows while their efficiency decreases. Using of UniTesK technology helps to overcome such problems.

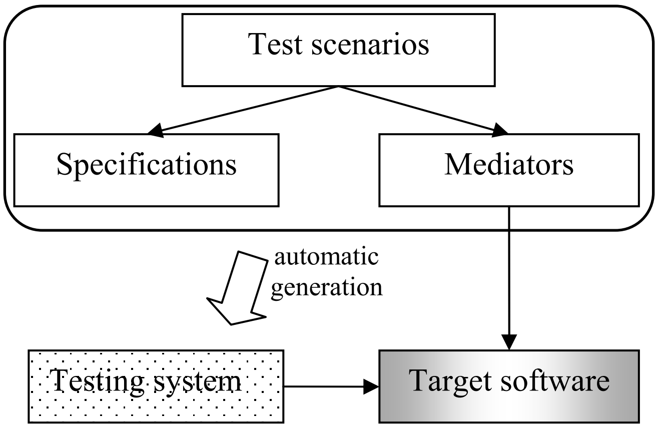
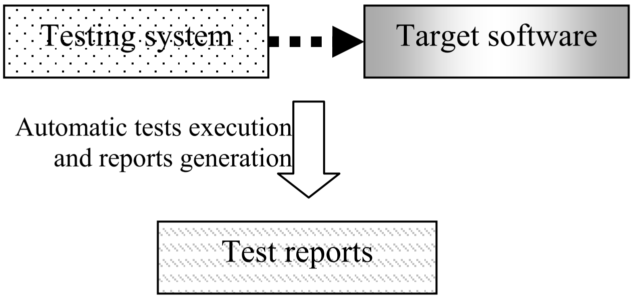
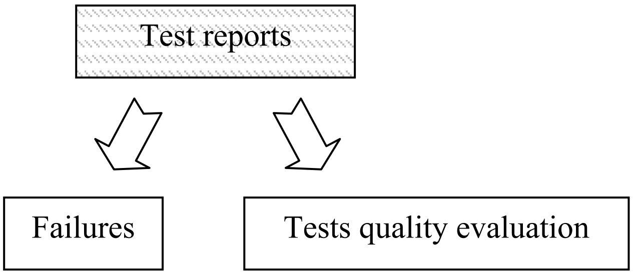
The main features of UniTesK technology are as follows:

- In order to provide clear definition of software functionality *formal specifications* are developed. This may be done both for newly developed software, even prior to completion of implementation, and for already existing one. Therefore, the technology may be applied to the tasks of both forward and reverse software engineering.
- Tests are developed on the base of formal specifications instead of implementation. This allows checking for conformance of the software behavior to its requirements. This type of testing is called the "black box" testing. It provides an opportunity to develop a set of tests that does not take into account peculiarities of a specific implementation.
- *Test coverage criteria* are also created on the base of formal specifications. These criteria allow evaluation to what extent the conformance of the software behavior to the requirements has been checked.

- A *test scenario* is constructed to achieve maximum coverage according to a criterion chosen. Widely used test scripts are analogous to test scenarios. However, UniTesK test scenarios give a possibility to improve significantly quality of testing within the same effort.
- Formal specifications and test scenarios may be used in invariable form for testing of various implementations even if their interfaces differ. Binding of an implementation and tests is provided by specific test system components—*mediators*. This approach allows increasing of the degree of reusing the test system components, thus facilitating test suite engineering and maintaining.
- All components of the test system, i.e. formal specifications, mediators, and test scenarios, are recorded in specification extension of programming language used for software development. This significantly facilitates familiarization with the technology and understanding of how the test in connected with the system under test.

The following table describes actions necessary for test development by UniTesK technology:

1. Write the functional requirement to the target software in form of formal specifications, based on analysis of existing documents or project members' knowledge.	
2. Formulate testing quality requirements on basis of derived specifications, i.e. what level of coverage for each criterion is sufficient for testing.	
3. Develop a set of test scenarios to achieve a desired level of coverage. Scenarios are developed on basis of specifications and are not connected to any particular implementation of target software or its particular version.	
4. Develop a set of mediators to connect obtained tests to a particular implementation of the target system. A definite interface of the implementation must be known, thought the implementation can not be ready for testing at the moment.	

5. Translate specifications, mediators, and scenarios from the extension of programming language to a complete test system in this programming language in order to obtain ready tests.	 Test scenarios Specifications Mediators automatic generation Testing system Target software
6. Execute tests after translation. Execution can reveal inconsistencies between the test system and the target software. One should determine the cause of each inconsistence, which can be either a target system failure, or test system components failure. After all failures in specifications, scenarios, and mediators are corrected, all tests should be finally executed to obtain the results in form of test reports for further analysis.	 Testing system Target software Automatic tests execution and reports generation Test reports
7. Analyze the results to determine what failures is detected, whether the desired level of coverage is reached, and whether additional scenarios should be developed.	

Stages of test scenarios and mediators development are independent, so steps 3 and 4 can be carried out in any order or even simultaneously.

UniTesK implementation in CTesK

CTesK implements UniTesK for C programming language.

CTesK uses specially developed specification extension of C programming language, called *SeC*, for test development. SeC extends C with special constructions, introduced to describe requirements to a system under test and other components of the test system in a compact and convenient manner. This makes test development maximally comfortable, and allows reduction of training costs for specialists who are already familiarized with C. SeC enables development of specifications and scenarios, absolutely independent from the implementation, thus allowing their reuse.

CTesK toolkit includes SeC-to-C translator, test system supporting library, library of specification data types, and test reports generators.

SeC-to-C translator generates test components from specifications, mediators, and tests scenarios.

Test system supporting library provides the test engine, i.e. implementation of algorithms for test sequence generation in C, and provides tracing of tests execution.

Specification data types library supports data types integrated with standard functions for creating, copying, comparing, and destroying data of these types. It also contains a set of predefined specification data types.

Generators of textual and graphical test report generates easy-to-analyze representations of test execution trace.

What is contained in the document

The "*SeC Language*" chapter gives a detailed description of specification extension of C.

- "*General Information*" section lists distinctions of SeC from C, and presents C-extending concepts and constructions.
- In "*Data Types*" section, a concept of *allowable* and *specification data types* is introduced, operation with existing specification data types and rules for creating new data types are considered in details. The mechanism of *invariants* for data types and variables is described.
- In "*Specifications*" section, the method of formal description of requirements for a system under test in the form of *preconditions*, *postconditions*, and *access constraints* within *specification functions* and *deferred reactions* is reviewed. The method of specifying a *coverage criterion* by means of *functionality branches* is described.
- "*Mediators*" section explains the way of connecting specification to a system under test implementation in the form of *mediator functions*, which carry out a *test action* and *state synchronization*.
- In "*Scenarios*" section, building of a *test scenario* is described, which unites the set of *scenario functions* for parameter iteration, a test construction mechanism, the function for evaluating *scenario state*, and the method of initialization and finalization of the test system and the system under test.

In "*Test results analysis and tests debugging*" chapter, the issues associated with analysis of test execution data are reviewed.

- "*Test trace*" section describes the format of a *trace*, generated during test execution.
- "*Static reports*" section reviews html reports, which contain the information on the failures found, on coverage of functionality branches of specification functions, on traversed finite state machine structure.
- "*Graphic representations of the trace*" section considers various representations of a trace as a sequence of events that has occurred in the process of test execution.

- "*Analysis of results*" section describes various messages, which may appear in a report, as well as the methods of finding failures in accordance with the information obtained. The method of evaluating achieved coverage is described.

In "*Examples of CTesK usage*" chapter, completely described examples of different systems testing are collected.

- "*Systems that provide API*" section reviews testing of programs that provide application program interface, by the example of a system for queue operations.

Other documents

Additional information CTesK and supported test development technology can be found in other documents, included in the CTesK 2.2 Community Edition documentation set: "*CTesK 2.2 Community Edition: Installation Instructions*", "*CTesK 2.2 Community Edition: Getting Started*", "*CTesK 2.2 Community Edition: SeC Language Reference*".

Our site www.unitesk.com contains information on UniTesK, CTesK, and other tools supported UniTesK.

Any questions on UniTesK technology and CTesK usage can be addressed to support@unitesk.com.

Conventions

Terms of the main concepts along with parts of the text, containing important information, are shown in *italics*.

Links to sections of this document and to other CTesK documents are shown as "*italicized text in quotes*".

```
Examples of SeC code are shown in formatted paragraphs.
```

Fragments of code in the main text are shown in `fixed-width font`. SeC keywords are shown in **`semi bold fixed-width font`**.

Menu items, commands, and files and directories names are shown in **semi bold font**.

SeC language

General information

SeC completely supports ANSI C standard. Additionally, *specification data types*, *types* and *variables with invariants* are introduced, along with four kinds of functions: *specification function*, *reactions*, *mediator functions*, and *scenario functions*. These types, invariants and functions are defined in specification files with **.sec** extension. Specification header files, that contain declarations of specification types and functions should be located in the files with **.seh** extension.

Specification header files are included to specification files by means of `#include` C preprocessor command. Specification files may also contain usual C functions, required for various auxiliary purposes. When use of data types, constants, variables and functions is necessary, usual C header files are included.

For the convenience of writing and reading of logical expressions, the implication operator **=>** is additionally introduced in SeC, which is a binary infix operator, whose priority is below that of the disjunction operator `||`, but is above the priority of the conditional operator `?:`. The expression `x => y` is equivalent to the expression `!x || y`, and in the process of its evaluation, similar to evaluation of other logical operators, the rules of short logic are applied. The implication operator is associative from left to right, i.e. the expression `x => y => z` is equivalent to the expression `(x => y)` **=>** `z`.

Data types

SeC language fully supports C data types. Besides, additional data types and their kinds are introduced in SeC:

- *Boolean type* `bool` for presentation of logical expressions, and `true` and `false` constants.
- *Specification types*, that integrate the types of C with basic operations with the data of the types: creating, copying, comparing, destroying (refer to subsection "*Specification types*").
- *Invariant types* or *subtypes* are the data types, which ranges are the subranges of other data types. The latters are called supertypes for these data types. A subrange is defined by means of constraints, specified in type invariant (refer to subsection "*Type invariants*").

SeC allowable types

Only the following data types are acceptable for arguments and return values of specification functions, deferred reactions, and mediator functions, for iteration variables and scenario state variables, and for global variables used in the above functions:

- *Arithmetical types* (`int`, `char`, `double`, ...).
- *Enumerated types* (`enum`), the range of which, unlike C, is limited to their constants and may not contain an arbitrary integer value.
- *Typed pointer*. A pointer to any allowable type. The pointer is assumed to be either equal to zero, or point to a single value of a corresponding type. Structure of pointers must be tree-type, i.e. it must not contain undirected cycles.
- *Non-typed pointer* (`void*`). Values of the type are interpreted simply as an address of a certain memory cell.
- *Functional pointer*. Values of the type are interpreted simply as an address of a certain function.
- *Structural type*. The structure should be of a complete type, i.e. the definition of its body must be visible in the point of its usage. Structure fields must be of allowable types.
- *Fixed length array*. The elements of the array must be of an allowable type.

The same constraints are applied to the basic types in definitions of the types with invariants, as well as to the types of variables with invariants. Such constraints allow the test system to handle data of such types automatically: to allocate and free memory, copy and compare values.

Hereinafter, the types that are allowable in the above-mentioned cases are called *SeC allowable types*.

Specification data types

Often the constraints imposed by *allowable types* are excessively rigid. For instance, a pointer used as an array reference or recursive data structures with cycles are not *allowable types*. In order to overcome such constraints, the *specification types* are used.

Besides, the *specification types* are necessary for CTesK test system supporting library (refer to "*CTesK test system supporting library*" chapter of "*CTesK 2.2 Community Edition: SeC Language Reference*").

Specification data type integrates a C data type with basic operations to handle data of this type: creating, copying, comparing and destroying.

The values of specification types are always stored in the dynamic memory and accessible only via specification references, i.e. the pointers of corresponding types that must *either be zero*, *or* *point to an allocated and initialized memory*. In other words, *declarations of variables and parameters of the specification types themselves (not references to them), as well as immediate use of specification types in structures, union, and arrays definitions, is not allowed*.

A specification reference automatically gets a zero value if it is not initialized explicitly in the declaration.

Specification references are dereferenced similar to pointers in C, by means of the dereferencing operators `*` and `->`. The result of dereferencing is l-value of the same type as the basic type used in the definition of the specification type, or as the type of the field of the specification structure (refer to "*Creating new specification types*" subsection).

It is not allowed dereferencing of zero references (it will lead to a fault in execution time).

It is not allowed dereferencing of specification references, which are returned by function calls, without assigning returned references into variables (it will lead to memory leaks or a fault in execution time).

```
List* l = create(type_List, type_Integer)/* List – library
                                            specification type
                                          */;
Integer* spec_i/* Integer – library specification type */;
int i;

add_List(create(type_Integer, 1));
i = ~~*get_List(0)~~; /* invalid use */
spec_i = get_List(0);
i = *spec_i; /* i is equal to 1 */

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

IntPair* ip = create(type_IntPair, 1, 1);
int ai ~~= create(type_IntPair, 1, 1)->a~~; /* invalid use */

ai = ip->a; /* ai is equal 1 */
```

It is not allowed address arithmetic operations over the references and their indexing.

It is allowable to compare addresses in the references by C operators == and `!=`.

It is not allowed comparing specification references, which are returned by function calls, without assigning returned references into variables (it will lead to memory leaks).

```
List* l = create(type_List, type_Integer) /* List – library
                                            specification type
                                          */;
Integer* spec_i/* Integer – library specification type */;
int i;

add_List(create(type_Integer, 1));
spec_i = get_List(0);
if(~~get_List(0)!= NULL)~~ /* invalid use */
if(spec_i != NULL)
  i = *spec_i; /* i is equal to 1 */
```

Data types of specification references can only be casted to pointers to `void` or `Object` types (discussed below), as well as to the specification references of *compatible specification types*. The specification types are considered to be *compatible* if they are the subtypes of the same specification type.

Management of the memory, to which specification references refer, is automated through the mechanism of references computing with tracking of cycling references. When specification references are used in the assignment statements, operations of passing of references as the function arguments, returning of a reference from a function, exit of a reference from the visibility scope, reference counters are altered automatically. The memory that has been allocated for the value of a specification type, will then be automatically deallocated as the reference counter is zeroed to such value.

Usage of the pointers to specification references and aggregate types of C, which contain specification references, is not recommended for, in such cases, automatic altering of reference counters is not supported.

In SeC, the built-in specification type `Object` is defined is an incomplete specification type. It is basic type for all specification types, but values of the type can not exist. Reference type `Object*` is applied similarly to `void*`. A reference to any specification type may be casted to a reference to `Object` and vice versa. If, in case of an inverse cast, reference type of value is not compatible with the type to which the cast is performed, then the system behavior is not defined.

The functions that implement basic operations with the values of specification types are located in the library of specification types of CTesK (refer to "*Specification types library*" *section*).

Function of reference creation

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

As the first parameter, the function receives a pointer to *a specification type descriptor*. The descriptor constant of the specification type always has the identifier that is compiled of the name of a type with the `type_` prefix:

```
const Type type_name_of_specifiction_type;
```

The remaining parameters are those of type initializing. They differ for various types and are passed in a list of the `va_list*` type to the function of the specification type initialization (refer to "*Creating new specification types*" subsection).

The function allocates memory for the value of the specification type, fills it with zeros, invokes the type initialization function, passing to it the received values of the type initialization parameters in a list of the `va_list*` type, and returns the pointer to the allocated and initialized memory.

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

In the code above, references of the library specification types `Integer` and `String` are created and initialized, which are, respectively, the specification representation of the C built-in type `int` and of the string specification type (refer to "*Library of Specification Types*" section). When a reference of the `Integer*` type is created, an integer value should be passed to the function `create()`, which will be stored according to the reference. When a reference of the `String*` type is created, a normal string of C must be passed.

Functions copying values by references

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

The function copies the data stored by the reference `src`, to the location of the data by the reference `dst`. The references must be of nonzero value and belong to the same type, in other words, they must have similar type descriptors. If these conditions are not complied with, termination of the program will occur in the execution time, accompanied with an error message. The `copy()` function fills the memory pointed by the reference `dst` with zeros before it calls the function of coping specification type.

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

In the example above, the references `ref1` and `ref2`, after initialization, refer to different values of the `SpecificationType` specification type. As soon as the `copy()` function is invoked, the value of the reference `ref2` becomes equivalent to that of the reference `ref1`.

`Object* clone(Object* ref)`

The function allocates memory for a value of the type, to which `ref` refers, initializes the allocated memory with the value equivalent to that of the reference `ref`, and returns the pointer to the allocated and initialized memory.

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

Values of the references `ref1` and `ref2` become equivalent after invoking `clone()`.

Functions comparing values by references

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

When the values of the references passed are equivalent, the function returns a zero value. Otherwise, the function returns a non-zero value, which may be interpreted depending on the type of the values being compared. For instance, in respect to `String` library type, the result will be similar to that of the function `strcmp()` for `char*` type of C language. If the parameters are of incomparable types, i.e. the references' types are not equivalent, are not subtypes of the same type, and the type of one reference is not the subtype of the second reference's type (refer to "*Type invariants*" subsection), then the function returns a non-zero value. If one of the references is zero, and the other is not, a non-zero value will be returned. If both the references are zero, then zero will be returned.

```
/* creating two references of SpecificationType* type */
SpecificationType* ref1 = create(&type_SpecificationType);
SpecificationType* ref2 = create(&type_SpecificationType);
...
/* comparing values */
if (!compare(ref1,ref2)) {/* values are equivalent */
  ...
}
else {/* values are not equivalent */
  ...
}
```

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

The function returns either the `true` value, in the case the values of the references passed are equivalent, or `false` otherwise. When the parameters are of different types, the function returns `false`. If one of the references is zero, and the other is not, it returns `false`. If both the references are zero, `true` will be returned.

```
/* creating references of SpecificationType* type */
SpecificationType* ref1 = create(&type_SpecificationType);
SpecificationType* ref2 = create(&type_SpecificationType);
...
if (equals(ref1,ref2)) {/* values are equivalent */
  ...
}
else {/* values are not equivalent */
  ...
}
```

Function stringifying value by reference

`String* toString(Object* ref)`

The function returns the reference to the value of `String` type, i.e. the specification presentation of a string type.

```
/* creating a reference of SpecificationType* type */
SpecificationType* ref = create(&type_SpecificationType);
/* reference to a value of String type */
String* str;
...
/* stringifying *ref */
str = toString(ref);
/* print it out */
printf("*ref == %s\n", toCharArray_String(str);
...
```

In the code above, the library function `toCharArray_String()`is used, which returns the string content in a form of the array of `char` type, which ends with the zero value `'\0'`. The function returns a pointer to the internal data accessible via the passed reference of `String*` type. Therefore, on one hand, `free()`cannot be invoked for the returned pointer, and on the other hand, the pointer may not be used after destroying of the value of the reference passed.

Creating new specification types

Specification types are declared with the help of usual C `typedef` construct marked with the SeC keyword **`specification`**.

Declaration of a specification type

```
specification typedef basic_type new_type;
```

is different from its definition, which should contain an initializer:

```
specification typedef basic_type new_type = {
  .init = pointer_to_initialize_function
, .copy = pointer_to_copy_function
, .compare = pointer_to_compare_function
, .to_string = pointer_to_stringify_function
, .enumerate = pointer_to_enumerate_function
, .destroy = pointer_to_destroy_function
};
```

Prior to use of a specification type it must be declared or defined in each translation unit.

Definition of a specification type must occur only once, and only in one of the translation units that are integrated in a united system.

During *definition of a specification type*, *none of the following may be used* as the basic type:

- specification types, incompletely defined structures and arrays with unknown length;
- structures, unions, and arrays with fixed length, where elements of the above-defined types are contained;
- pointers to all of the above-listed types, other than specification references.

It is *acceptable* to use incompletely defined structures in *declarations of specification types*.

Within definition of a specification type, an initializer determines which functions will be applied to the basic operations with the data of the specification type.

Field `init` has the type `Init`:

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

According to the field, the function of specification type initializing is invoked from `create()` library function in the process of creating a reference (refer to "*Function of reference creation*" subsection). Within the first argument, it receives a pointer to the allocated area of memory which must be initialized. Within the second argument, a list of parameters is passed, based on which the data of specification type are initialized. The list of the parameters is built up of the parameters of `create()` function, following the first parameter, a type descriptor. Therefore, the parameters of `create()` function should, by the types and order, correspond to those expected in the specification type initializing function. The type descriptor (a global constant of `Type` type) is implicitly defined or declared in the process of defining or declaring the specification type, and has the name, which consists of the type name and the prefix `type_`: `type_`*`type_name`*.

Field `copy` has the type `Copy`:

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

According to the field, the function of copying the value of the given specification type is invoked from the library functions `copy()` and `clone()` (refer to "*Functions copying values by references*" subsection). The function of copying specification type is invoked if the references passed to the function `copy()` or `clone()` are non-zero. With the first parameter, it receives a reference, the value of which must be copied to the memory area according to the reference, passed to it within the second parameter.

Field `compare` has the type `Compare`:

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

According to this field, the function of comparing the values of the given specification type is invoked from the library functions `compare()` and `equals()` (refer to "*Functions comparing values by references*" subsection). The function of comparing specification type is called if the references passed to the functions `compare()` or `equals()`, are non-zero and the reference types are either similar, or *subtypes* of the same type, or if the type of one reference is *a subtype* of another reference (refer to "*Type invariant*" subsection). References as parameters are passed to the function in the same order as they have been passed to the function `compare()` or `equals()`.

Field `to_string` has the type `ToString`:

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

According to the field, the function of constructing a string presentation of the given specification type is invoked from the library function `toString()` (refer to "*Function stringifying value by reference*" subsection). The stringifying function is invoked if the reference transferred to `toString()` is non-zero.

Field `enumerate` has the type `Enumerate`:

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

According to the field, the function of enumerating references of specification types that are contained in the value of the given specification type, is invoked. The function is applied to resolution of specification references cycles in the process of automatic management of the dynamic memory.

Field `destroy` has the type `Destroy`:

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

According to the field, the function of deallocating resources is invoked, after zeroing of the counter of references to the specification type value.

If the basic type in definition of a specification type is allowable SeC type (refer to "SeC allowable types" subsection), then initialization of any field may be omitted. In such case, the *function by default* will be applied for a corresponding basic operation with the specification type value.

Function of initialization by default

The function of initialization by default for all the specification types defined on the basis of simple types (other than composite ones), has a single additional parameter of the basic type. It initializes the value of a specification type through deep copying of the parameter, with the consideration given to possible pointers and specification references cycles. The function of initializing structure specification types has additional parameters, the types and the order of which coincide with the types and the order of the fields of the basic structure. Fields by the passed reference are initialized through deep copying of the parameters passed. The function of initializing specification types defined on the basis of fixed-length array has one additional parameter, which acts as the pointer to the set of values of the type of the array elements in an amount that coincides with the length of the array. The array by the passed reference is initialized through deep copying of each value by the received pointer to the element of the array that corresponds to it. The technique of copying used in the initialization function, coincides with that used in the function of copying by default.

Function of copying by default

The function of copying by default provides deep copying, taking into account possible pointers and specification references cycles, which a reference being copied contains:

- values of allowable simple types of C, other than typed pointers, are copied byte by byte;
- specification references are copied with the use of the function of copying a corresponding specification type;
- typed pointer are interpreted to be the pointers to a single value, which does not depend on a location in the memory, and, therefore, a single value under the non-zero pointer is copied according to the rules enumerated in this list;
- values of composite types are copied by means of application of the above-listed rules to each of the elements that compose them.

Function of comparison by default

The function of comparing by default compares the basic type values in the following way:

- arithmetic types, functional pointers and non-typed pointers are compared byte by byte;
- typed pointers are considered as the pointers to a single value that does not depend on its location in the memory, in other words zero values are always considered to be equal, a non-zero value and a zero value are always unequal, and non-zero values are equal if and only if the values are equal, which they point at, and the values by the pointers are being compared under the rules of this list;
- specification references are compared with the assistance of `compare()` library function of comparing;
- composite types are compared with the application of the present rules to every one of their individual elements.

Function of stringifying by default

The stringifying function by default returns the string presentation of the basic type value:

- for arithmetic types—a numeric value;
- for untyped and functional pointers—address;

- for typed pointers—either NULL, or string presentation of a value by a non-zero pointer, marked with its address;
- for specification references—the result of invoking of toString() library function;
- for structural types—concatenation of string presentations of the structure fields, divided with commas, framed with curled braces, and with the word "**struct**" prior to it;
- for the fixed length array—concatenation of string presentations of array elements, divided with commas, and framed with square brackets.

Function of enumeration of inner specification references by default

The function of enumeration of inner specification references by default does not act if a basic type is a simple non-specification type. In the case when a basic type is a specification reference, the function through the passed functional pointer callback is called with a specification reference and auxiliary parameter par, which have been passed to the function of enumeration of inner specification references. If a basic type is composite, these rules apply to each of its component.

Function deallocating resources by default

The function of deallocating resources by default:

- does not act, provided a basic type is a arithmetic, functional, or untyped pointer;
- if a basic type represents a specification reference, the counter of references to its value is reduced by a unity;
- if a basic type represents a typed pointer, then the value by a non-zero pointer is processed according to the listed rules, following which the function free() is called for the pointer itself;
- if a basic type is composite, this rules apply to each component.

If in definition of a specification type the basic type represents an allowable SeC type (refer to "*SeC allowable types*" subsection), and the default functions of all basic operations implement the sufficient functionality, then an empty initializer is used in definition of the specification type.

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

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

In the example above, the specification type Point is created on the basis of the structure that contains two fields of the type int. In definition of the type, an empty initializer is used. That is why for implementation of the basic operations with the data of this type, the default functions are used. When creating a reference of the type Point*, the values used for initiation of the fields of the basic structure should be passed to create() function (refer to "*Function of initialization by default*" subsection). After creation of a reference of the type Point*, it may be handled similar to a basic structure pointer.

For the purpose of creation the structures of data with complex topology, for instance, as in case with definition of recursive structures, it is recommended to use only specification references.

```
struct link;

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

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

l1->next = l2;
```

In the above code fragment, the specification type `Link` is defined which implements a unidirectional list. When the field `next` of the reference `l1` is assigned a value of the reference `l2`, an automatic increase of the reference counter occurs by the value of reference `l2`. That is why after destroying of the reference `l2`, the value that it refers to, is not destroyed.

When a recursive structure that contains a non-specification pointer to itself is used in definition of the type `Link`, correct management of the dynamic memory will be more difficult.

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

In the above fragment of the code, after invocation of `free()` for pointer `s`, the filed `next` of the reference `l` will point at the deallocated memory. In order to avoid such problems, one should define special functions of initializing and destroying for the type `Link`.

If a default function of a certain basic operation does not implement functionality, that is necessary for a specification type being defined, then, for this operation, a special function must be defined, the pointer to which will initialize a corresponding field within the type definition initializer. Such necessity mostly appears, when a basic type in definition of a specification type represents a pointer to the first component of a dynamic array, or a union, or a pointer to one of such types, a structure, or a fixed length array, which contain elements of the listed types.

Function of initialization of specification type

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

The function does not have a return value. Within the first argument, the function receives a pointer of the type `void*` to the area of the memory, allocated for the purpose of storing the data of the specification type and *filled with zeros*, and initializes the area with the values passed within the second argument in the list of the type `va_list*`. When necessary, the additional memory is allocated within the function with aim to store the data.

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

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

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

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

In the example above, the specification type `IntegerSeq` is created, which is intended to store sequences of unknown length, containing references of the type `Integer*`, i.e. references to the values of the library specification type `Integer`, which is a specification representation of the built-in type of C `int` type. Type `IntegerSeq` is defined on the basis of the structure `struct integer_seq` with two fields: the sequence length (`length`) and the pointer to an array that contains the very sequence (`items`).

The library function `create()` allocates memory only for the purpose of storing values of the structure `struct integer_seq` itself. The function of initializing by default (refer to "*Function of initialization by default*" subsection) for such structure specification type has two parameters of initialization: the first one of the type `int` and the second of the type `Integer**`. In this case, the second parameter in the function of initialization by default is interpreted as a pointer to a single value. Therefore, the field `items` of the default function is initialized by the pointer to the reference, which contains a copy of the reference value, passed via the pointer. In our case, such functionality is unacceptable. That is why it is necessary to apply a special function of initializing `init_IntegerSeq`, which implements a sufficient functionality.

The function `init_IntegerSeq` expects the passed list of the type `va_list*` contains the single parameter of initiation of the type `IntegerSeq`, i.e. the sequence length. Its value initializes the field `length` by the initialized reference. The second field `items` is initialized by the pointer to the dynamically allocated and filled with zeros area of the memory sufficient to store the reference sequences of necessary lengths.

The pointer to initializing function `init_IntegerSeq` initializes the field `init` in definition of the type `IntegerSeq`.

Function deallocating resources of specification type

`void` ***`name_of_deallocating_function`*** **`(void* p)`**

The function without a return value has one parameter of the type `void*` that represents a pointer to the memory location, where data of the specification type are stored. The function must deallocate *only additional memory*, allocated within the function of initialization the given specification type.

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

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

In the example above, the function deallocating resources `destroy_IntegerSeq` for the specification type `IntegerSeq` is defined. Definition of the function is necessary for the function of deallocating resources by default (refer to "*Function deallocating resources by default*" subsection) for composite types interprets the field `items` as the pointer to the single value of the type `Integer*`, and therefore the reference counter decreases only for the first reference of the sequence, following which `free()` for the pointer `items` is invoked.

The function `destroy_IntegerSeq` reduces the reference counter by a unity for each element of the sequence by the passed reference, after which it deallocates memory by the pointer `items`. Reference counters are reduced by means of assigning the value `NULL` to the references.

The pointer to the function deallocating resources `destroy_IntegerSeq` initializes the field `destroy` in the definition of the type `IntegerSeq`.

Function of copying specification type

void *name_of_copying_function*(void* src, void* dst)

The function without a return value has two parameters of the type `void*`. The function must copy *into a sufficient depth* the data values by the pointer `src` passed within the first parameter to the memory area *filled with zeros* by the pointer `dst` passed within the second parameter.

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

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

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

In the example above, the function of copying vales of the type `IntegerSeq` is defined. Definition of the function is necessary, for the function of copying by default (refer to "*Function of copying by default*" subsection) interprets the field `items` as a pointer to the single value of the type `Integer*`, i.e. only the first value by the reference of the first element of the sequence `src` is copied. The function `copy_IntegerSeq` provides deep copying of the total sequence, through the library function of copying `clone()`(refer to "*Functions copying values by references*" subsection). Initialization by zeros of the memory allocated for `is_dst->items` is necessary to ensure during assigning within the cycle `is_dst->items[i] = clone(is_src->items[i])`, an attempt to reduce the reference counter by a nonexistent value of the reference `is_dst->items[i]` does not occur.

The pointer to the function of copying initializes the field `copy` in definition of the type `IntegerSeq`.

Function of comparing specification type

int *name_of_comparing_function*(void* left, void* right)

The function has a return value of the type `int` and two parameters of the type `void*`. The function compares the values by the passed pointers and returns zero, if the values are equivalent, or a non-zero value otherwise. A non-zero value may depend on relation of the values by the references passed.

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

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

In the example above, the function of comparing values of the type `IntegerSeq` is defined. Definition of the function is necessary, as the function of comparing by default (refer to "*Function of comparison by default*" subsection) interprets the field `items` as a pointer to the single value of the type `Integer*`, i.e. only the values by the references of the first elements of the sequences `left` and `right` are compared.

The function of comparing `compare_IntegerSeq` ensures comparison of sequences element by element. If the sequences are of different length, then the difference of lengths of sequences by the references `left` and `right` is returned. If the length of the sequences are equal, then the sequences are compared element by element with the use of the library function `compare()`. In this case, if all elements of the sequences coincide, the result is zero, otherwise the result of comparison of the first non-matching elements is returned.

The pointer to the function of comparing initializes the field `compare` in definition of the type `IntegerSeq`.

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

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

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

In the example above, the specification type `OneDimSimplex` is created, which represents a segment in plane. The segment is specified by the coordinates of two points in plane: two coordinates of the first point (`x1` and `y1`) an two coordinates of the second point (`x2` and `y2`). The segments are equal if their lengths match. The function of comparing by default implements element by element comparison of the components of the specification types. In the present case, the function of comparing by default will return zero, when a value of each field by the first reference coincides the value of a corresponding field by the second reference. Therefore, for the type `OneDimSimplex`, a special function of comparing `compare_OneDimSimplex` should be implemented, and the field `compare` within the type definition initializer should be initialized by a pointer to the function. The function `compare_OneDimSimplex` computes lengths of the segments based on the passed references and returns either `0`, if they are equal, or `1`, if the first segment is longer than the second one is, or `-1`, if the first segment is shorter than the second one is.

Function of stringifying specification type

String* ***name_of_stringifying_function*****(void* p)**

The function has a return value of the type `String*` and a parameter of the type `void*`. The function returns a reference to the specification type `String` (refer to "*Library of specification data types*" section of "*CTesK 2.2 Community Edition: SeC Language Reference*" document), by which a string presentation of the specification type should be contained, that corresponds to the value by the reference, passed within the single parameter of the function.

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

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

  String *res = start;

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

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

In the example above, the stringifying function `to_string_IntegerSeq` for the type is defined. Definition of the function is necessary, for the stringifying function by default (refer to "*Function of stringifying by default*" subsection) interprets the field `items` as a pointer to the single value of the type `Integer*` and creates a string which contains, within curly braces and divided by commas, the value of the field `length` and a string presentation of the value by the reference of the first element of the sequence.

The function `to_string_IntSeq` returns the reference of the type `String*`, which contains a string, where, within angle bracket ('<' and '>'), string presentations of the values by the references of all elements of the sequence are enlisted divided by commas, with preservation of their order. The function `to_string_IntSeq` utilizes functions `create_String()` and `concat_String()`, described in details in "*Library of specification data types*" section of "*CTesK 2.2 Community Edition: SeC Language Reference*" document.

The pointer to the function `to_string_IntSeq` initializes the field `to_string` in definition of the type `IntegerSeq`.

Function of enumeration of inner specification references of specification type

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

The function of enumerating references should invoke a callback function, the pointer to which is passed to it in the second argument, for each reference of the specification type. In all invocations of a callback function, the enumerated references are passed in the first argument, the second argument passes parameters via the pointer `par`, passed within the third argument of the function of enumerating.

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

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

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

In the example above, the function of enumerating the references `enumerate_IntegerSeq` for the type `IntegerSeq` is defined. Definition of a special function of enumerating the references is necessary, for the function of enumerating references by default (refer to "*Function of enumeration of inner specification references by default*" subsection) does not provide enumeration of the references, accessible via the pointer to array.

A callback function is invoked within the function `enumerate_IntegerSeq` for all references of the sequence, which are accessible via the field `items` of the type `Integer**`, by a pointer passed to the enumeration function `enumerate_IntegerSeq` within the second parameter. The first parameter of such invocations passes enumerated specification references, the second parameter passes the pointer passed in the third parameter to the function of enumerating `enumerate_IntegerSeq`.

Invariants

In a *specification*, the requirements for a system under test are contained, including the requirements for data. Such requirements consist in limitation of a range of allowable values and may be imposed both on a certain data type as a whole (by means of *type invariants*), and on the values of individual global variables (with the use of *variable invariants*).

Invariants may be also interpreted as common part of *preconditions* and *postconditions of specification functions*, which utilize the data of relevant types.

Invariants are automatically checked within the *specification functions* prior to the check of a *precondition*:

- for function parameters
- for expressions, described in the **`reads`** or **`updates`** access constraints

and prior to check of *a postcondition*:

- for function parameters
- for expressions, described in the **`writes`** or **`updates`** access constraints
- for return value

Additionally, the method of explicit check of an invariant is provided.

In checking of invariants of composite types, the check of invariants for all of its components is automatically executed.

Type invariant

A type invariant introduces the constraint for the range of values of a certain type. A resulting new type, the range of which is the subrange of a *basic type*, is called *a subtype*. The following constraint is imposed on a basic type: it must be an *allowable type*.

The type with invariant is defined with the use of the typedef construction, marked with the keyword **invariant**:

```
invariant typedef int Nat;
```

In this case, int represents *a basic type*, and Nat represents the *subtype* defined. Here, unlike a usual typedef construction of C, which only introduces a new identifier for the previous type, a new type with an own range is defined.

Constraints for the range of a subtype are defined in the **invariant** construction, similar to definition of functions with a single parameter of the defined *subtype*. The function returns a boolean value: `true`, if a value passed satisfies the constraints, or `false`, if it does not. As far as the type of a return value is fixed, it is not indicated explicitly:

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

If a *specification type* is used as a basic type, the parameter of the invariant function will have the type of a relevant *specification reference*, for values of *specification types* are only accessible via pointer:

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

At that, the functions of operating with a *subtype* will be assumed from definition of *the basic type*.

The invariant function shall not have side effects: apparent data shall not be altered, and the dynamic memory allocated within the function shall be deallocated in the same place.

It is allowable to define new specification types with indication of an invariant:

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

In such case, *a subtype* and a *basic type* coincide.

A *specification type* may not be defined on the basis of another *specification type*, but one may create *subtypes of specification types*. Moreover, one may define *subtypes for subtypes*, owing to which an hierarchy of types is created. In this case, for a value of *subtype* invariants of all parent *subtypes* upward the hierarchy will be also checked.

An invariant of a variable of a relevant type may be checked explicitly using the function invariant:

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

A subtype may be casted to a basic type; moreover, such transformation is executed implicitly. A basic type may also be casted to a subtype. However, due to the fact that values of a subtype are the subset of values of a basic type, such transformation should be written explicitly:

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

A situation may appear, when the invariant of the type turns unsatisfied. If necessary it may be checked explicitly after assignment.

Variable invariant

An invariant of a variable introduces constraints for a range of an individual global variable of an *allowable type*.

A variable with an invariant is defined with the help of usual declaration or definition, with **invariant** keyword:

```
invariant int Qty;
```

Constraints for the range of a variable are defined within the **invariant** construction, similar to definition of a function without parameters. The function returns a boolean value: `true`, of a value of a global variant satisfies the constraints, or `false`, if it does not. As far as the type of a return value is fixed, it is not indicated explicitly. The name of the variable, for which the invariant is defined, is indicated in brackets:

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

However, a variable is not a parameter of an invariant function. The function provides access immediately to the value of a global variable. At that, the function shall not have side effects: apparent data shall not be altered, and the dynamic memory allocated within the function shall be deallocated in the same place.

A variable invariant may be explicitly checked with the help of **invariant** function:

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

If the variable with an invariant has the type, for which *the type invariant* is defined, then *the type invariant* will be checked first, and then goes the variable invariant.

Specifications

A specification represents a formal description of requirements to a system under test. Interface functions of the system under test and its data, which represent its internal state, are defined. Within a specification, behavior of the interface functions is described by *specification functions*, while the state of the system is modeled by global *state variables*. Requirements to a tested system are formulated as constraints for behavior of interface functions (in the form of *precondi-*

tions and *postconditions* in *specification functions*) and for the data values (in the form of *type invariants* and *invariants of state variables*).

Binding of *specification functions* and model data to the functions and data of an implementation of system under test is performed through *mediators*.

Additionally, *coverage criteria* are derived from the specification (which are described in *specification functions*), which allow evaluating of testing completeness.

In the case of systems with deferred reactions, their behavior in response to outer actions consists of immediate and deferred reactions. The former are described with usual *specification functions*, while *deferred reactions* shall be added to the specification for the purpose of describing the latter. Binding of *deferred reactions* to a system under test is performed with the assistance of *mediators* and *reactions catcher*. Unlike *specification functions*, *coverage criteria* can not be specified for *deferred reactions*.

Specification functions

Specification functions describe behavior of interface functions of a system under test. In general, a specification function defines behavior of a system under a certain influence on it via a certain part of the interface.

Specification functions describe behavior in the form of data *access constraints*, *preconditions, coverage criteria* and *postconditions*.

Declaration of a specification function consists of the keyword **`specification`**, function signature (in the general sense of C) and, possibly, of *access constraints*.

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

Specification function body consists of three parts: *a precondition* (which may be omitted), *coverage criteria* (which may be either in any or no amount) and a *postcondition* (which is necessarily single).

A precondition checks applicability of a function to a given set of parameters values and state variables. *Coverage criteria* divide behavior of the system into *functionality branches*. Both *the precondition* and *coverage criteria* are executed at the pre-state, i.e. prior to interaction with the system under test. Expression values at the moment are called *pre-values*.

Prior to computing *a postcondition,* the interaction with the system under test is executed through invoking of a *mediator*. *The postcondition* checks compliance of the obtained results to the expected ones. It is executed at the post-state, i.e. after interaction, and deals with *post-values* of expressions.

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

Access constraints define the way of applying parameters and global variables in the specification function prior and after interaction with the system under test

In a general case, auxiliary code may be used within the body of a specification function between the described blocks. If within curly braces typed with bold face, no extra code is present, they may be omitted:

```
specification signature access_constraints {
  auxiliary_code_1_1
  pre { ... }
  {
    auxiliary_code_2_1
    coverage name_1 { ... }
    ...
    coverage name_n { ... }
    {
      auxiliary_code_3_1
      post { ... }
      auxiliary_code_3_2
    }
    auxiliary_code_2_2
  }
  auxiliary_code_1_2
}
```

Auxiliary code shall not have side effects: apparent data shall not be altered, and the dynamic memory allocated within a code block, shall be deallocated either within the same block, or in the block paired to it.

Specification functions are normally invoked in *scenario functions*. Invocation of a specification function consists in checking of *invariants* in compliance with *access constraints*, checking of the *precondition*, computing of covered branches in compliance with *coverage criteria*, executing of a testing interaction and synchronizing of model and implementation states in the *mediator*, secondary checking of *invariants* and checking of *a postcondition*. A specification function returns a value computed in the *mediator* (provided it is not declared as void).

Deferred reactions

Deferred reactions describe behavior of the system under test in the event of deferred reacting on outer influences. Deferred reactions describe behavior in the form of data *access constraints*, *preconditions* and *postconditions*. Unlike *specification functions*, *coverage criteria* are not used in the deferred reactions.

Declaration of a deferred reaction consists of the keyword **`reaction`**, function signature (in the common sense of C) and, possibly, of *access constraints*.

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

Deferred reaction:

- never has parameters,
- should return a specification reference.

Deferred reaction body consists of two parts: *a precondition* (which may be omitted) and a *postcondition* (which is necessarily single).

Precondition checks possibility of appearing of a reaction in a given state. *Precondition* is executed in pre-state and has the access to *pre-values* of expressions only.

Postcondition checks compliance of the result obtained when reaction emerges, to the expected one. It is executed in post-state after emerging of reaction and deals with *post-values* of expressions.

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

Access constraints define the way of applying global variables prior and after emerging of reaction.

In a general case, extra code may be used in the body of deferred reaction between described blocks. If, within the curly braces typed in bold face, there is no extra code, they may be omitted.

```
reaction signature access_constraints {
  auxiliary_code_1_1
  pre { ... }
  {
    auxiliary_code_2_1
    post { ... }
    auxiliary_code_2_2
  }
  auxiliary_code_1_2
}
```

Auxiliary code shall not have side effects: apparent data shall not be altered, and the dynamic memory allocated within a code block, shall be deallocated either within the same block, or in the block paired to it.

Deferred reaction can never be invoked explicitly, for it is initiated by the system under test.

Access constraints

Access constraints determine the way to applying of global variables and parameters in specification functions and deferred reactions, as well as of expressions where the former are used. Three types of access constraints are supported: reading (**`reads`**), writing (**`writes`**) and updating (**`updates`**).

Constraints are written after the signature of a specification function or deferred reaction in a form of a list of expressions, divided by commas, which is marked with one of keywords **`reads`**, **`writes`**, or **`updates`**:

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

Access constraint **`reads`** for a certain expression means that the value of the expression is not updated in the result of interaction, i.e. *the pre-value* of the expression coincides with *the post-value*.

Invariants for such expressions are automatically checked prior to checking of *a precondition*, and, prior to checking of a *postcondition* it is checked whether the expression value has been altered.

The access constraint **`writes`** for a certain expression means that *the pre-value* is not used in the specification function and may be not determined, while the *post-value* is generated in the result of interaction with system under test. Expressions with **`writes`** access may not be used in the operator of pre-value `@` (refer to "*Postcondition*" subsection).

Invariants for such expressions are automatically checked prior to checking of a *postcondition*.

The access constraint **`updates`** for a certain expression means that the pre-value of the expression is the input parameter, on which the behavior of the system may depend, while *the post-value* is generated in the result of interaction and may not coincide with the *pre-value*.

Invariants for such expressions are automatically checked prior to checking of both *the precondition* and the *postcondition*.

Expressions in the access constraints may be assigned with an identifier which is called an *alias*. Subsequently, the *alias* may be used for access to the expression value, including that the operator `@` may be applied to the *alias*:

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

Precondition

In interacting described by a specification function, behavior of the system under test may be not defined in certain situations. In order to single out such situations, precondition is used. During testing, precondition is checked every time when the specification function is invoked. Violation of a precondition represents that the test is made incorrectly.

In the case of deferred reaction, the precondition defines if appearance of such reaction in the given state is possible. During testing, precondition is checked every time when reaction appears. When precondition is violated, incompliance between the system under test behavior and its specification is registered.

Precondition is written in a form of instructions, included in curly braces and marked with the keyword **`pre`**. Such instructions represent the body of the function that has the parameters similar to those of either the specification function or deferred reaction, and returns the result of the type `bool`, which indicates whether the precondition is satisfied.

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

When the system behavior is defined for all values of input parameters and in any of model states (or appearance of reaction is acceptable in any state), precondition may be omitted.

Precondition is evaluated prior to interaction with the system under test. Expression values at the moment are called *pre-values*.

Prior to checking of precondition, *invariants* of parameters of specification function and expressions described in the *access constraints* both **`reads`** and **`updates`** are automatically checked.

Precondition must not have any side effects: apparent data shall not be altered, and the dynamic memory allocated within the precondition shall be deallocated in the same place.

Specification function precondition may be evaluated explicitly with the help of **`pre`** construction (as a rule it is used in *scenario functions* so that a specification function with incorrect parameter is not invoked):

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

Coverage criterion

A coverage criterion breaks down behavior of the system under test in *branches of functionality*. Coverage criteria are used for the evaluating testing completeness: a task is put in testing to pass each of the functionally branches at least once.

Coverage criterion is written in a form of instructions included in curly braces and marked up with the keyword **`coverage`** and an identifier. Such instructions represent the body of the function which has similar parameters to those of the specification function, and returns a special construction—a branch identifier and the string literal with the branch short description, which are included in curly braces:

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

For every set of pre-values of parameters and global variables, which satisfy the precondition, the coverage criterion shall return a construction, which identifies the branch of functionality. Otherwise, an error is registered in the process of test execution.

When not a single coverage criterion is defined in the specification function, it is considered that a pseudo criterion with a pseudo branch take place.

Similar to *precondition*, a coverage criterion is computed prior to interacting with the system under test and has access to *pre-values* of expressions. Coverage criterion shall not have any side effects, i.e. it must not alter apparent data and shall deallocate the dynamic memory, which has been allocated within it.

One of the coverage criteria may be declared as the default criterion, for which purpose the keyword **`default`** is used:

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

If none of the coverage criteria of the specification function is declared as the criterion by default, the last defined criterion becomes the same. A default criterion may be established also at the stage of execution with the use of the function `set_coverage_`*`name_of_specification_function`*`()`. Such function receives the string presentation of the criterion identifier and returns `true`, if the function performed successfully, or `false`, if a criterion with such name does not exist:

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

The quantity of functionality branches in the coverage criterion by default may be calculated with the use of the function `get_coverage_size_` *`name_of_specification_function`*`()`, which has no parameters:

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

The number of a functionality branch achieved under the certain parameters in the default coverage criterion may be computed with the use of **`coverage`** construction:

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

For the example of application of the above functions, refer to "*Iteration by coverage criterion*" subsection.

Whenever repeated computation of expressions that specify breakdown to functionality branches is necessary in postcondition or in coverage criteria, defined after the given coverage criterion, the construction **`coverage`**`(`*`branch_identifier`*`)` may be used. The value of the construction is the identifier of the covered functionality branch of the given coverage criterion. The construction may be used in if-else condition statements and in the switch statements of C language:

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

Postcondition

Postcondition of a specification function is used to describe constraints, which the results of the system under test performance should satisfy during interaction, described by the specification function. During testing the postcondition is checked every time after interaction is performed. If the postcondition is violated, inconsistency of the system under test behavior with its specification is registered.

Deferred reaction postcondition describes constraints, which the results of interaction should satisfy after emerging of the reaction. If the postcondition is violated, inconsistency of the system under test behavior with its specification is registered.

Postcondition is written down in a form of instructions in curly braces and marked with the **`post`** keyword. Such instructions represent the body of the function that has the same parameters as the specification function, and returns the result of `bool` type. The value `true` indicates that behavior of the system under test conforms to the expected one (postcondition is met), while `false` value indicates that the behavior differs from the expected one (postcondition is violated).

There must be always exactly one postcondition in the specification function and deferred reaction.

To access the value returned by *the mediator* of the specification function, identifier of the specification function (provided the specification function is not defined as void) is used. Similarly, to access the registered value of reaction, identifier of deferred reaction is used.

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

Postcondition is evaluated after interaction with the system under test. Actually, **`post`** keyword is interpreted as the test interaction. Expressions values after interaction are called *post-values*.

Prior to checking of postcondition, *invariants* of the parameters of the specification function and the expressions described in **`writes`** and **`updates`** *access constraints*, as well as *the invariant* of the return value are checked automatically.

To access *pre-values* from postcondition, the unary operator `@` is used. The expression under the operator must have the *allowable type* and must be *computable immediately before* **`post`** keyword (for the values of such expressions are automatically saved immediately before executing interaction with the system under test). It is prohibited to use the operator `@` for expressions that have **`writes`** access.

```
specification void f(List* l/*List is a library specification type*/) {
  int j;
  post {
    int i;
    Object* pre_item;
    for(i = 0, j = 0; i < @size_List(l)/* valid use */; i++, j++) {
      pre_item = @get_List(l, ~~i~~); /* invalid use: i is undefined
                                    out of post-block
                                 */
      pre_item = @get_List(l, ~~j~~); /* invalid use: j has the only
                                    unknown value out of post-block
                                 */
      pre_item = get_List(@l, j); /* valid use*/
      if(equals(get_List(l, i), pre_item))
        return false;
    }
    return true;
  }
}
```

If it is necessary to provide access to *pre-value* of an expression of a not *allowable* type, one should manually save the value of the expression in the local variable before **`post`** block (possibly, use of proper existing specification type or creating a new one is better solution):

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

Postcondition must not have any side effects: it must not update apparent data, and the dynamic memory allocated within the postcondition must be deallocated in the same place.

Mediators

Mediators are intended to bind *a specification* to the implementation of the system under test.

Mediators execute three tasks. First, they perform testing interaction, converting interaction parameters from model into implementation presentation. Second, they receive the result and convert it from implementation into model representation. Third, mediators synchronize the model state with that of the system under test implementation.

Mediators are implemented as *mediator functions* and *catchers*.

The table below shows which tasks are performed by which part of a mediator:

Testing	**Interaction execution**	**Result receiving**	**State synchronization**
without deferred reactions	call-block of mediator function	call-block of mediator function	state-block of mediator function
with deferred reactions	—	catcher	state-block of mediator function

Mediator function

Mediator function implements mediator by means of binding *the specification function* or *deferred reaction* with the implementation.

Declaration of a mediator function consists of **mediator for** keywords, splitted by identifier of the mediator function, and of a complete signature of a relevant specification function or deferred reaction, including the access constraints:

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

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

Mediator function body consists of two parts: *call-block* and *state-block.*

Within *call-block,* testing interaction is executed along with conversion of data from model into implementation presentation and back. Call-block is necessary for mediators of the *specification functions* and is not present in the mediators of *deferred reactions* (for interaction is initiated by the system under test in the case of deferred reactions).

State-block brings the model state in compliance with the state of the system under test implementation. State-block may be omitted in the mediators of *the specification functions* and is mandatory for the mediators of *deferred reactions*.

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

In a general case, extra auxiliary code may be used in the body of mediator function before and after the above-described blocks:

```
mediator identifier for specification_function_signature {
  auxiliary_code_1
  call { ... }
  state { ... }
  auxiliary_code_2
}
```

Auxilliary code may not have any side effects: apparent data must not be altered; the dynamic memory allocated in the code block, must be deallocated either in the same place, or in the block paired to it.

Mediator function shall be bind to the specification function (or deferred reaction) with the help of the function `set_mediator_`*`specification_function_name`*`()` (`set_mediator_`*`deferred_reaction_name`*`()`), which assumes the pointer to the mediator function. Normally, binding is executed in *the initialization function* of scenario:

```
set_mediator_push_spec(push_media);
```

If an error is revealed during mediator function execution (e.g. conversion of data from model into implementation presentation and back is impossible), one should register it with the assistance of the function `setBadVerdict()`, assuming the string with description of the situation encountered:

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

Call-block

Call-block is used only in the mediators of *specification functions*. The following operations are performed in it:

- the parameters of the specification function are converted into implementation presentation
- interface function (or several functions) of the system under test is invoked
- the result of the interface function and its output parameters are converted from implementation presentation into model one
- model presentation of the result is returned from the call-block

Call-block is written in the form of instructions in curly braces and marked with **`call`** keyword. Such instructions represent the body of the function, which has the parameters similar to those of the relevant *specification function*, and returns the result of the same type as the *specification function* has.

```
stack *stack_impl;
List*  stack_model;

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

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

Call-block is executed by the testing system when invoking relevant *specification function* in the point before **`post`** keyword.

State-block

State-block brings the model state in compliance with the state of the system under test implementation after executing of interaction or emerging of deferred reaction.

State-block is written in a form of instructions in curly braces and marked with **state** keyword. Such instructions represent the body of the function without a return value, which has the same parameters as the relevant *specification function* or *deferred reaction* has.

If the relevant *specification function* or *deferred reaction* has the return value type other than void, then the access to this value can be obtained through the identifier of such function (reaction).

When a system with open state is tested (i.e. when the testing system has the access to internal data of the implementation), the state-block must bring the model state in compliance with the implementation state:

```
stack *stack_impl;
List*  stack_model;

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

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

As far as building of the model state according to internal implementation state is similar for all *specification functions*, it is convenient to do it in a separate function, which is often called `mapStateUp()`.

When a system with hidden state is tested (i.e. when the testing system has no access to internal data of the implementation), the call-block should operate in the assumption that the implementation completes without errors in compliance with the specification, and then bring the model state to one that is expected in the result of interaction:

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

State-block of specification functions is executed by the testing system immediately after the *call-block* and prior to checking of *postcondition* in the specification function.

State-block of deferred reactions is executed after appearing of a *deferred reaction* prior to checking of *postcondition*.

Catcher

Catcher is intended to receive the result of *deferred reactions*.

Catchers are implementation-dependent components. Their purpose is to assemble all deferred reactions of the target system and register them in the *interaction registrar* (refer to "*CTesK 2.2 Community Edition: SeC Language Reference*").

Scenarios

A usual task of testing consists in checking of the system behavior when interacting with it through the set of interface functions until the specified level of coverage is achieved. The sequence of testing interactions for the purpose of solving such task is named a test.

Invocations of one or more *specification functions* and the method of enumerating of its parameters are specified in a *scenario functions*. In the process of testing, the testing system is in one of states that are called *scenario states*. Every invocation of a *scenario functions* transits the testing system from one state to another. All parameters are automatically enumerated in each achieved scenario state.

Test scenario consists of several *scenario functions* indicating a mechanism of building a test, the function of scenario state evaluation, the function defining the ways to initialization and finalization of test system and system under test.

A proper test is automatically generated on the basis of the data contained in a *testing scenario*.

Scenario function

Scenario functions describe sets of testing interactions. For this purpose, scenario function defines interactions (invocations of *specification functions*) and the way to enumerate their parameters. All interactions are automatically executed in every scenario state achieved in the process of testing.

Scenario functions are able to execute extra check for correctness of behavior of the functions invoked.

A scenario function is defined as a function without parameters, which returns a value of the type `bool` and is marked with **`scenario`** keyword:

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

In the plainest case, every scenario function corresponds to exactly one *specification function*. If a *specification function* has no arguments, then the scenario function body should contain a single invocation of a *specification function*. The scenario function must return `false`, if the system behavior is incorrect, and `true` in a normal situation. It must be noted, that a check of postconditions of invoked *specification functions* goes automatically and its results shall not be taken in account, i.e. check operation in a scenario function is of an extra nature.

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

Iteration statement

In the case when a specification function has arguments, a necessity emerges to enumerate them. To do so, *iteration statements* are used.

An iteration statement is used in a scenario function only and is intended for parameterized enumerating of testing interactions. From the syntax perspective, *an iteration statement* is similar to the `for` cycle:

```
iterate(decl; continuation_cond; increment; filtration_cond) body;
```

A declaration is a mandatory expression and represents a declaration of an iteration variable and its initialization (unlike C, where a variable is declared beyond a cycle). The iteration variable must be of an *allowable type* (refer to "*SeC allowable types*" section). The iteration variable is not a regular local variable: as far as iteration is executed independently in all scenario states, there exists a copy of iteration variable for each state, with its own value. When scenario function is invoked in a certain state, the value, which the iteration variable has received in the previous invocation within the same state, will be used as a value of such variable.

Continuation condition and *increment* perform in the same way as similar expressions of the `for` cycle do.

Filtration condition represents a logical expression. If it is `false`, then a transition to the next value of the iteration variable occurs. Filtration condition may be omitted, and in this case it is equivalent to the expression `true`, i.e. none of the iteration variable values will be rejected.

Therefore, the following construction:

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

to a certain sense is similar to the `for` cycle:

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

This model may be used in order to represent a sequence of invocations that will be generated within a single scenario state. However, a distinction should be recognized between an *iteration statement* and a cycle. First, a value of the iteration variable depends on a scenario state, while the cycle variable does not. Second, at a single scenario function invoking, only a single iteration is executed; it defines transition from one scenario state to another. In the case that a usual cycle is used within a scenario function, all the invocations enumerated by the cycle will be executed within the same transition.

Nested *iteration statements* are acceptable. However, successive *iteration statements* may not be used.

In the process of testing without deferred reactions, a scenario state alters together with the process of testing interaction that occurs at the moment of invoking of a *specification function*. Therefore, if in the result of the specification function performance global variables change, then already updated values will be accessible within a scenario function after it has been invoked. But the state variables as well as iteration variables will preserve their values that correspond to the previous scenario state, until the scenario function finishes.

In the process of testing with deferred reactions, a scenario state does not alter within the scenario function performance. Such alteration occurs in the end of performance of the scenario function, as soon as all deferred reactions have been caught.

Iteration by coverage criterion

Iteration by coverage criterion represents an important special case of iteration. The purpose of such iteration is in executing an interaction exactly once for every branch of functionality defined in the coverage criterion. Such iteration reduces the interactions number, without losing of the quality of coverage, and represents a way to fighting nondeterminism that emerges in the process of generalization of model state of the specification within a scenario.

Iteration by coverage criterion is executed with the help of **`coverage`** construction, which calculates the achieved *functionality branch* in the *coverage criterion* by default. Thus, during iteration by coverage elements (*functionality branches*) of coverage appointed in run-time with the help of the function `set_coverage_`*`specification_function_name`*, or, otherwise, explicitly (by the keyword **`default`**) or implicitly (the last coverage defined in the specification function) declared as the default coverage criterion (refer to *“Coverage criterion”* section), are iterated.

Suppose, there is a certain iteration of parameters:

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

In order to transform it in such a way so that no more than a single invocation of a *specification function* is executed for every *functionality branch* defined in the *criterion of coverage*, an extra iteration by functionality branches should be introduced with replacing of the existing iteration statements with usual cycles:

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

The function `get_coverage_size_`*`specification_function_name`*`()` returns the amount of the functionality branches in the coverage criterion by default. The construction **`coverage`** returns the number of *a branch* that is achieved under the specified parameters, while check of the number for equality to the iteration variable `cov` provides invoking of a *specification function* that corresponds to the successive *functionality branch*.

In more complicated cases, a necessity appears to execute several testing interactions within a single scenario function. Such situation may come normal in the process of testing of the systems with deferred reactions, when several *specification functions* are invoked in a scenario function. In this case, all necessary invocations shall just be written in the iteration block.

Scenario state variables

Along with iteration variables, there is another type of variables the value of which depends upon the scenario state—*scenario state variables*. For every scenario state, there is a copy of such a variable proper to it, with its individual value. If a scenario function is invoked in a certain scenario state, the value which has been received by the variable during the previous invocation in the same state will be used as the variable value.

Declaration of the scenario state variables starts with the modifier **`stable`** and must contain initialization. The scenario state variables should be of *the allowable type* (refer to *“ SeC allowable types”* section).

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

The location where such variables are declared in the scenario function, will affect only their visibility scope, not the functionality.

Function of evaluating scenario state

Within the process of performing, the testing system is in one of the states. The function of evaluating a scenario state shall normally compute such state on the basis of the values of variables which define the model state. A scenario state shall often represent a generalization of the model state, however, it also may coincide with the model state and on the other hand it may be totally not associated with it.

The function has no parameters and returns a reference to the value of *a specification type*:

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

An example of function that generalizes the model state, representing a list, as its length:

```
List* l;

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

Function of determining state stationarity

The function of determining state stationarity is only used in testing of the systems with deferred reactions.

The function has no parameters and returns a logical value that indicates whether the current specification model state is *stationary* or not. The specification model state is a *stationary state* if no deferred reactions are expected in this state).

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

Example:

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

Function of saving model state

The function of saving model state is only used in testing of the systems with deferred reactions.

The function has no parameters and returns a reference to the value of *a specification type* that contains the current specification model state.

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

In the case when the specification model state is represented as a variable of non-specification type or as several variables, it is necessary to define a new *specification type* that includes all required data.

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

Function of restoring model state

The function of restoring model state is only used in testing of the systems with deferred reactions.

The function has as its input a reference to the value of a *specification type*, which is the same as a type returned by *the function of saving model state*:

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

The value returned by *the function of saving model state* is input to the function of restoring model state. The purpose of the function is to restore the current specification model state of the system in compliance with this value.

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

Function of initialization

The function of initialization is intended to perform preliminary work prior to testing. The function assumes two parameters similar to those of the function *main*, and returns a logical value that demonstrates whether initialization has been successful:

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

As a rule, the function provides:

- initializing of the global variables of a model
- initializing of the system under test implementation
- setting up mediators with the help of the functions `set_mediator_`*`specification_function_name`*`()`

- if necessary, setting up coverage criterion by default for specification functions with help of the functions `set_coverage_`*`specification_function_name`*`()`

Whenever necessary, the function of initializing can use the initialization parameters passed to it.

```
char   *impl_data;
String* model_data;

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

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

In the case of testing of the systems with deferred reactions, the function is additionally used to:

- define the time of expecting of deferred reactions
- if necessary, allocate channels for processing of the immediate and deferred reactions

```
ChannelID chid;

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

Function of finalization

The function of finalization is intended to execute concluding operations after executing of test. The function has no parameters and no return value:

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

Normally, the function of finalization deallocates resources, allocated in the *function of initialization*.

```
char   *impl_data;
String* model_data;

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

Scenario variable

Test scenario provides all information necessary to automatically generate a test. It corresponds to a variable (*scenario variable*) of a special structural type with the name `dfsm` or `ndfsm`, marked with the modifier **`scenario`**:

```
scenario dfsm testScenario;
```

The name of the test engine which defines the technique of test generation is used as the name of the type:

- `dfsm` — Deterministic Finite State Machine;
- `ndfsm` — Nondeterministic Finite State Machine.

During test run `dfsm` applies the test actions that can change scenario state. `Dfsm` automatically keeps track of all state changes and constructs a finite state machine in accordance to test process. All reaches scenario states become the states of the machine, and transitions of the machine are marked by appropriate test actions. `dfsm` testing mechanism finishes the testing when it performed all test actions, defined by the user, in all states of the machine reachable from the starting state.

For this condition to be possible, the following constraints must be satisfied:

- **Finiteness**

  Number of states, reachable from the starting state by performing test actions from the defined set, must be finite.

- **Determinancy**

  Performing the same test action in any state of the system must lead the system to the same state.

- **Strong connectivity**

  Any scenario state is reachable from any other scenario state by performing test actions.

The `ndfsm` test engine works correctly with a wider class of finite state machines, in particular, with finite state machines having deterministic strongly connected complete spanning submachine:

- **Spanning submachine**

  A spanning submachine contains all reachable scenario states.

- **Complete submachine**

  For each scenario state and an allowable test action a complete submachine either contains all transitions from this state marked by this test action or does not contain such transions at all.

The `ndfsm` test enfine does not intended for testing systems with deffered reactions.

Definition of a scenario variable should contain the initializer of the following form:

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

In the `init` field, the name of the *function of initialization* is specified. The field may be omitted, but normally, a *function of initialization* is used at least for the purpose of setting up *mediators*.

In the `getState` field, the name of *the function of evaluating scenario state* is specified. If the field is omitted, testing is executed in a single state.

In the `actions` field, the list of the scenario functions, which are included in a given test, is specified. The list finishes with the value `NULL`. This field is obligatory.

In the `finish` field, the name of *the function of finalization* is specified. The field may be omitted.

A test scenario is invoked as a function with an identifier of the scenario variable with two parameters, which are the same as ones of the function `main()`, and without a return value. Usually, invocation is executed in the function `main()`:

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

As a test scenario is invoked, the parameters that have been passed to it are parsed. The test system processes the following standard parameters:

- **-tc** — send tracing to the console
- **-tt** — send tracing to the file with a unique name, composed of the scenario name and current time
- **-t *file_name*** — send tracing to the file with specified name
- **-nt** — disables tracing

Starting a test without any of the command line parameters described above has the same effect as starting with the **–tt** parameter.

- **--trace-accidental** — enables tracing of the accidental transitions
- **-uerr** — execute testing until the first error appears (by default)
- **-uerr=*number*** — execute testing until ***number*** errors apper (for `ndfsm` only)
- **-uend** — execute testing until complete, despite errors
- **--find-first-series-only**, **-ffso** — find only first success series

Standard parameters processed are deleted from the parameters list, and the updated list is passed to *the function of initialization* of scenario.

Several invocations of scenarios from the same program are acceptable. *Note* that, if the parameters passed to the executable test file from the command line, are simply sent to all invoked scenarios, then as soon as tracing is sent to the file, the trace of a successive scenario will overwrite the trace of the previous scenario. In order to provide that traces of all scenarios get in the same file, it is necessary to use functions `addTraceToFile()` and `removeTraceToFile()`[1]:

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

In the case of testing of the systems with deferred reactions, greater number of fields should be specified in the definition of the test scenario:

[1] For more details about these functions refer to the section "*Tracing services*" of the chapter "*CTesK test system support library*" of the document "*CTesK 2.2 Community Edition. SeC Language Reference*".

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

Fields `init`, `getState`, `actions` and `finish` have the same meaning as usually. There are three extra fields that are obligatory.

In the `isStationaryState` field, the name of *the function of determining state stationarity* is specified.

In the `saveModelState` field, the name of *the function of saving specification model state* is specified.

In the `restoreModelState` field, the name of *the function of restoring specification model* is specified.

Tests translation and building

Before running a test the SEC files with its source code should be translated into the C files. The files obtained can be compiled by an arbitrary C compiler, for example, gcc. To make an executable test it is necessary to link object files obtained with CTesK libraries.

This chapter describes how to translate SEC files into C files, as well as macrodefenitions available from SEC files and CTesK libraries.

SEC files translation

The `sec` command is used to translate SEC files into C files.

On Linux platform the command has the following format:

```
sec.sh sec_file_name c_file_name preprocessed_file_name
[preprocessor_options]
```

To translate the **account_scenario.sec** file into C code on Linux platform 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
```

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

Standard macrodefinitions

The following macrodefinitions can be used in test source code to control a translation:

- SEC file indicator — `SEC`

  Defines a file type in which code is located: SEC file, if `SEC` is defined, or C file, otherwise.

- CTesK version — `CTESK_VERSION`

  Defines the version of the CTesK translator used.

- CTesK build identifier — `CTESK_BUILD`

  Defines the build identifier of the CTesK translator used.

CTesK libraries

CTesK is delivered with the following set of libraries:

- **atl** — abstract types library;
- **tracer** — events tracing functions library;
- **ts** — SEC run-time library;
- **utils** — test system support library.

Analysis of test results and test debugging

During the testing process a *trace* is generated automatically, which collects all the detailed information about events happened during the test. In order to analyze results of the test, *test reports* are built for the test trace. *Static reports* contain information about the errors detected, coverage achieved, states reached and transitions between them. *Graphic reports* provide a detailed study of the testing process in its dynamics. Analysis of results shows to what degree the testing was complete, whether there were errors in implementation and whether further test revision is required.

In this chapter the account example is used, which is available in the **examples/account** folder of the CTesK.

Test trace

The test trace is generated in the process of the test performance and represents a file in the XML format. To analyze test results, it is much more convenient to use *test reports* than a trace itself.

Messages

Below are the main message types, which can appear in the trace:

1. Beginning and end of trace
2. Beginning and end of scenario

3. Scenario state reached
4. Beginning and end of transition between states
5. Scenario object value (state, scenario function name, iteration variable value etc.)
6. Beginning and end of specification function invocation
7. Model object value (function argument value etc.)
8. Beginning and end of serialization
9. Beginning and end of oracle work
10. Coverage structure
    a. Logical formulas (invariants and read-only access checks)
    b. Coverage criteria together with functionality branches
11. Value of logical formula
12. End of postcondition check
13. Functionality branch coverage
14. Error message
15. User message

The following table contains an example of the test trace:

<trace>	Trace beginning
<scenario_start trace="1" name="pqueue" time="1068567432000" host="CAMEL" os="Linux"/>	Scenario beginning
<state trace="1" id="0"/>	Scenario state
<scenario_value trace="1" kind="state" type="" name=""><![CDATA[struct { 0, 0 }]]></scenario_value>	Scenario object value
<transition_start trace="1" id="0"/>	Transition beginning
<scenario_value trace="1" kind="scenario method" type="" name=""> <![CDATA[enq_scen]]> </scenario_value>	Scenario object value
<scenario_value trace="1" kind="iteration variable" type="int" name="i"> <![CDATA[0]]> </scenario_value>	Scenario object value
<model_operation_start trace="1" signature="void enq_spec(Item item)"/>	Specification function beginning
<model_value trace="1" kind="argument" type="Item" name="item"> <![CDATA[00303478]]> </model_value>	Model object value
<oracle_start trace="1" signature="void enq_spec(Item item)"/>	Oracle beginning
<coverage_structure trace="1">	Coverage structure beginning
<formulae> <formula id="0"><![CDATA[invariant type Item (@item)]]></formula> <formula id="1"><![CDATA[invariant type Item (item)]]></formula> <formula id="2"><![CDATA[reads item]]></formula> </formulae>	Logical furmulas
<coverage id="c1"> <element id="0" name="single branch "/> </coverage>	Coverage criteria and functionality branches

</coverage_structure>	Coverage structure end
<user_info trace="1"><![CDATA[Oracle call]]></user_info>	User message
<prime_formula trace="1" id="0" value="true"/>	Formula value
<precondition_end trace="1"/>	Precondiition check end
<coverage_element trace="1" coverage="c1" id="0"/>	Functionality branch coverage
<prime_formula trace="1" id="1" value="false"/>	Formula value
<exception trace="1" internal="false"> <where></where> <info><![CDATA[Postcondition failed]]></info> </exception>	Error message
<oracle_end trace="1"/>	Oracle end
<model_operation_end trace="1"/>	Specification function end
<transition_end trace="1"/>	Transition end
<state trace="1" id="0"/>	Scenario state
<scenario_value trace="1" kind="state" type="" name=""><![CDATA[struct { 0, 0 }]]></scenario_value>	Scenario object value
<scenario_end trace="1"/>	Scenario end
</trace>	Trace end

Tracing control

When running a test, one can use the following command line parameters, which control the tracing:

- **-tc** — sends the trace to console
- **-tt** — sends the trace to a file with a unique name, which consists of a scenario name and a current time
- **-t *file_name*** — sends the trace to a file with a specified name
- **-nt** — disables tracing

Starting a test without any of the command line parameters described above has the same effect as starting with the –**tt** parameter.

- **--trace-accidental** — enables tracing of the accidental transitions

In the program, tracing to a console is assigned by the `addTraceToConsole()` and `removeTraceToConsole()` functions, and tracing to a file is assigned by the functions `addTraceToFile()` and `removeTraceToFile()` [2]:

[2] For more details about these functions refer to the section "*Tracing services*" of the chapter "*CTesK test system support library*" of the document "*CTesK 2.2 Community Edition. SeC Language Reference*".

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

By default, *accidental transitions* (i.e. transitions without a call of the oracle) do not get into a trace. One can change this using the `setTraceAccidental()` function[3]:

```
int main(int argc, char **argv) {
  setTraceAccidental(true); // trace all transitions
  testScenario(argc,argv);
  return 0;
}
```

During testing one can write arbitrary user messages into a trace using the `traceUserInfo()` function[4]:

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

One can also write formatted user data into a trace using the `traceFormattedUserInfo()`[5] function. This function supports all the conversion specifiers supported by the standard function `printf()` and the special specifier `$(obj)` to convert a specification object into a string. All the `$(obj)` specifiers shall precede the `printf()` specifiers:

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

By default, character encoding of the trace is UTF-8. One can change this using the `setTraceEncoding()`[6]:

```
int main(int argc, char **argv) {
  // set trace character encoding to ISO-8859-1
  setTraceEncoding("ISO-8859-1");
  testScenario(argc,argv);
  return 0;
}
```

[3] For more details refer to the section "*Tracing services*" of the chapter "*CTesK test system support library*" of the document "*CTesK 2.2 Community Edition: SeC Language Reference*".

[4] Refer to the same place.

[5] Refer to the same place.

[6] Refer to the same place..

Static reports

A static report is generated for one or several traces and contains information about results of the testing, about the coverage of the functionality branches of specification functions and about states reached and transitions between them. The test trace contains all information about the test progress, and a static report represents this information in a representative and compact form.

A report is a set of HTML files linked together by references. Logically it consists of three sections:

- Completed test scenarios
- Specification functions coverage
- Errors detected

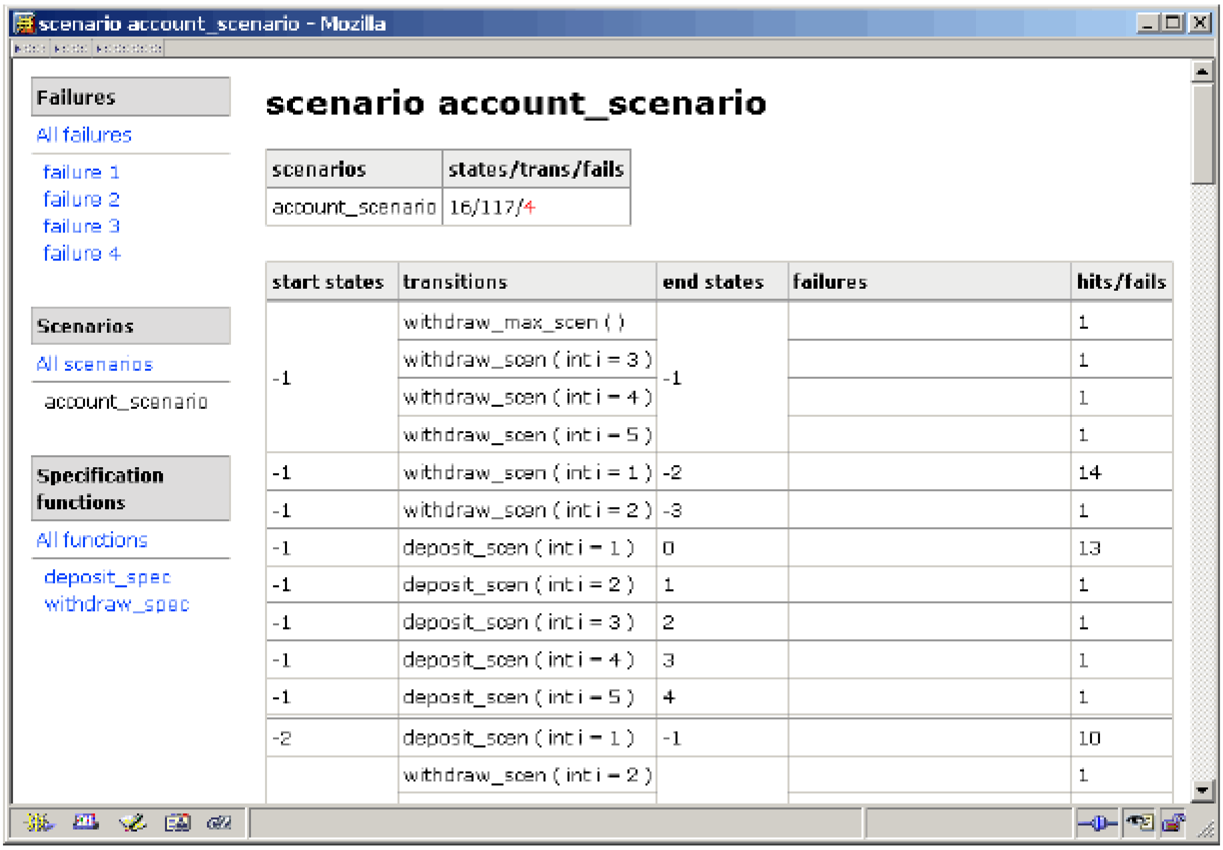
Each section contains general information and subsections with detailed information about specific errors, scenarios, or functions.

Scenarios

A general page of the scenario report contains information about the number of states, transitions, and errors for each scenario.

Detailed information about scenario includes:

- Number of states, transitions, and errors
- List of errors (if any)
- List of transitions between states with information about number of falls into each transition



scenario account_scenario

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

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

Figure 1. Test scenario report page.

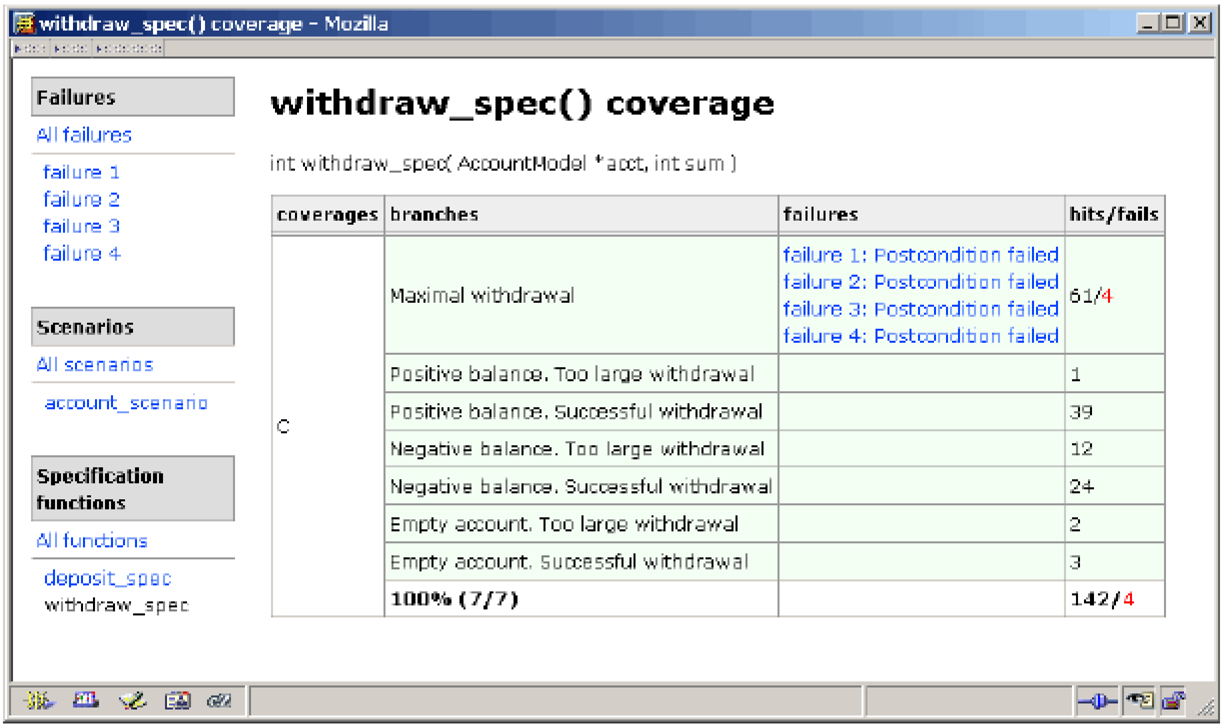
Specification functions

The general page of the report about specification function coverage contains a table, which represents the following information for each coverage of each tested specification function:

- Specification function name
- Name of the coverage
- Functionality branches coverage percentage (number of branches covered and total number of branches is specified in the brackets)
- Number of calls of this specification function and number of errors detected during these calls

Detailed information about a specification function includes:

- Full signature of the specification function
- Table, which specifies the number of falls into a branch and the number of errors detected in it for each functionality branch of each coverage. If there are no falls, the corresponding line is highlighted in red, and in green otherwise
- List of errors related to this function (if any)



Failures

All failures

failure 1
failure 2
failure 3
failure 4

Scenarios

All scenarios

account_scenario

Specification functions

All functions

deposit_spec
withdraw_spec

withdraw_spec() coverage

int withdraw_spec(AccountModel *acct, int sum)

coverages	branches	failures	hits/fails
C	Maximal withdrawal	failure 1: Postcondition failed failure 2: Postcondition failed failure 3: Postcondition failed failure 4: Postcondition failed	61/4
	Positive balance. Too large withdrawal		1
	Positive balance. Successful withdrawal		39
	Negative balance. Too large withdrawal		12
	Negative balance. Successful withdrawal		24
	Empty account. Too large withdrawal		2
	Empty account. Successful withdrawal		3
	100% (7/7)		**142/4**

Figure 2. Specification function coverage report page.

Errors

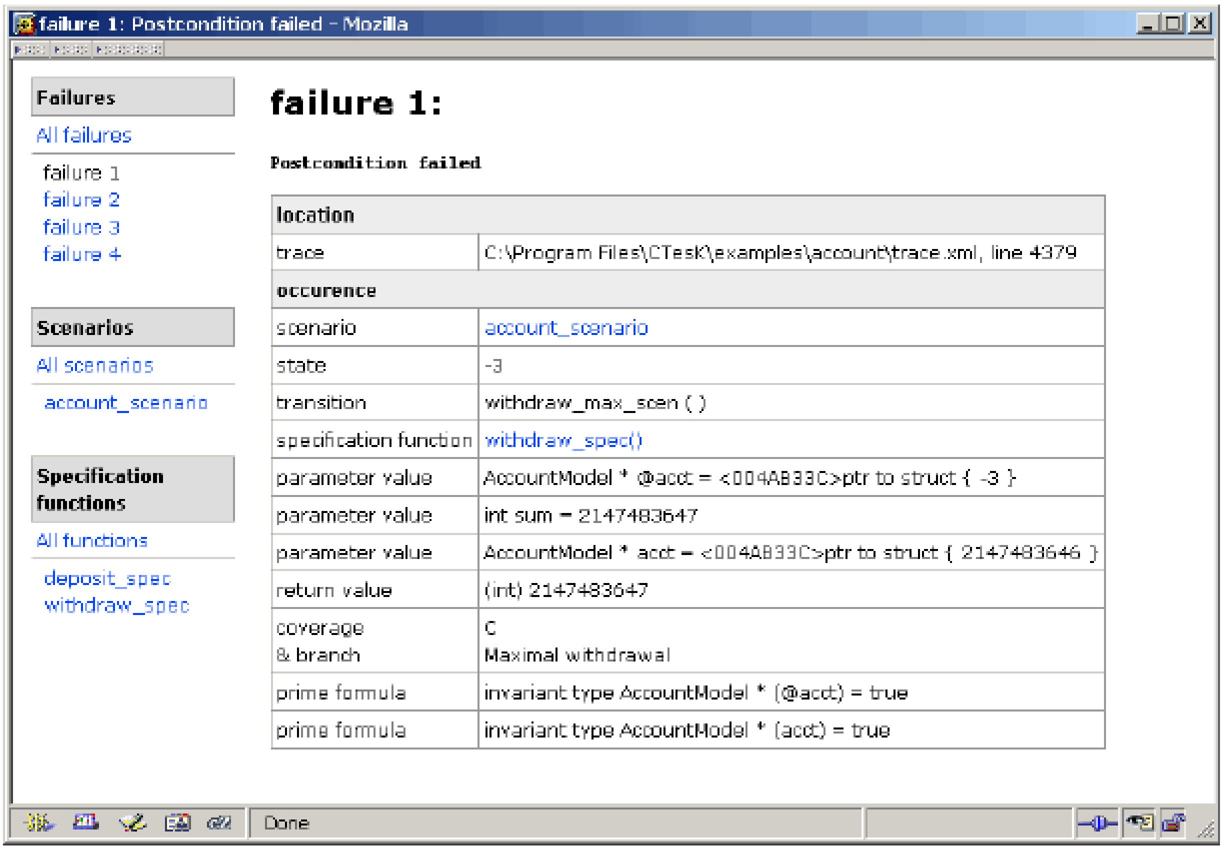
The general page of the report about errors contains a list of all errors detected (if the error is detected in the specification function oracle, the name of this function is specified).

Detailed information about an error includes at least:

- Trace filename and the line number in it
- Scenario name

Depending on the context in which the error was detected, there may be other elaborating information as well:

- State
- Transition (name of the scenario function and values of iteration variables)
- Specification function name and its parameters
- Information about falling into a functionality branch
- Values of logic formulas



failure 1:

Postcondition failed

location	
trace	C:\Program Files\CTesK\examples\account\trace.xml, line 4379
occurence	
scenario	account_scenario
state	-3
transition	withdraw_max_scen ( )
specification function	withdraw_spec()
parameter value	AccountModel * @acct = <004AB33C>ptr to struct { -3 }
parameter value	int sum = 2147483647
parameter value	AccountModel * acct = <004AB33C>ptr to struct { 2147483646 }
return value	(int) 2147483647
coverage & branch	C Maximal withdrawal
prime formula	invariant type AccountModel * (@acct) = true
prime formula	invariant type AccountModel * (acct) = true

Figure 3. Failure report page.

Analysis of results

After the test trace has been obtained, it is necessary to perform analysis of test results in order to find out, how successfully and completely the specification requirement conformance has been checked, and whether test revision is required.

Two tasks are performed during analysis:

1. Error search
2. Determination of coverage completeness

Error search

Assume that some error was detected during the test. We need to determine the reason for this error and localize it.

The following error types may be detected during testing:

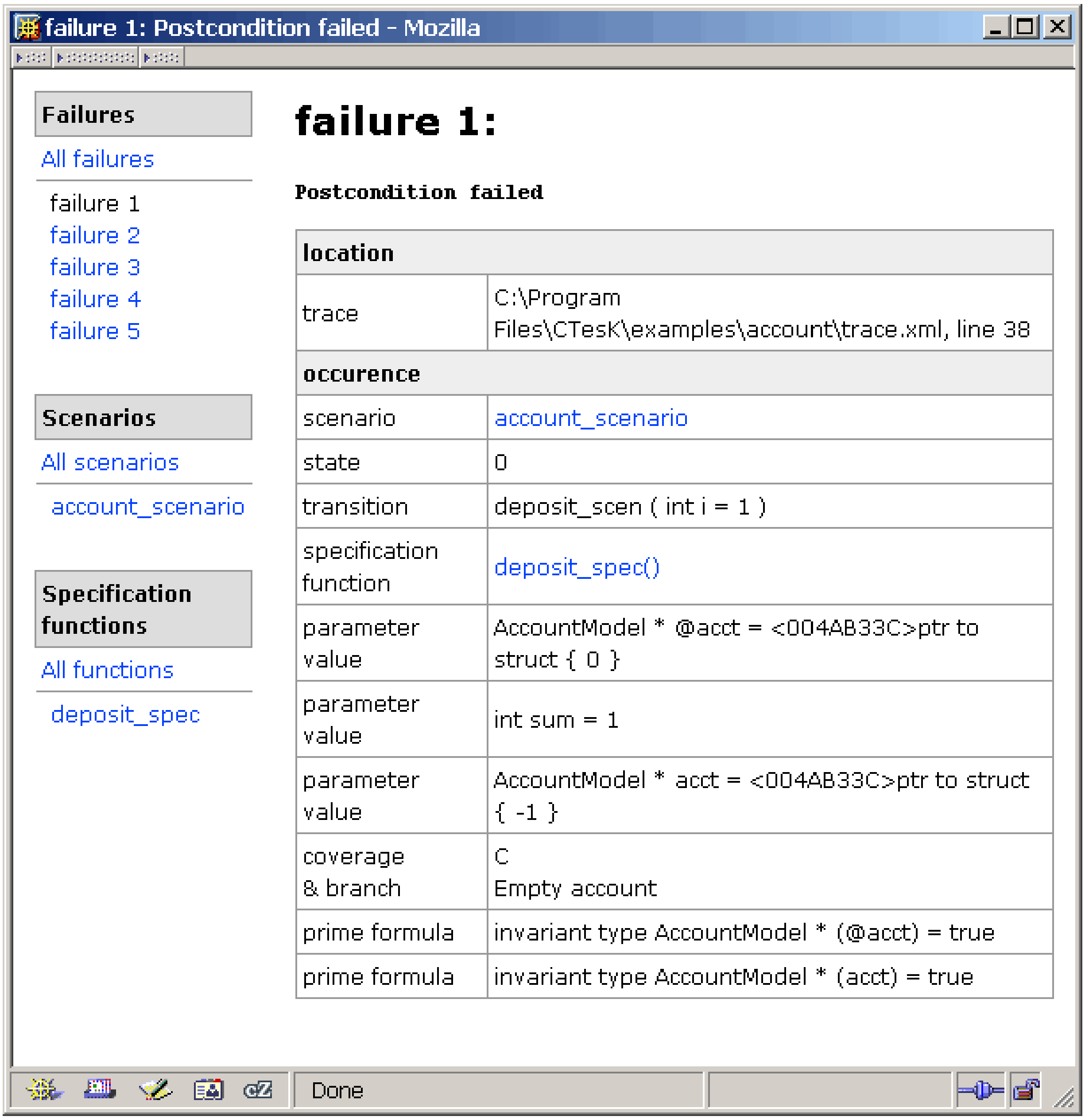
- Postcondition violation
- Violation of an invariant or access constraint
- State graph non-determination
- Violation of the strong connectivity of the graph of states
- Error during initialization
- Internal and user errors

Each type is described below in detail.

Postcondition violation

The violation of a postcondition indicates inconsistency between a specification and an implementation of the system under test.

Assume we got a report during testing, which is shown in the illustration. Let us analyze it.



Failures

All failures

failure 1
failure 2
failure 3
failure 4
failure 5

Scenarios

All scenarios

account_scenario

Specification functions

All functions

deposit_spec

failure 1:

Postcondition failed

location	
trace	C:\Program Files\CTesK\examples\account\trace.xml, line 38
occurence	
scenario	account_scenario
state	0
transition	deposit_scen (int i = 1)
specification function	deposit_spec()
parameter value	AccountModel * @acct = <004AB33C>ptr to struct { 0 }
parameter value	int sum = 1
parameter value	AccountModel * acct = <004AB33C>ptr to struct { -1 }
coverage & branch	C Empty account
prime formula	invariant type AccountModel * (@acct) = true
prime formula	invariant type AccountModel * (acct) = true

Figure 4. Postcondition failure detail report.

We can see that there is a postcondition violation ("**Postcondition failed**"), which happened in the following situation:

- The **account_scenario** scenario
- The **deposit_scen** scenario function is called in the state **0** with the parameter **i** = **1**
- The oracle of the **deposit_spec** specification function is called with the parameter **sum** = **1**, the prevalue of the parameter **acct** is a reference to the structure with a zero field, the post-value of the parameter **acct** is a reference to the structure with a field equal to **-1**
- The error was detected in the "**Empty account**" branch of the "**C**" coverage criterion
- The **sum** parameter invariant and the invariants of the prevalue and postvalue of the parameter **acct** are met

Let us consider the `deposit_spec()` specification function code:

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

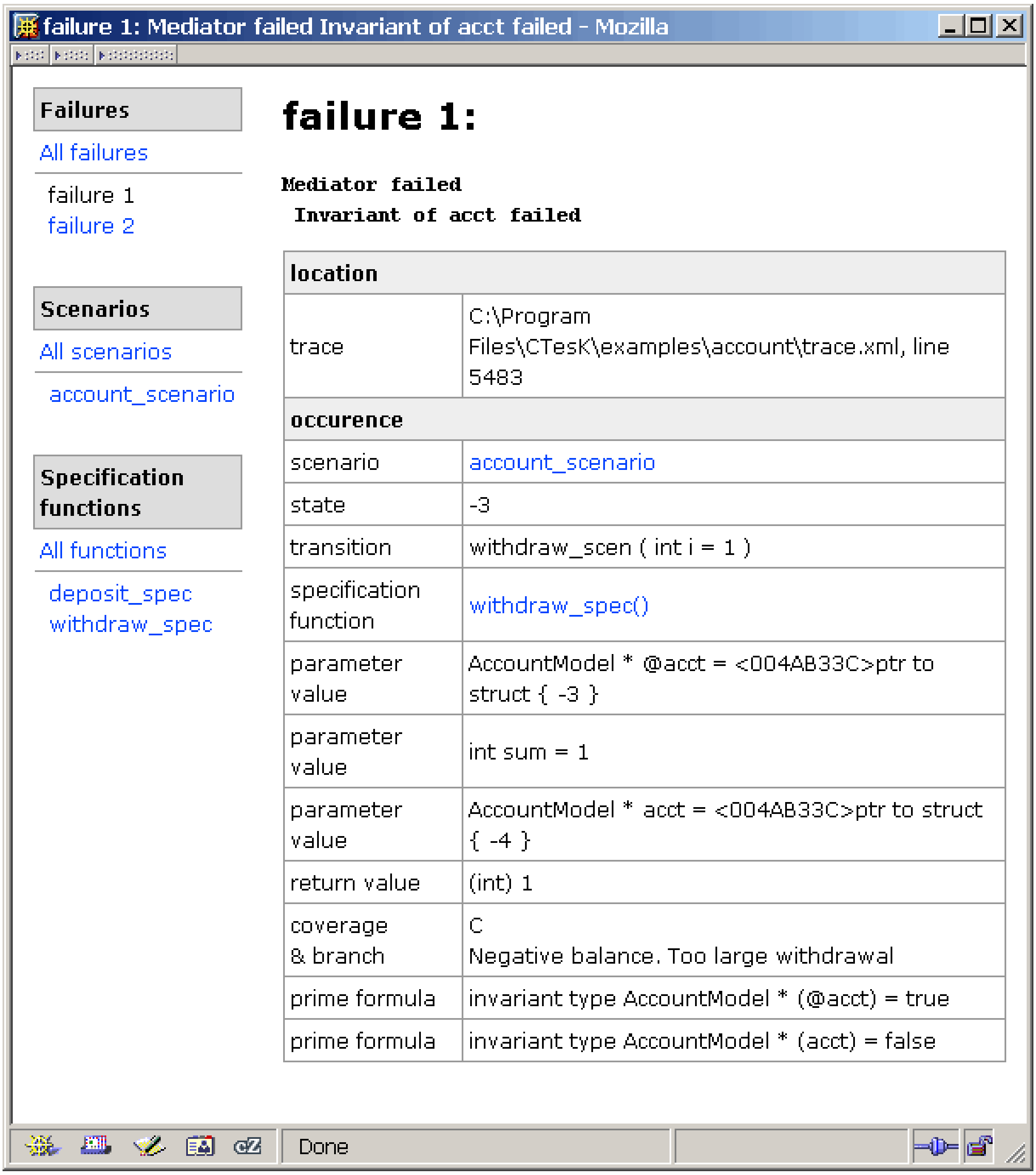
On the basis of it we can see that the `balance == @balance + sum` condition has been violated. Indeed, the expression `-1 == 0 + 1` is incorrect. One should look for the error in the implementation function `deposit()`:

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

Indeed, the implementation function withdraws this sum from the account instead of depositing it into.

Violation of the invariant or access constraint

Violation of the invariant or access constraint in the report looks as follows:



failure 1: Mediator failed Invariant of acct failed - Mozilla

Failures

All failures

failure 1
failure 2

Scenarios

All scenarios

account_scenario

Specification functions

All functions

deposit_spec
withdraw_spec

failure 1:

```
Mediator failed
 Invariant of acct failed
```

location	
trace	C:\Program Files\CTesK\examples\account\trace.xml, line 5483
occurence	
scenario	account_scenario
state	-3
transition	withdraw_scen (int i = 1)
specification function	withdraw_spec()
parameter value	AccountModel * @acct = <004AB33C>ptr to struct { -3 }
parameter value	int sum = 1
parameter value	AccountModel * acct = <004AB33C>ptr to struct { -4 }
return value	(int) 1
coverage & branch	C Negative balance. Too large withdrawal
prime formula	invariant type AccountModel * (@acct) = true
prime formula	invariant type AccountModel * (acct) = false

Done

Figure 5. Invariant failure in postcondition failure detail report.

Let us analyze the report. We can see that the postcondition has been violated in the following situation:

- The **account_scenario** scenario
- The scenario function **withdraw_scen** is called in the state **-3** with the **i = 1** parameter
- The oracle of the specification function **withdraw_spec**, the parameter **sum = 1**, the prevalue of the parameter **acct** is a reference to the structure with a field equal to **-3**, the postvalue of the parameter **acct** is a reference to the structure with a field equal to **–4**, and the returned value equal to **1**
- The error has been detected in the "**Negative balance. Too large withdrawal**" branch of the coverage "**C**".

- The type invariant for the prevalue of the parameter **acct** has been met
- The type invariant for the postvalue of the parameter **acct** has been violated

The following invariant for the `AccountModel` invariant type has been defined:

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

Hence, the `acct.balance >= -MaximalCredit` condition has been violated. We can conclude that during the withdraw function there has been an attempt to withdraw such a sum from the account, that the credit exceeded the maximum value, but the implementation under test allowed the withdrawal nevertheless. Let us turn to the withdraw function implementation:

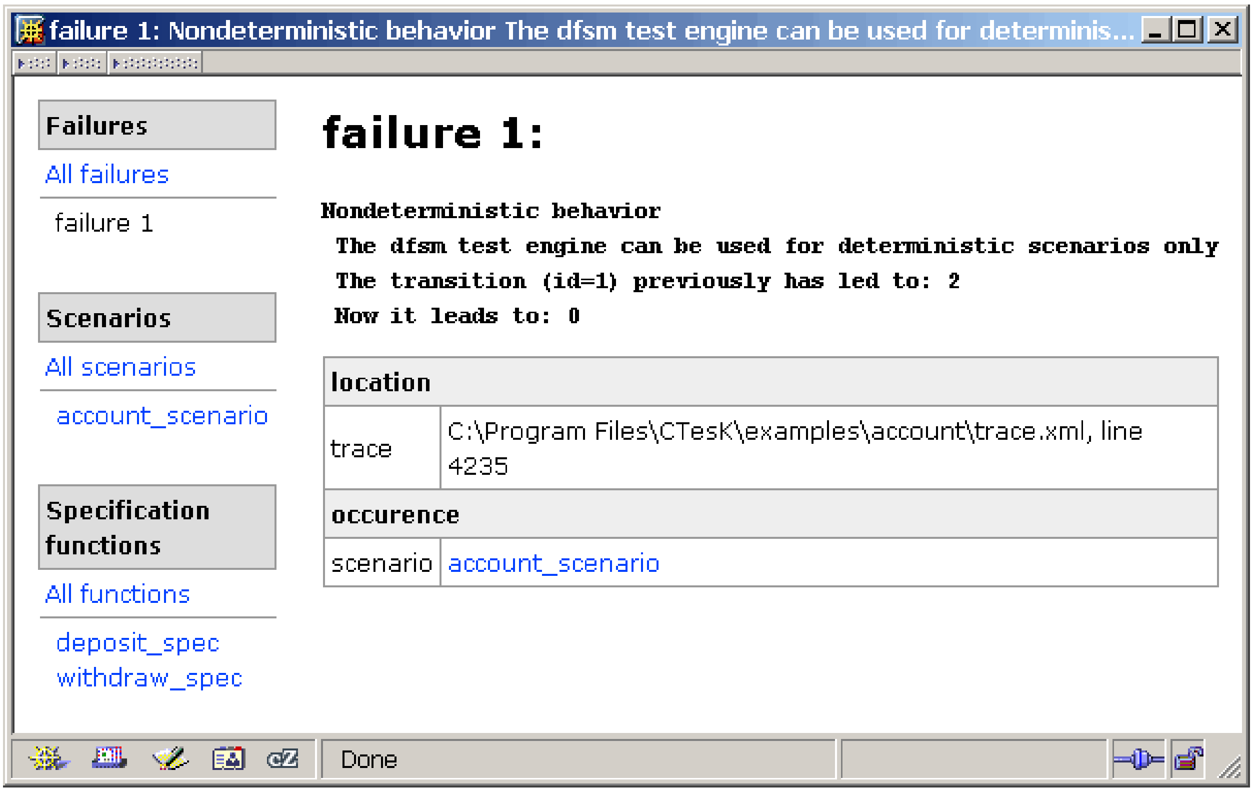
```
int withdraw (Account *acct, int sum) {
  //if (acct->balance - sum < -MAXIMUM CREDIT)
  //  return 0;
  acct->balance -= sum;
  return sum;
}
```

Indeed, the code, which prohibits withdrawal of an amount exceeding the credit, is commented.

State graph non-determination

Non-deterministic emerges when in the same generalized state the same scenario function, called with the same iteration variables' values, transits the system into different generalized states in different cases.

Non-deterministic failure looks in the report as follows:



failure 1:

Nondeterministic behavior

The dfsm test engine can be used for deterministic scenarios only

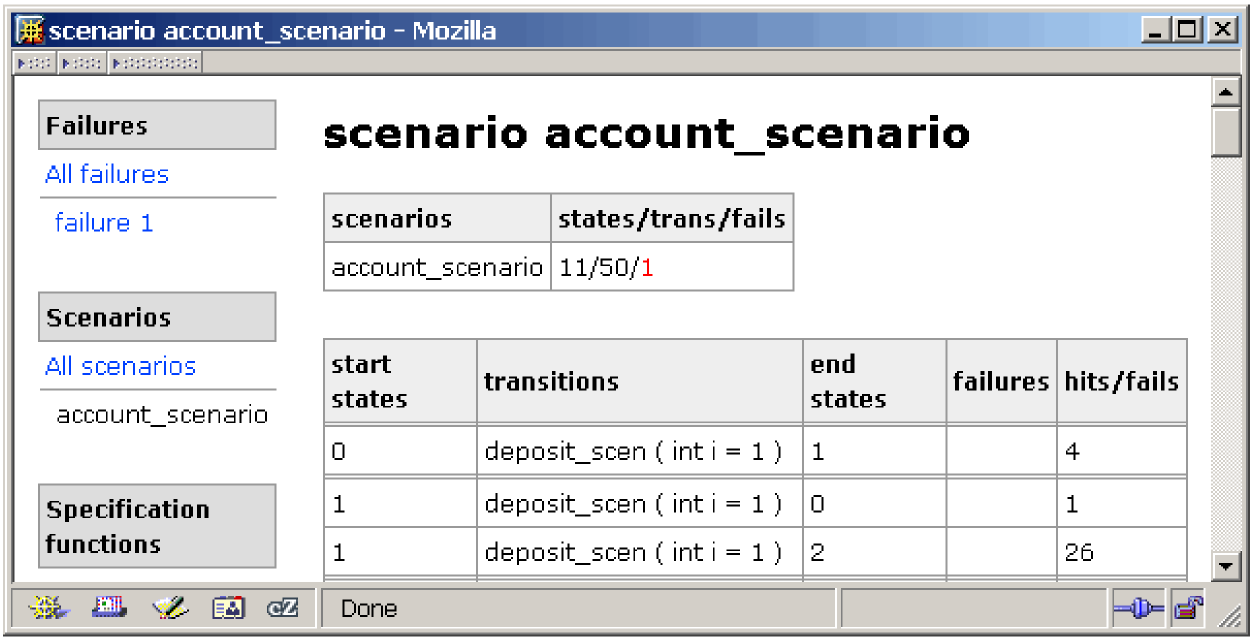
The transition (id=1) previously has led to: 2

Now it leads to: 0

location	
trace	C:\Program Files\CTesK\examples\account\trace.xml, line 4235
occurence	
scenario	account_scenario

Figure 6. "Non-deterministic behavior" failure report

In order to understand which particular function has transited the system into different states, one should refer to the report about scenario, which contains a table with all the transitions:



scenario account_scenario

scenarios	states/trans/fails
account_scenario	11/50/1

start states	transitions	end states	failures	hits/fails
0	deposit_scen (int i = 1)	1		4
1	deposit_scen (int i = 1)	0		1
1	deposit_scen (int i = 1)	2		26

Figure 7. Scenario detail report with "non-deterministic behavior" failure.

As we can see, the scenario function **deposit_scen** has been called from the state **1** with **i = 1** parameter 27 times altogether, and **26** calls sent the system to the state **2** and one call has sent it to the state **0**.

The reason for such behavior can be both real non-determination of the behavior of the function and problems with the state generalization. In our case the reason is exactly these problems, which is determined after consideration of the code of the function, evaluating the model state:

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

Here the absolute value of the balance was chosen as a generalized state. In such situation, the generalized scenario state `1` corresponds to the specification model states the `1` and `-1`, but the behavior of the function in those specification states will be different, leading to a failure.

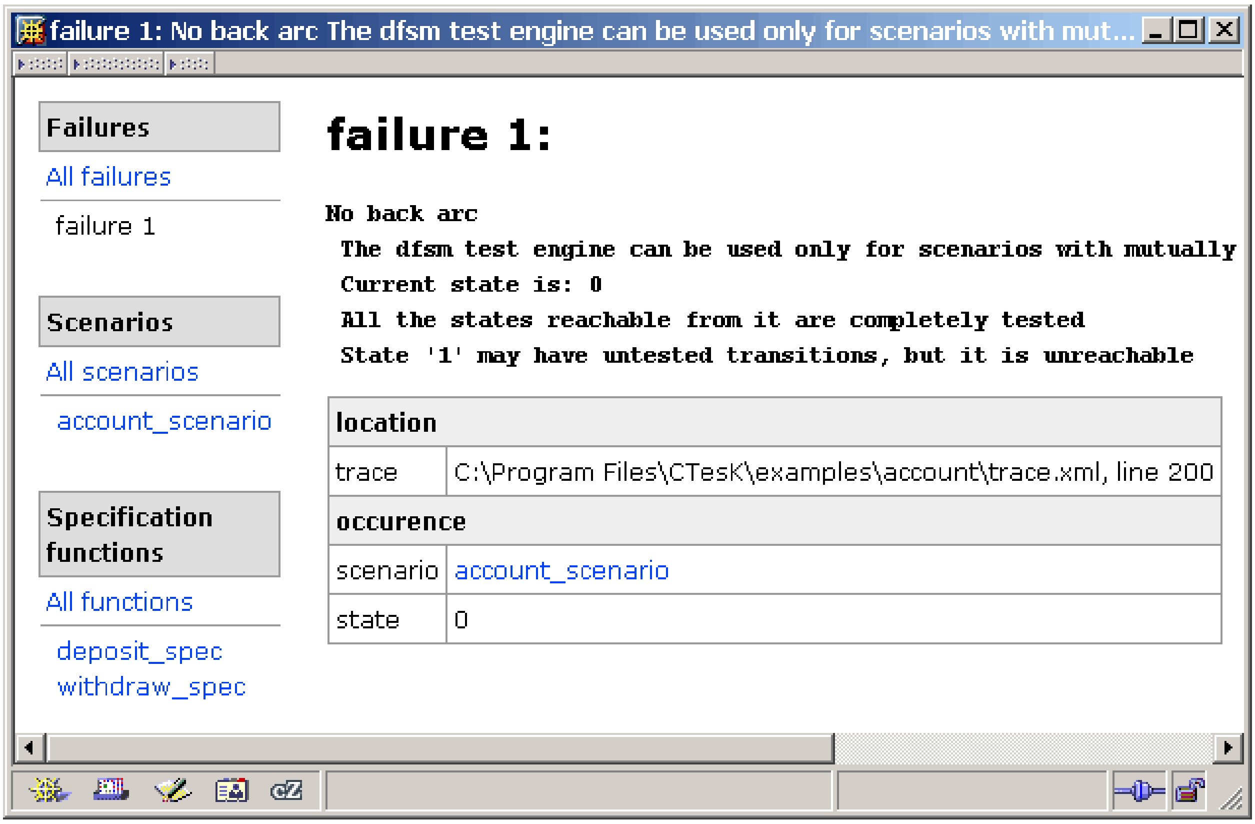
Another situation of non-determinism is related to an incorrect generalization of state. It is possible, when in some generalized state some functionality branch is achieved one time, and not achieved next time. The "**Can't find suitable parameters**" failure will be returned in this case.

It means that some functionality branch has been achieved earlier in this generalized state. However none of the parameter values, enumerated by iteration operators at this moment, lead to the achievement of this branch.

Violation of strong connectivity of the graph of states

Strong connectivity violation takes place when for some transition from one generalized state to another it is impossible to return to the initial state via any sequence of transitions.

The report in this case looks as follows:



failure 1: No back arc The dfsm test engine can be used only for scenarios with mut...

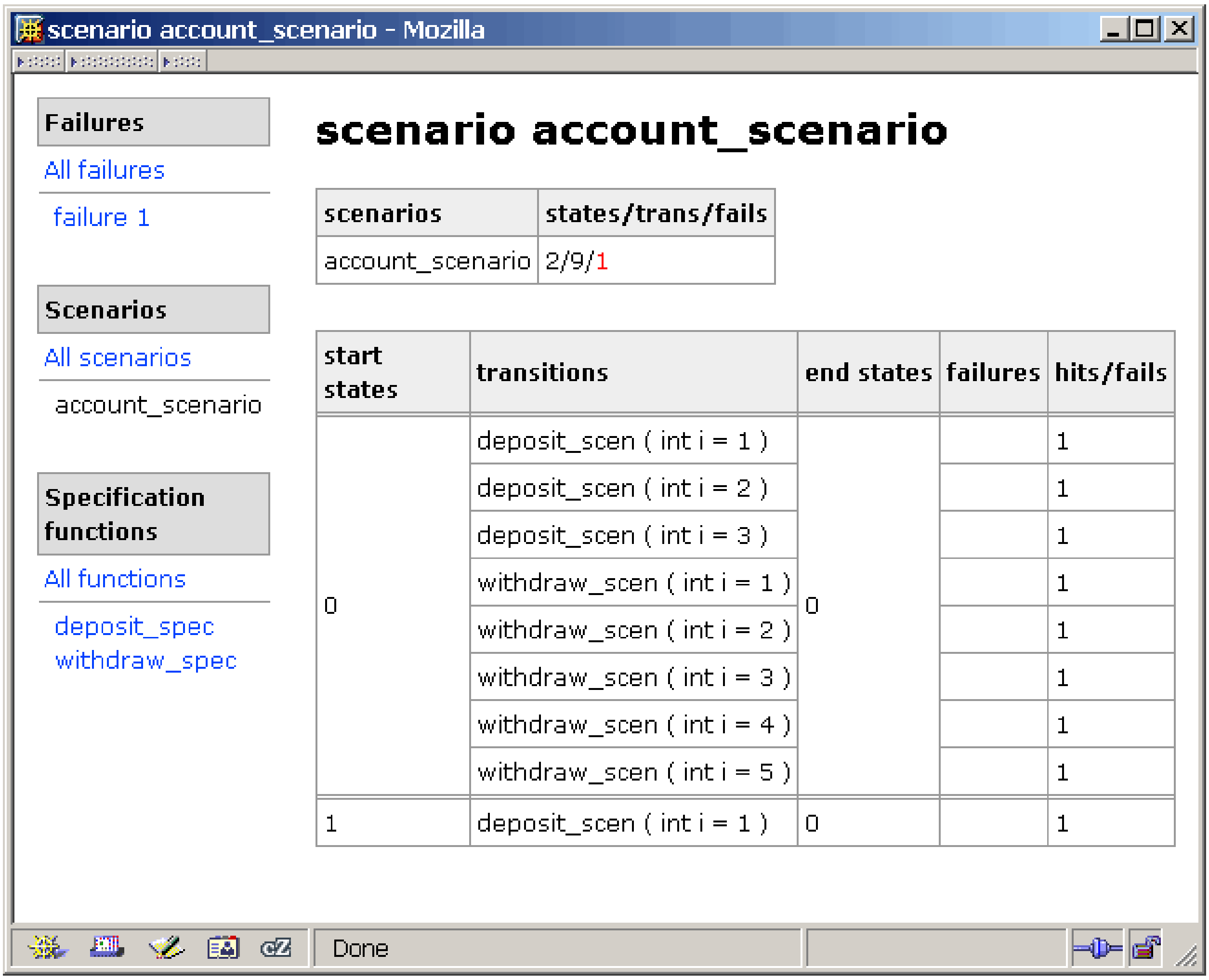
failure 1:

```
No back arc
 The dfsm test engine can be used only for scenarios with mutually
 Current state is: 0
 All the states reachable from it are completely tested
 State '1' may have untested transitions, but it is unreachable
```

location	
trace	C:\Program Files\CTesK\examples\account\trace.xml, line 200
occurence	
scenario	account_scenario
state	0

Figure 8. “No back arc” failure report.

There is a situation when some transition has been completed and it is not possible to go back.



Failures

All failures

failure 1

Scenarios

All scenarios

account_scenario

Specification functions

All functions

deposit_spec
withdraw_spec

scenario account_scenario

scenarios	states/trans/fails
account_scenario	2/9/1

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

Figure 9. Scenario detail report with "no back arc" failure.

As we can see, there has been a transition from the state **1** to the state **0**, all the transitions from which lead to the same state **0**. This suggests incorrect factorization, since the generalized state has been chosen in such a way, that the strong connectivity condition be violated.

Let us now consider the code of the function, which returns the model state:

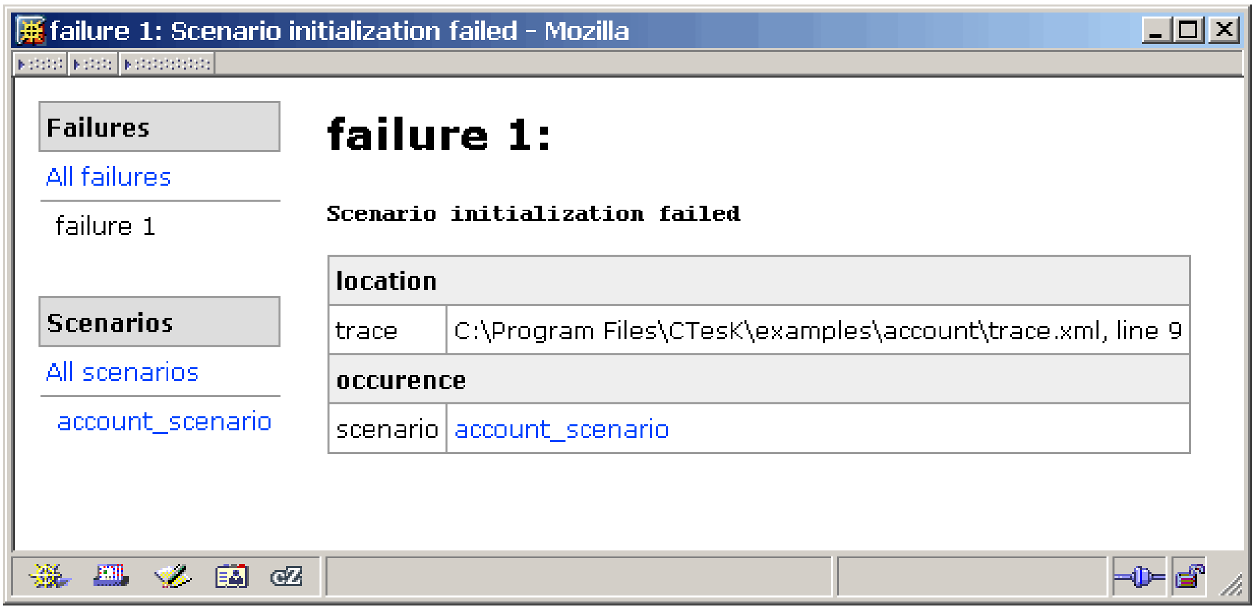
```
static Integer* account_state() {
  return create_Integer(acct.balance == 0);
}
```

Indeed, two generalized states, corresponding to the zero and nonzero balance, are determined here. However scenario functions do not provide parameters, which would allow the transition back to the nonzero balance state.

Error during initialization

Scenario initialization function can complete with an error if, for example, it was unable to initialize the system under test or allocate sufficient memory.

In the report error during initialization looks as follows:



failure 1: Scenario initialization failed - Mozilla

Failures

All failures

failure 1

Scenarios

All scenarios

account_scenario

failure 1:

Scenario initialization failed

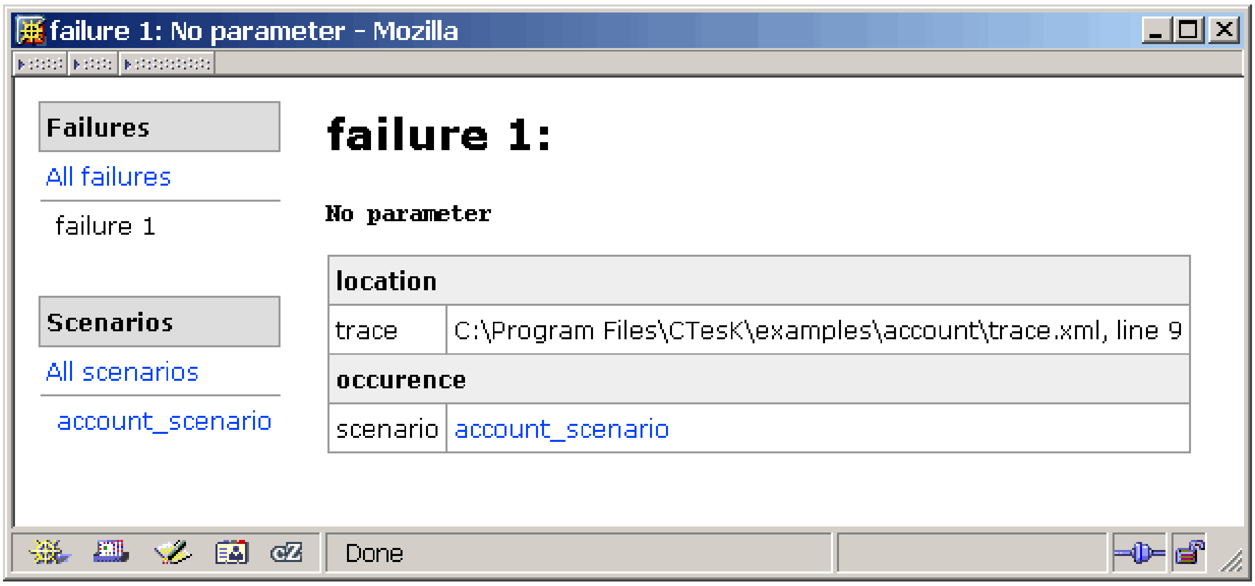
location	
trace	C:\Program Files\CTesK\examples\account\trace.xml, line 9
occurence	
scenario	account_scenario

Figure 10. Scenario initialization failure report.

If there is such error, one should search for its cause in the scenario initialization function.

Internal and user errors

Internal error results from a malfunction in the testing system. The user can also cause the error in the trace using the assertion function:



failure 1: No parameter - Mozilla

Failures

All failures

failure 1

Scenarios

All scenarios

account_scenario

failure 1:

No parameter

location	
trace	C:\Program Files\CTesK\examples\account\trace.xml, line 9
occurence	
scenario	account_scenario

Done

Figure 11. Report with user defined failure

The name of such error will contain the text, specified in the assertion function. The context allows localizing the place where the error emerged: since the context contains neither a state nor a transition, the error has occurred before the test engine started, i.e. in the scenario initialization function.

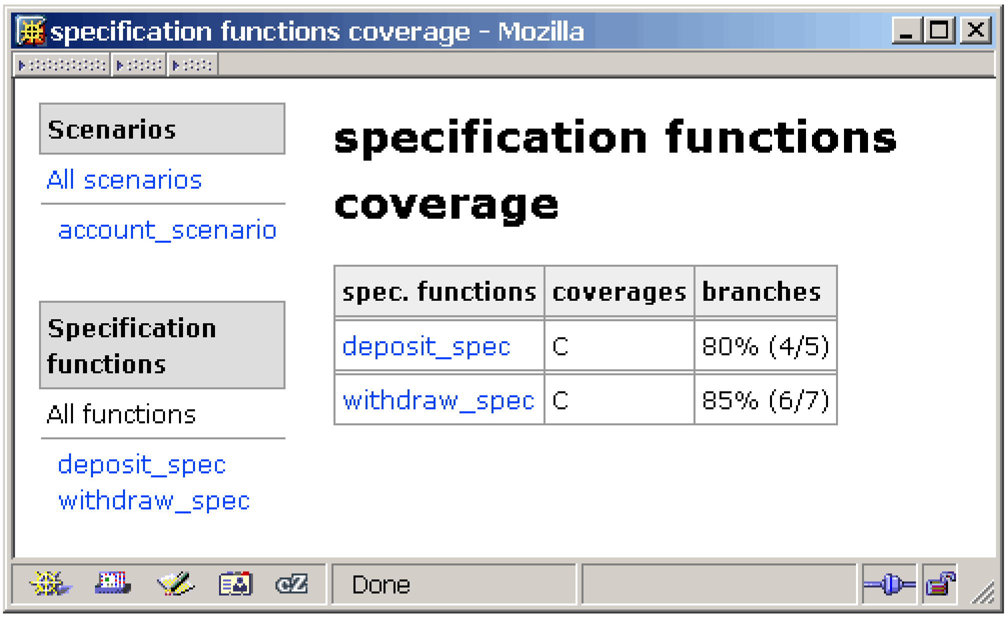
Let us check the code of the initialization function:

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

In this case the test has been launched without passing the necessary parameter from the command line into it, which has caused the error.

Coverage completeness analysis

Analysis of the completeness of coverage, achieved during testing, is performed using the report of the coverage of specification functions. Let us consider the report:



specification functions coverage

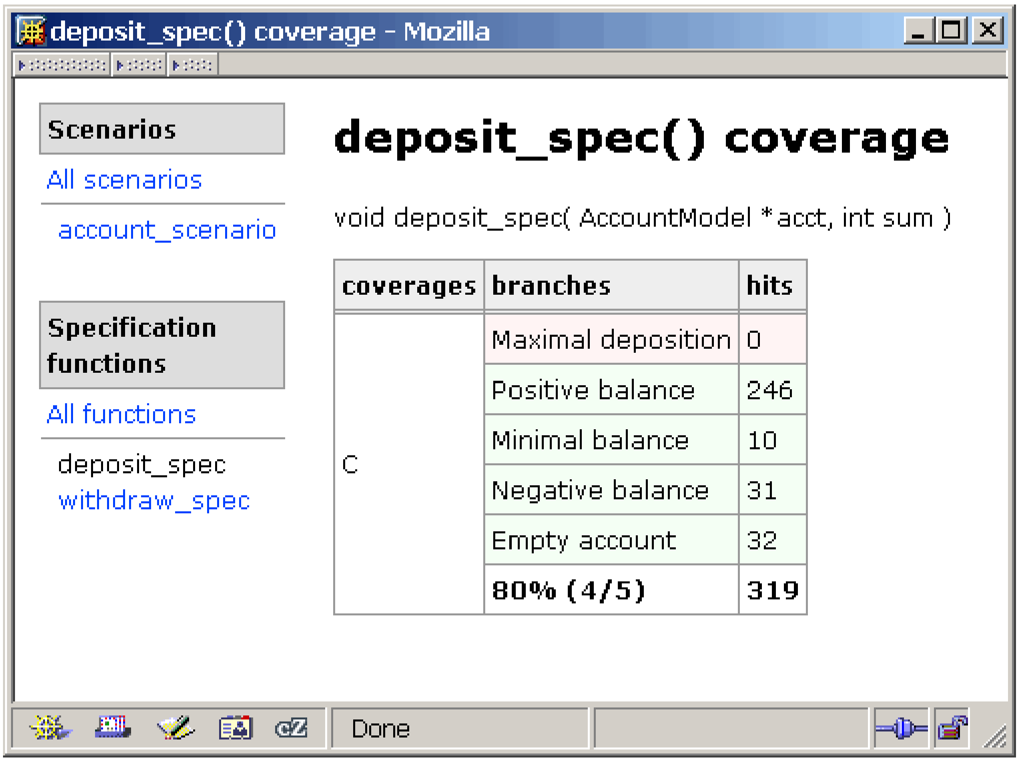
spec. functions	coverages	branches
deposit_spec	C	80% (4/5)
withdraw_spec	C	85% (6/7)

Figure 12. Specification functions' coverage report.

The table lists all specification functions, which have been tested. For each coverage criterion of each function the table lists percentage of the coverage of the functionality branches of this coverage as well as number of calls of this specification function and number of errors detected during these calls (if any).

For example, the function **deposit_spec** here has one coverage criterion **C**. In it, **80%** of the functionality branches were covered (**4** out of **5**), and there have been **319** calls of this function altogether.

For detailed information about coverage it is necessary to see the report about coverage of this function:



Scenarios

All scenarios

account_scenario

Specification functions

All functions

deposit_spec
withdraw_spec

deposit_spec() coverage

void deposit_spec(AccountModel *acct, int sum)

coverages	branches	hits
C	Maximal deposition	0
	Positive balance	246
	Minimal balance	10
	Negative balance	31
	Empty account	32
	80% (4/5)	**319**

Figure 13. Specification coverage detail report.

It is clear from this report that the "**Maximal deposition**" branch has never been passed (it is highlighted in red). The other branches have been passed at least once (they are highlighted in green).

In case when some branch was not covered, one should either change the iteration of the specification function parameters or create an additional scenario function, or create a separate scenario for the additional testing of this branch.

Examples of CTesK usage

Systems that provide API

The approach considered in this section is applicable to the testing of systems provided application program interface (API), which is a set of functions, using which the application can communicate with the system. A typical example of such a system is a library containing mathematical functions or functions for the work with some data structures.

As an example of CTesK tool usage, the process of testing of the system for queue handling is considered.

Description of the target system interface

A queue is a container, which implements the *FIFO* (*First In—First Out*) semantics. There are two major operations defined for the queue: element addition and deletion; herewith the element added first to queue is extracted from it first. The elements are passed to the queue using the non-typified references to them.

A queue is implemented as a unidirectional list, and the first element of the list does not contain an element of the queue.

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

The target system interface contains the following functions:

- `struct queue *create_queue (void)` — creates an empty queue
- `void delete_queue (struct queue **queue)` — deletes the queue and resets the pointer to it

- `int empty (struct queue *queue)` — checks whether the queue is empty
- `void enq (struct queue *queue, void *item)` — adds the nonzero element `item` to the end of the queue
- `void *deq (struct queue *queue)` — returns the first element and deletes it from the queue

Functional requirements to the system are formulated as follows:

- A queue can contain only those pointers, which are not equal to `NULL`.
- The `create_queue()` function returns a non-zero pointer to the new empty queue. This pointer can be passed to other system functions as a parameter. The queues created earlier are not changed.
- A queue can be deleted using the `delete_queue()` function. After that, the old value of the pointer to the deleted queue cannot be used as a parameter of other system functions.
- A queue is checked for emptiness using the `empty()` function. A nonzero value returned by this function corresponds to an empty queue and a zero value corresponds to a queue, which is not empty.
- An element is added to the queue using the `enq()` function.
- An element kept in the queue and placed there earlier than the others may be extracted from the queue by the `deq()` function. The function returns this element and deletes it from the queue.

Specification development

A specification consists of *specification functions*, which describe the behavior or the system under test, and of a *specification model of data*, which contains additional information necessary for the behavior description.

Specification model of data

A specification model contains description of types and data, in terms of which the system under test behavior is described.

Target system considered in this example works with the following data:

- Queue element
- The queue itself
- A set of queues created
- An integer value, which indicates the emptiness of the queue (returned by the *empty* function)

Queue element

A queue element is a pointer of the `void*` type. It follows from the requirements to the system that the element of the queue cannot be equal to `NULL`. The SeC language allows describing the data type with restrictions for the values of this type (*invariant*). To do this, the typedef construction is used with the **`invariant`** keyword.

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

This definition introduces a new type `item_t` with an invariant, which is structurally equivalent to the `void*` type. The invariant is described below by the following construction:

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

Queue

A queue is an ordered list of elements contained in it. It would be possible to use the queue implementation structure but, firstly, it will complicate the specification and, secondly, such specification will be implementation-dependent. It is much more convenient to use the type `List` defined in the CTesK specification type library[7].

The library of specification types contains a set of predefined types (including container types), a set of standard functions for working with data of these types (copying, comparing etc.) and general mechanism of memory management, occupied by the data. The data of specification types are always stored in dynamic memory and are accessed via the *specification reference*, which is the pointer of a specification type.

The type `List` can store elements of a specification type only. Using the construction **`specification`** `typedef` it is possible to create a specification type, corresponding to some type of the C language.

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

To create a specification type object, the library function `create()` is used. The first parameter of this function is a pointer to the *descriptor* of the type of the object to be created. The *type descriptor* is a structure containing information the system needs for working with objects of this type. For each specification type a variable is automatically defined, which contains the type descriptor and has the name, obtained from the type name with the prefix `type_` (`type_Item` in our case). The other parameters depend on the specific type: in our case, the value of the type `item_t` is passed to the function. For convenience, we can describe an auxiliary function of the creation of object of the type `Item`.

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

To simplify work with queue models, let us describe two functions: adding an item and its removing.

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

The function `add_list_aux()`creates the object of the type `Item` on the basis of the item value, and places it to the end of the list. The `append_List()` library function is used for the latter action.

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

The function `remove_first_aux()`uses the library function `get_List()`, which returns the list item according to the index specified, and function `remove_List()`, which deletes an item from

[7] For more details about the library specification types refer to the chapter "*Library of specification data types*" of the document "*CTesK 2.2 Community Edition. SeC Language Reference*".

the list according to the index specified. In order to access the value of the type item_t via the reference to the object of the type Item, one can just dereference this reference.

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

Set of the queues created

Since the queues, with which the system under test is working, are located in dynamic memory, this memory is a state of the system. It is impossible to model all memory, and it does not make sense either. To describe functionality, it is sufficient to know that some pointer is a pointer to a queue. To model a set of existing queues one can use a map of pointers to implementation queues in their model.

For working with maps, the library type Map is used. Since the data, located by the pointer to implementation queue, are not important for description of the system, and, secondly, from a point of SeC language view, a typed pointer refers to a single value of the corresponding type, then a map key should be the type void*. For convenience, it is possible to describe a typedef name for this type. The same way, it is possible to describe a corresponding constant for representation of a zero pointer.

```
typedef void* queue_t;

const queue_t null_queue = NULL;
```

When using the type Map, both the keys of the map and its values should be objects of the specification types. Therefore one should define a specification type corresponding to the type queue_t. For convenience one can also describe a function for the creation of the object of this specification type.

```
specification typedef queue_t Queue;

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

To simplify work with the model, let us describe the following auxiliary functions related to the queue identifier:

- Check of correctness of some identifier, i.e. of its existence as a key in a given map;
- Obtaining the queue model representation via the queue identifier;
- Queue deletion from the map;
- Obtaining the queue size via its identifier.

The function exists_queue_aux checks whether a given queue identifier is a key in a given map via the library function containsKey_Map.

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

The function get_queue_aux() returns the model queue, corresponding to the given identifier, via the library function get_Map() returning the value in the map by the key.

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

To add a queue, the function `add_queue_aux` is used, which calls the library function `put_Map()`, adding a {*`key`*, *`value`*} pair into the map.

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

The function `remove_queue_aux()` uses the library function `remove_Map()` to remove a key and a corresponding value from the map.

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

To obtain the number of elements in the queue, the function `size_queue_aux()` uses the above mentioned function `get_queue_aux()` and the library function `size_List()`, which returns the length of the list.

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

Now we can define the global variable `model_queues` containing the specification model state. The zero pointer does not point to any queue, and this restriction can be included in the variable invariant. The variable with an invariant is defined as a usual variable with addition of the keyword **`invariant`**. The invariant is described using a special construction.

```
invariant Map *model_queues;

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

During testing, the model state should be created before tested functions are called. In our case it is necessary to create an object of the type `Map` and assign a reference to it to the `model_queues` variable. The function for the creation of the type `Map` object accepts two additional parameters as an input, namely the pointers to the type descriptors of keys and values of the map.

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

The value denoting the queue emptiness

As a value, returned by the function, checking whether the queue is empty (`empty()`), the type `bool` will be used, which can assume two values: `true` and `false`.

Specification of behavior

The behavior of the functions tested is described in the form of *specification functions*. The definition of the specification function consists of three parts:

- Function signature, which is analogous to the signature of the usual function (returned value, function name, and its arguments) and contains the **`specification`** keyword;

- Access constraints to the global variables and parameters;
- Body of the specification function.

Access constraints to global variables can be of three types: reading, writing, and updating. If the behavior of the function depends on the variable's value, but this value does not change as a result of completion of the function, then this function provides reading access to this variable. If the behavior of the function does not depend on the variable's value, but after the call of the function this variable will contain the results of completion of this function, then this function provides writing access. In the case of the updating access constraint, the behavior of this function depends on this variable's value, and this value can be changed as a result of the work of the function.

The behavior of the tested function is described in the body of the specification function as a pre- and postcondition. A *precondition* describes a definitional domain of the target function. A *postcondition* describes the function behavior itself in terms of dependence between the values of the parameters and global variables before and after the call of the function. Besides, the body of the specification function may contain coverage criterion descriptions. The coverage criterion defines the input data breakdown into subareas, in which the function behavior differs significantly. Each of these subareas is called a *functionality branch*.

Queue creation function

The signature of the specification function, describing the *create_queue* function behavior, looks like the following:

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

The queue creation function changes the system state by adding a new queue, so the access constraint will apply to the change of the variable `model_queues`.

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

It is necessary to describe the following requirement in the postcondition of the function formally:

The function `create_queue()` returns the nonzero pointer to the new empty queue. This pointer can be passed to other system functions as a parameter. The queues created earlier stay unchanged.

In the postcondition, there should be a check if the function has returned the queue identifier not equal to `null_queue` (1), corresponding to an empty queue (2), and not corresponding to any existing queue directly before the call of the function (3).

In the postcondition, to access a value returned by the function, the name of the function is used, `create_queue_spec` in this case. To check the third statement, it is necessary to have an access to the value of `model_queues` map before the function was called, i.e. to its *prevalue*. However, the global variables and variable arguments in the postcondition have *postvalues*, i.e. the values obtained after the call of the function. To access a prevalue of some variable or expression, the `@` operator is used. In our case, we need to get the map prevalue:

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

Now we need to check if the other queues did not change. To do this, we can create a copy of the current `model_queues` map (the variable itself must not be changed since the specification function should not have a side effect), delete a newly created queue from it and compare to the state of the system before the function was called. Comparison of two objects of a specification type is performed using the library function `equals()`.

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

The postcondition is described in the body of the specification function in a block marked by the **`post`** keyword.

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

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

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

Queue deletion function

The signature and access constraints of the function look as follows:

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

The requirements to the queue deletion function are provided below:

The function `delete_queue()` accepts a pointer to the queue as an input. If the pointer's value is equal to `NULL`, the function does nothing. Otherwise this value should point to an existing queue. The corresponding queue is destroyed. The `NULL` value is written by the pointer parameter. After that, the old value of the pointer to a deleted queue cannot be used as a parameter of other system functions.

It follows from the requirements that the function parameter should point either to the `NULL` value or to the pointer to an existing queue. Such conditions are formulated in a precondition of specification functions. The precondition is described in the body of the specification function in a block marked by the **`pre`** keyword.

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

Depending on the value of the pointer `pqueue`, two functionality branches are singled out explicitly in requirements to the function. These branches should be described in a *coverage criterion*. The coverage criterion is described in a block marked by the **`coverage`** keyword and its name:

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

For each functionality branch, there should be the return operator in this block with two values, surrounded by curly braces. The first value is the identifier of a functionality branch and the second value is a string with a description of the branch.

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

Different checks should be formulated for these two branches in the postcondition. So that not to repeat calculations performed during definition of the coverage criterion, the construction `coverage()` is used, to which the coverage identifier is passed. The returned value is the identifier of the functionality branch, which was achieved in this coverage.

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

In the case of a zero pointer one should check that the `model_queues` map value and the value of the pointer `pqueue` do not change.

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

Otherwise, the pointer value should be equal to `null_queue`. In order to check if the queue were deleted, one can create a copy of the map before the function has been called, delete the corresponding key and value from it and compare to the current value.

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

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

Finally we have the following specification function:

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

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

Queue emptiness check function

A signature of the specification function, which describes behavior of the function for queue emptiness checking, is represented below.

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

The function for queue emptiness checking should not change the queue, but its behavior depends on the state. Therefore it provides reading access to the `model_queues` variable.

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

The function must correspond to the requirements provided below.

The function `empty()` accepts a pointer to the existing queue as an input and returns a nonzero value if the queue is not empty, and zero otherwise.

The function has a precondition, which checks if the queue exists:

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

For this function, two functionality branches can be singled out, which correspond to an empty and non-empty queue.

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

The following statement should be formulated in the postcondition: the value returned should be `true` if the queue is empty and `false` otherwise.

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

Complete specification function is provided below.

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

Element addition function

The function `enq()` accepts a pointer to the existing queue and a non-zero pointer to the element placed as an input. The queue does not cease to exist. The element is added to the corresponding queue. The other queues stay unchanged.

This function changes the queue, therefore it provides updating access to the variable model_queues.

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

According to the requirements, the function has a precondition: the queue must exist and the element must not be equal to zero. Since the item parameter is of the type item_t, for which the invariant is defined (non-equivalence to the NULL value), this restriction will be checked automatically, and the function precondition is the queue existence checking:

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

For this function, a coverage criterion can be defined, which is analogous to the criterion of coverage of the function empty_spec().

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

Elements in the queue model are arranged according to the order of placing them into the queue: the elements with a lesser index were placed into the queue earlier. Therefore, a new element should be added to the end of the queue, and the other elements should stay the same. In order to check that the other queues do not change, one can place the obtained queue (tmp_list) into the model_queues prevalue copy, and compare to the model_queues postvalue.

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

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

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

Below follows the complete specification function.

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

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

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

Element removing function

The function `deq()` accepts a pointer to the existing non-empty queue as an input. The queue does not cease to exist. The function deletes the element from the queue, which was added to the queue before the other elements in it were. The function returns the removed element. The other queues stay unchanged.

Two functionality branches can be singled out in the coverage criterion: the first branch corresponds to the extraction of the last element, and the second branch corresponds to all other cases.

The specification function can be written using the same procedure as in the case of the previous function.

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

    add_queue_aux (tmp_map, queue, tmp_list);

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

CTesK has the capability to group specification functions into subsystems. This information is used by the report generators. To group the specification functions developed into the **queue** subsystem put the following string at the beginning of the specification source code file:

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

The "queue" string literal is used by the report generators to show the subsystem name. If it is equal to the subsystem identifier, as in our case, it can be omitted. The univocal correspondence between subsystems identifiers and names should exist.

A complete specification is shown in the "*Appendix A*". The specification is located in two files: the file **queue_specification.seh**, which contains declarations of types, variables, and functions, and the file **queue_specification.sec**, which contains their definitions. These files are also available in the **examples/queue** folder of the CTesK distribution.

Development of mediator functions

Mediator functions are intended to establish correspondence between the implementation functions and their relative specification functions. In a general case, a specification may be used to test various implementations of systems that are of similar functionality. Specification does not contain any assumptions regading the structure of an implementation. For example, any designed specification matches a system that stores queue element in a form of an array not a unidirectional list. To test such a system, it is enough just to write other mediators.

Each mediator function must invoke a relative implementation function and bring the model state in compliance with results of such invocation. In the case of the system under consideration, a model presentation of the queue can be built up on the basis of the given identifier of the queue (a value of the type `queue_t`), which is actiulaly a pointer to the implementation queue. To do so, one should walk along the unidirectional list of the implementation queue (ignoring the first "false" element), with adding relative elements to an object of the `List` type. In order to create an object of the specification type `List`, the pointer to the descriptor of the type `type_List` and the pointer to the list element type (`type_Item`) should be passed to the `create()` function.

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

  if (null_queue == queue) return NULL;

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

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

  return model;
}
```

The function `queue_to_list()` should be invoked for all the identifiers of the queue which represent the keys of `model_queues` map. The function `key_Map()` is used to enumerate the map keys. If such function is invoked within a cycle for a certain map of the size `size` and with the index values from zero to `size - 1`, then it is guaranteed that all map keys will be enumerated (in a certain voluntary sequence).

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

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

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

Definition of a mediator function consists of the following sections:

- the **`mediator`** keyword, the mediator name and the **`for`** keyword;
- the signature of a corresponding specification function and its access constraints;
- the body of the mediator function.

The mediator function signature for a queue creation function looks like the following:

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

The body of the mediator function consists of two blocks: a call-block where invocation of the tested function is perfromed, and a state-block, which alters the model state in compliance with the interaction.

The call-block of the function `create_queue_media()` should contain invocation of the function `create_queue()` and returning of a specification model representation of the result.

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

The state-block should contain ivocation of the function `queue_map_state_up()`, which transforms state for previously existing queues, and invokes of `add_queue_aux()`, which will add a specification model representation of a newly-created queue to the state.

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

The mediator funtion of deleting a queue looks as follows:

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

The call-block: invocation of the function `delete_queue()`.

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

A corresponding queue should be deleted from the map, and the function of evaluating a state should be invoked within the state-block. As far as the value of `*queue` alters after the function has been invoked, it should be saved in a local variable.

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

The call-block of the mediator function of checking a queue for emptiness contains conversion of the return value from `int` to `bool` type.

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

The mediators functions of adding and extracting an item return result of a call of the corresponding implementation function in the call-blocks, and invoke the function `queue_map_state_up()` in the state-block.

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

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

Detailed definition of mediator functions is represented in "*Appendix A*". Like as the specification, the mediator functions are in two files: **queue_media.seh**, which contains declarations of the mediator functions, and **queue_media.sec**, which contains their definitions as well as definitions of auxiliary functions and data. These files are also available in the **examples/queue** folder of the CTesK distribution.

Development of test scenario

In order to ensure testing completeness, behavior of the target functions must be tested in variou states. For instance, the functions of checking for emptiness, adding and extracting of an element should be tested in conditions of various queue lengths. For that purpose, a testing sequence must be developed which means the sequence of invocations of target functions with various parameter values.

CTesK automatically builds a testing sequence on the basis of a *test scenario* to be developed manually.

The following functions shall be defined within a test scenario:

- the function of initializing a scenario;
- the function of finalizing a scenario;
- the function of evaluating a scenario state;
- the scenario functions that define enumeration of parameters of the tested functions and their invoking.

A scenario used to test three implementation functions (`enq()`, `deq()`, and `empty()`) will be discussed within the example.

To do so, definitions of extra data will be necessary. First, one should limit the size of the tested queue, in order to provide that testing is not lasting "infinitely". Secondly, the data to be placed to the queue should be determined.

In order to limit the queue length, the integer variable `queue_max_size` is defined. To determine queue elements, the variable that holds their amount (`queue_items_num`) and the array `queue_items` are defined.

```
int queue_max_size = 10;

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

Thirdly, one should describe the variable to store the identifier of the single queue which is supposed to be tested:

```
queue_t queue;
```

The function of initializing a scenario must allocate a memory for such data and fill it in with the proper values. The function receives the parameters of the command line to process them and extract all necessary information. Additionally, the tested queue must be created, and its model presentation shall be synchronized with it. For that purpose, the specification function of creating a queue must be invoked. Complete initialization function looks as follows:

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

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

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

  queue = create_queue_spec ();

  return true;
}
```

The function of finalization of a scenario shall deallocate the resources, allocated by the scenario.

```
void queue_scenario_finish ()
{
  int i;

  delete_queue_spec (&queue);

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

If a scenario state is defined as a sequence of items that are in the queue, then the amount of such states is huge. On the other hand, it is enough to test functions under conditions of various lengths of the queue. Therefore, the scenario state may be defined as an integer number that is equal to a current length of the queue. In order to provide that the functions are tested in all such states, a function of evaluating a scenario state and parameter enumeration for the tested functions must be defined. Using the functions, CTesK testing system will automatically ensure testing in various states.

The function of evaluation of a state shall return an object of the specification type. CTesK specification type library offers the type `Integer`. To create an object of that type, the pointer to the type descriptor `type_Integer` and a value to initialize the object (a current number of queue items) must be passed to the function `create()`.

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

Now, a scenario functions for each tested function of the system should be written. In the simplest case, a scenario function performs a single *testing interaction* with the system under test in every individual scenario state. A testing interaction is described as an invocation of a specification function which describes behavior of the tested function. A scenario function has no parameters and returns value of the type `bool`. Within the present example, the scenario functions always return `true` as the indicator of successful performance.

The scenario function `empty_scen()` consists only of invocation of the function `empty_spec()` with the identifier of the tested queue as a parameter, and returning the `true` value.

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

The function `enq_spec()` should receive an item, which should be placed in a queue. For this purpose, the array `queue_items` has been prepared, every the item of the array should be placed into the queue in every state. In order to invoke the function `enq_spec()` in every state with the values of the parameter `item` for all indexes within the array `queue_items`, the SeC construction **`iterate`** should be used.

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

The statement **`iterate`** is similar, from the syntax perspective, to the cycle `for` of C. The first expression of the cycle header (`int i = 0`) represents definition and initialization of an *iteration variable*. A scenario function has an individual copy of the iteration variable for every scenario state. The second expression (`i < queue_items_num`) is the condition of continuation of the cycle execution. The third expression (`i++`) represents the calculation of the next value of the iteration variable. The statement **`iterate`** has also the fourth expression (which is absent in the present case), the condition of the filtration. If the condition is not satisfied, the body of the cycle will not be executed for current value of an iteration variable.

During every execution of the scenario function in a certain state, the body of the statement **`iterate`** is executed with the next value of an iteration variable that corresponds to a state. The body of the given cycle will be executed in every state of the scenario with the values of the variable `i` equal to `0`, `1`, ..., `queue_items_num - 1`.

In order to limit the queue length, the function of adding an item should not be invoked as soon as the maximal queue length is achieved. For the purpose, a check is performed in the scenario function:

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

Complete scenario function follows below.

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

The function of removing an item from the queue has no parameters that would demand enumerating, however, there is a precondition defined for it, which demands that the queue is nonempty. So, the scenario function may be defined as follows:

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

Now, the *scenario variable* shall be described, with all of the above defined functions to be included in it. A scenario variable has the type `dfsm` and is initialized in the style of C99 standard, by means of enumerating of the names of the fields and their values. The fields that correspond to the functions of initialization and finalization of a scenario are called `init` and `finish`, respectively. The function of evaluating a generalized state initializes the field `getState`. The filed with the name `actions` represents an array where the list of scenario functions that finishes with `NULL` is contained.

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

The files containing the test scenario (**queue_scen.seh** and **queue_scen.sec**) are represented in "*Appendix A*". These files are also available in the **examples/queue** folder of the CTesK distribution.

Main function of test

At this point, the function `main()` that starts execution of the test scenario shall be described.

Prior to run the scenario itself, setting up of mediators should be provided for the specification functions. To do so, the functions automatically generated out of the specification functions are used:

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

Besides that, a model state shall be created by means of previously defined function `init_state_queue()`.

```
init_state_queue ();
```

In order to run the test scenario, the function with a name which is the same one as the scenario variable name (`queue_scenario` in this case) should be invoked by passing to it the command line options.

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

Complete `main()` function looks as follows:

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

  queue_scenario (argc, argv);
}
```

Building and running test

In order to obtain an executable test, the developed files shall be translated to C, then the files and a file which contains the implementation, shall be compiled and linked into an executable module with CTesK libraries.

The resulting executable test is able to trace a report about its completion for further analysis. In order to ensure that tracing achieves the console, the parameter **–tc** should be passed to the test. As the parameter **–t** followed by the file name indicates that the test tracing shall be directed to the file specified.

On Linux platform translation shall be executed in the following way:

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

Compilation is performed with the following commands:

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

Test linking is performed as follows:

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

Start of the test obtained:

```
% queue_test –tc –t queue.trace 5 10
```

Report generation and result analysis

The test sends the trace of its execution in the XML format. In order to extract the data on the test coverage and inconsistencies found between the specification and the implementation out of the tracing, the *trace analyzer* is used. For the tracing analyzer, the directory should be specified where HTML-files with the test report should be generated and where a file that contains the trace to be analyzed is located.

Linux platform:

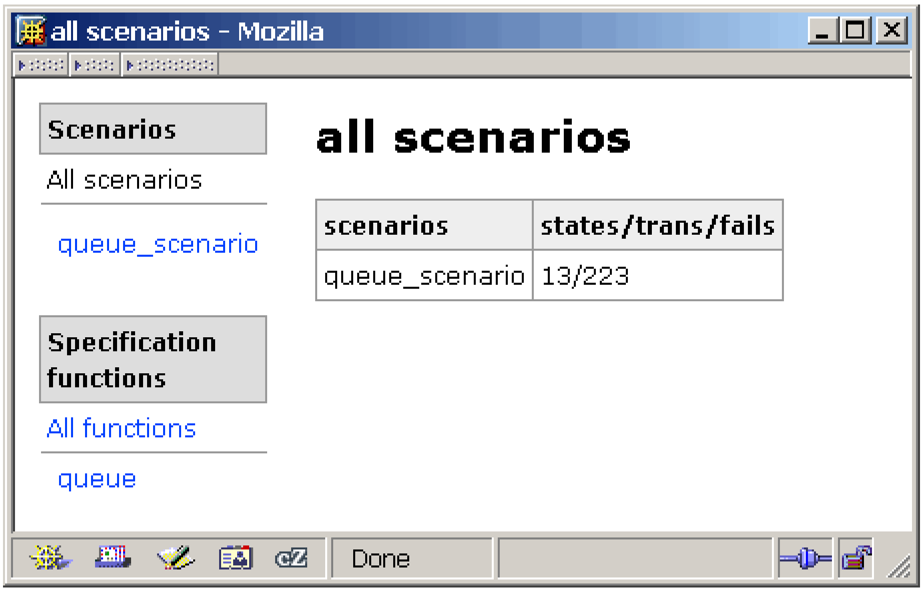
```
% ctesk-rg.sh –d report queue.trace
```

To view the test report, HTML browser should be invoked, and the file **index.html** from the directory specified should be opened.

Linux platform:

```
% mozilla report/index.html
```

In the opened window of the browser, the report on all scenarios that have been executed within the given test will be represented (Figure 14).



Scenarios

All scenarios

queue_scenario

Specification functions

All functions

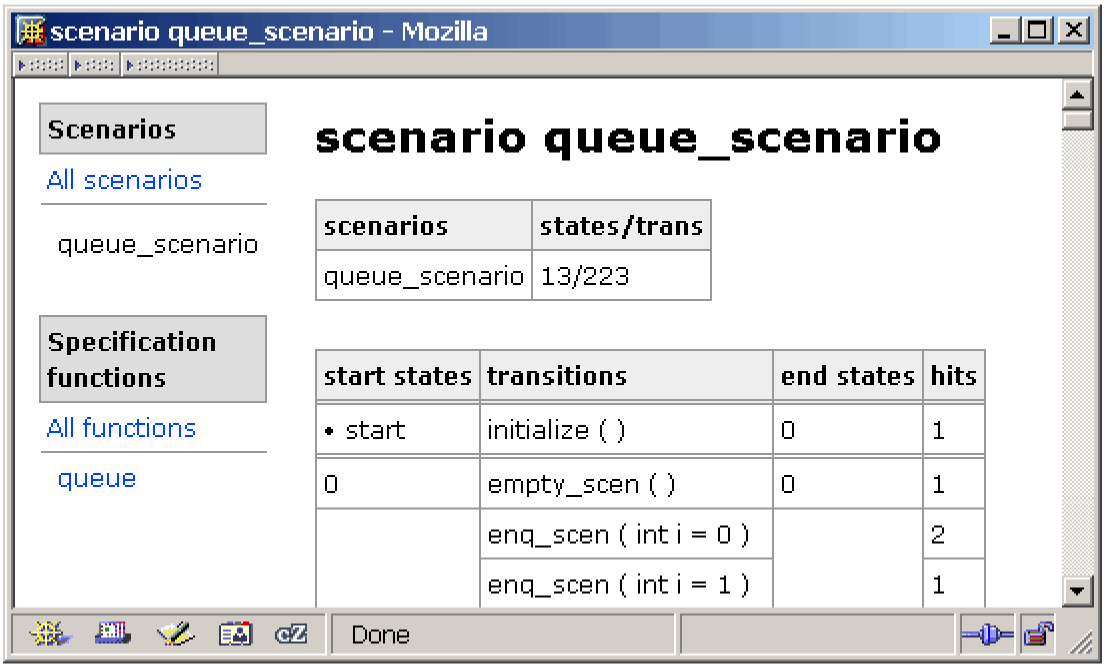
queue

all scenarios

scenarios	states/trans/fails
queue_scenario	13/223

Figure 14. General view of the test report.

Following the reference **queue_scenario**, one may obtain the data about passed states of the tested queue (Figure 15). The first table contains data on the scenarios executed within the given test, such as the name, the amount of states and the amount of transitions between the states. Then, a detailed description follows for transitions for every scenario. For each transition, the initial state, scenario function with the use of which the transition has been executed, the values of iteration variables, the staty after the transition, and the amount of executions of the transition within the scenario operation period, are described.



Scenarios

All scenarios

queue_scenario

Specification functions

All functions

queue

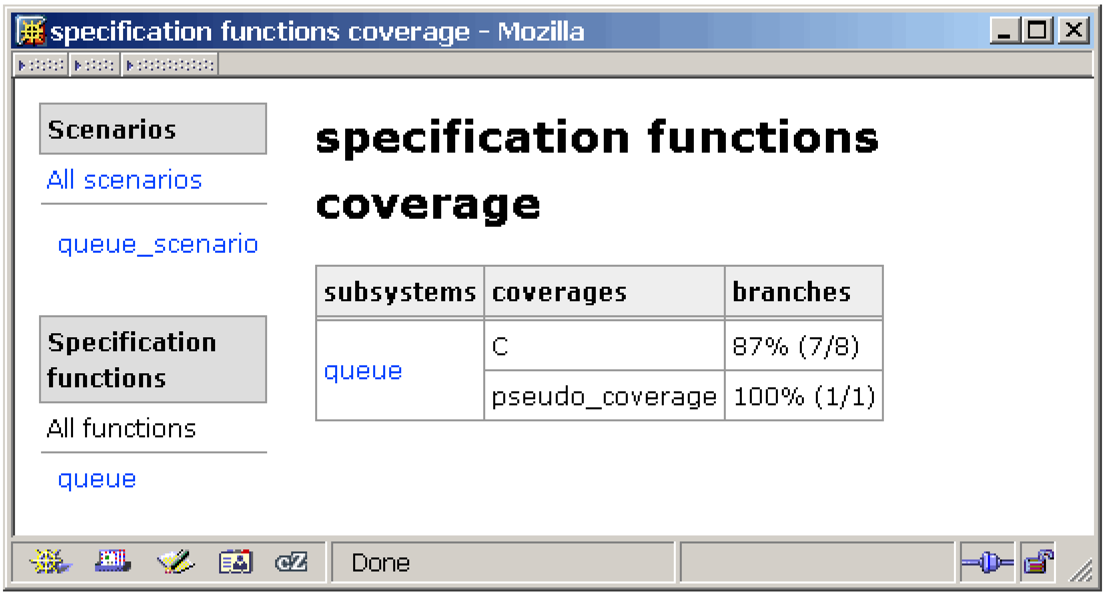
scenario queue_scenario

scenarios	states/trans
queue_scenario	13/223

start states	transitions	end states	hits
• start	initialize ( )	0	1
0	empty_scen ( )	0	1
	enq_scen (int i = 0)		2
	enq_scen (int i = 1)		1

Figure 15. Test scenario report.

In accordance with the reference *All functions*, a table is located that contains the data of the coverage of the tested subsystems (Figure 16). The first column contains the names of the subsystems. In respect to each subsystems, the second column contains the coverage criteria defined for its specification functions. Within the column *branches*, the table contains percentage of coverage of branches in respect to each coverage criterion the number of hits to a given branch.



Scenarios

All scenarios

queue_scenario

Specification functions

All functions

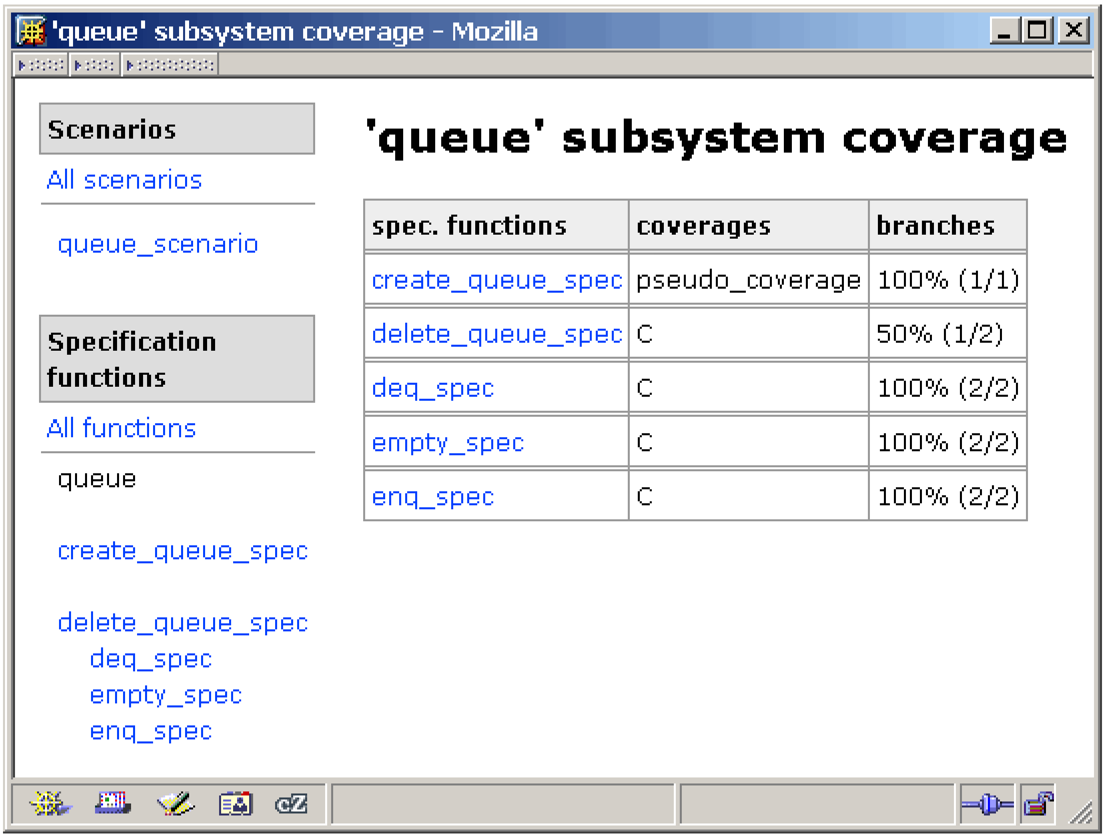
queue

specification functions coverage

subsystems	coverages	branches
queue	C	87% (7/8)
	pseudo_coverage	100% (1/1)

Figure 16. Specification function coverage report.

In accordance with the reference *queue*, a table is located that contains the data of the coverage of the tested functions of the *queue* subsystem.. The first column contains the names of the specification functions. In respect to each function, the second column contains the coverage criteria defined for it. Within the column *branches*, the table contains percentage of coverage of branches in respect to each coverage criterion and the number of hits to a given branch.



Scenarios

All scenarios

queue_scenario

Specification functions

All functions

queue

create_queue_spec

delete_queue_spec

deq_spec

empty_spec

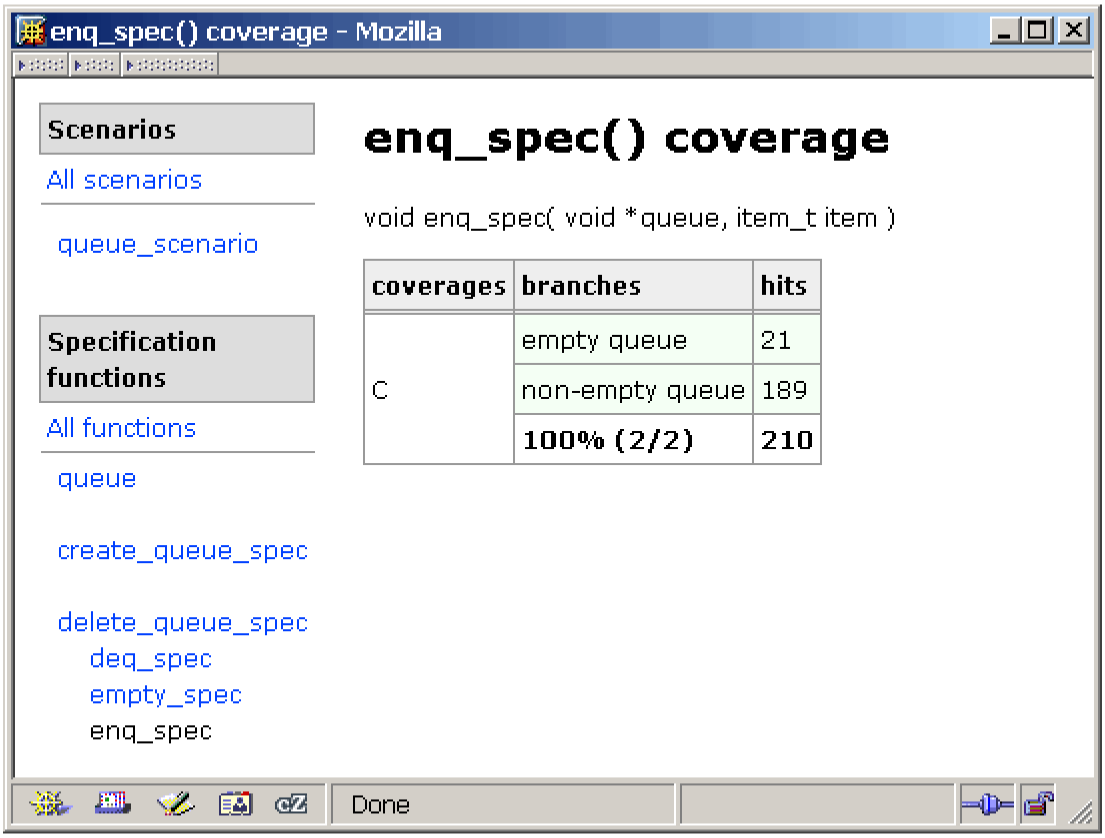
enq_spec

'queue' subsystem coverage

spec. functions	coverages	branches
create_queue_spec	pseudo_coverage	100% (1/1)
delete_queue_spec	C	50% (1/2)
deq_spec	C	100% (2/2)
empty_spec	C	100% (2/2)
enq_spec	C	100% (2/2)

Figure 17. Specification function coverage report.

More details on the function coverage can be found by clicking on its name (Figure 18). The first column of the appearing table will represent names of coverages, while the second column will show names of the functionality branches in every coverage criterion, and the third, the number of hits at the branches. The last table line that corresponds to a coverage criterion contains data on completion of the criterion, i.e. percentage of coverage by branches and the total number of invocation of a function.



enq_spec() coverage

void enq_spec(void *queue, item_t item)

coverages	branches	hits
C	empty queue	21
	non-empty queue	189
	100% (2/2)	**210**

Figure 18. Detailed report on specification function coverage.

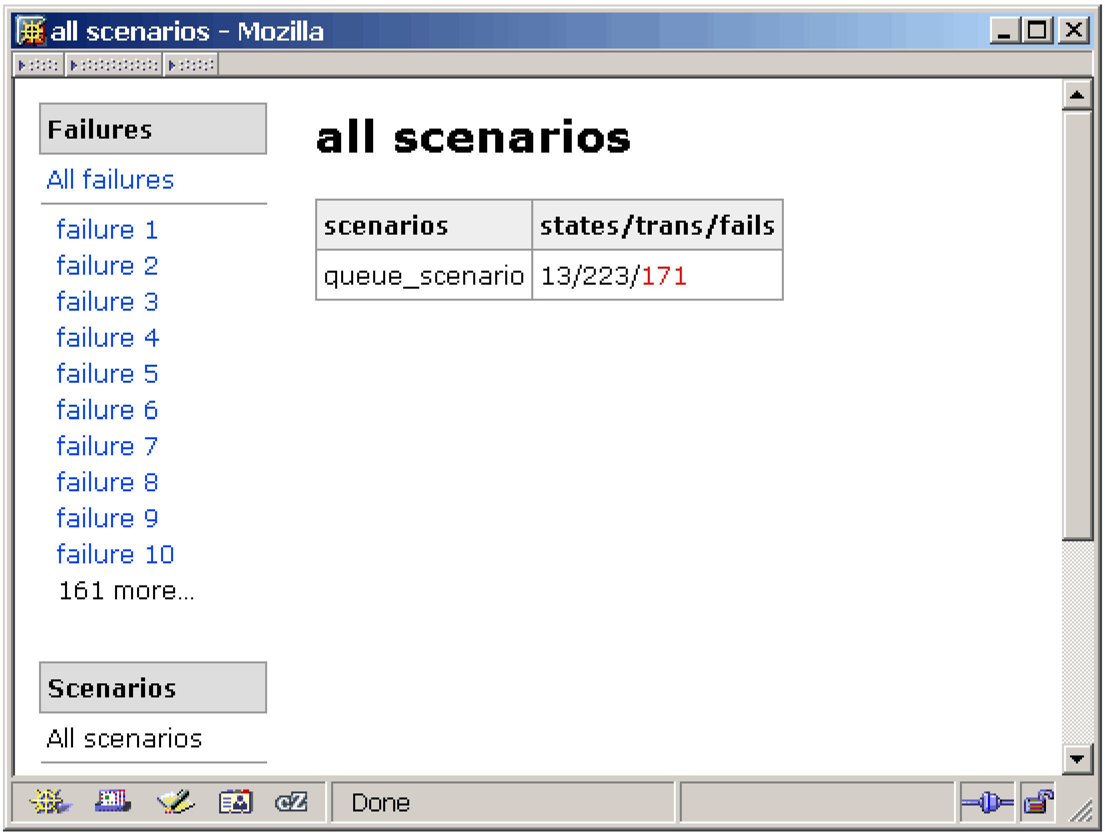
The given test found no errors, and therefore, there is no data on errors in the report. One may alter the implementation to demonstrate a report on inconsistencies found. As an example, let us alter the function `enq()` so that it places each item to the beginning of the list instead of its end.

```
...

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

...
```

Now, the implementation file **queue.c** should be compiled, the executable module relinked, and then one should start the test and regenerate the test report (Figure 19).



Failures

All failures

failure 1
failure 2
failure 3
failure 4
failure 5
failure 6
failure 7
failure 8
failure 9
failure 10
161 more...

Scenarios

All scenarios

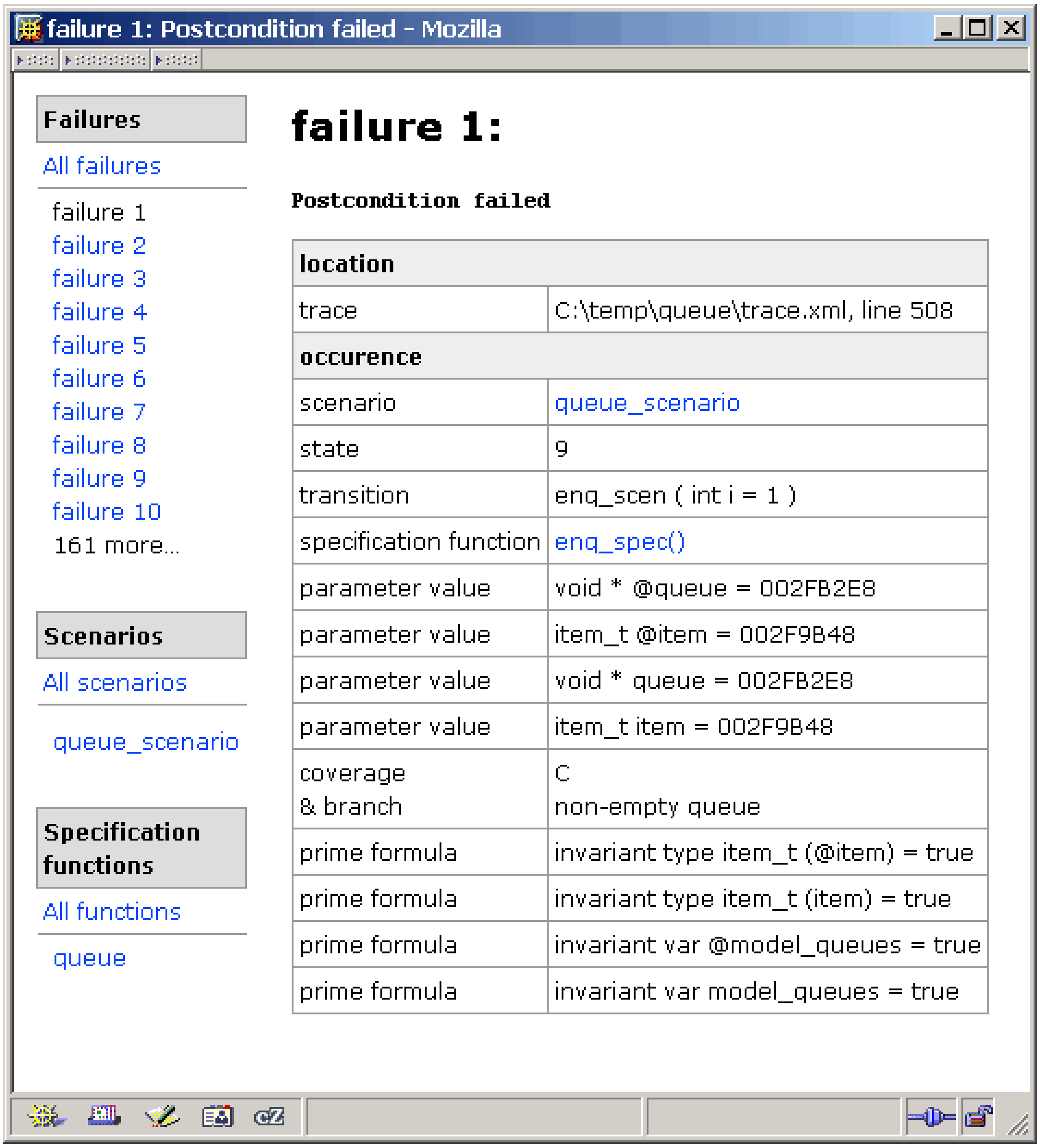
all scenarios

scenarios	states/trans/fails
queue_scenario	13/223/171

Figure 19. General view of the report, containing information about failure.

The list of failures appears on the left. The report on each failure contains the following data (Figure 20):

- Failure type (postcondition violation in this case)
- Failure location in the test trace
- The testing system components where the failure emerges (the scenario name, state, the scenario, and the specification function)
- Parameter and coverage elements values



Failures

All failures

failure 1
failure 2
failure 3
failure 4
failure 5
failure 6
failure 7
failure 8
failure 9
failure 10
161 more...

Scenarios

All scenarios

queue_scenario

Specification functions

All functions

queue

failure 1:

Postcondition failed

location	
trace	C:\temp\queue\trace.xml, line 508
occurence	
scenario	queue_scenario
state	9
transition	enq_scen (int i = 1)
specification function	enq_spec()
parameter value	void * @queue = 002FB2E8
parameter value	item_t @item = 002F9B48
parameter value	void * queue = 002FB2E8
parameter value	item_t item = 002F9B48
coverage & branch	C non-empty queue
prime formula	invariant type item_t (@item) = true
prime formula	invariant type item_t (item) = true
prime formula	invariant var @model_queues = true
prime formula	invariant var model_queues = true

Figure 20. Report on found failure.

Appendix A:
The code of the queue test

Implementation

queue.h

```
#ifndef __queue_h__
#define __queue_h__

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

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

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

#endif / __queue_h__ */
```

queue.c

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

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

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

  return q;
}

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

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

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

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

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

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

  return res;
}
```

Specifications

queue_spec.seh

```
#ifndef __queue_spec_seh__
#define __queue_spec_seh__

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

invariant typedef void *item_t;

extern invariant Map *model_queues;

typedef void *queue_t;

const queue_t null_queue;

specification queue_t create_queue_spec (void)
    updates model_queues;

specification void delete_queue_spec (queue_t *queue)
    updates model_queues;

specification bool empty_spec (queue_t queue)
    reads model_queues;

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

specification item_t deq_spec (queue_t queue)
    updates model_queues;

void init_state_queue (void);

specification typedef item_t Item;
specification typedef queue_t Queue;

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

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

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

#endif /* __queue_spec_seh__ */
```

queue_spec.sec

```
#include "queue_spec.seh"

#pragma SEC subsystem queue "queue"

const queue_t null_queue = NULL;
```

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

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

invariant Map *model_queues;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Mediators

queue_media.seh

```
#ifndef __queue_media_seh__
#define __queue_media_seh__

#include "queue_spec.seh"

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

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

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

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

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

#endif /* __queue_media_seh__ */
```

queue_media.sec

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

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

  if (null_queue == queue) return NULL;

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

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

  return model;
}

void queue_map_state_up (void)
{
  int qi;
  Map *old_model_queues = model_queues;
  model_queues = create (&type_Map, &type_Queue, &type_List);

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

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

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

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

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

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

Scenarios

queue_scen.seh

```
#ifndef __queue_scen_h__
#define __queue_scen_h__

extern scenario dfsm queue_scenario;

#endif /* __queue_scen_h__ */
```

queue_scen.sec

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

static int queue_max_size = 10;

static int queue_items_num = 20;
static item_t *queue_items;

static queue_t queue;

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

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

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

  queue = create_queue_spec ();

  return true;
}

void queue_scenario_finish ()
{
  int i;

  delete_queue_spec (&queue);

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

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

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

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

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

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

Function main

queue_main.sec

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

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

  init_state_queue ();

  queue_scenario (argc, argv);
}
```