CTESK 2.8:
User Documentation

Contents

1 Purpose of 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.

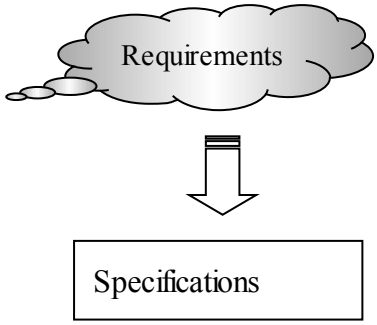
1.1 UniTESK Technology

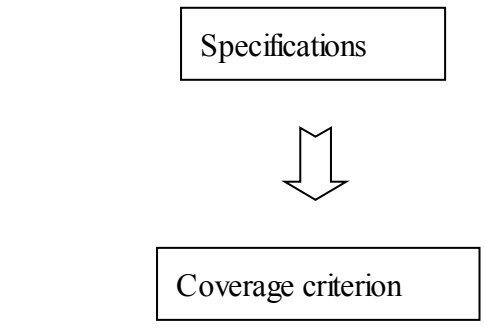
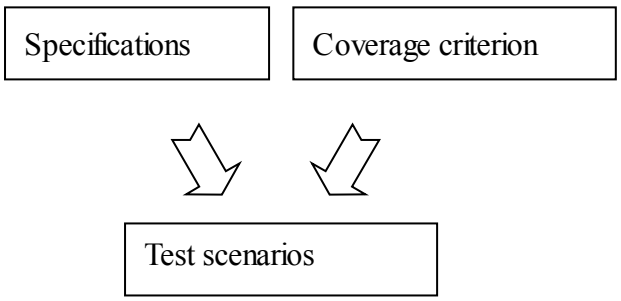
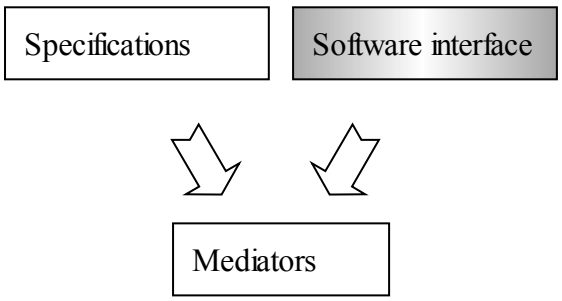
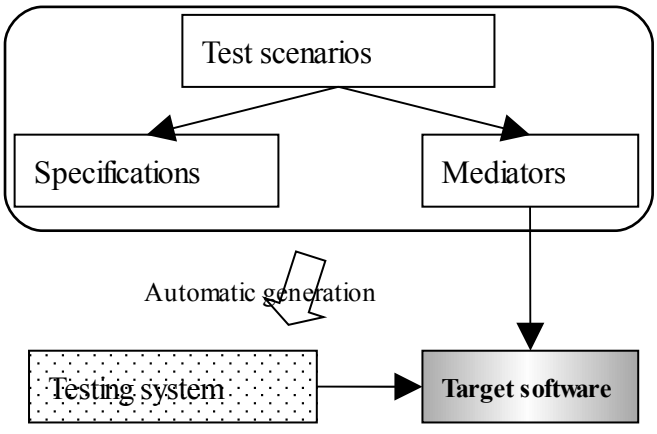
Quality control is an important problem which faces software engineers. Testing is the best-known and most widespread method of estimation and quality improvement. Functionality and complexity of modern software grow rapidly, while its life time increases permanently. Under such conditions labor-intensiveness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

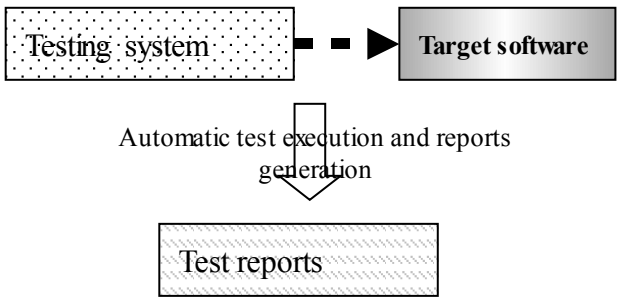
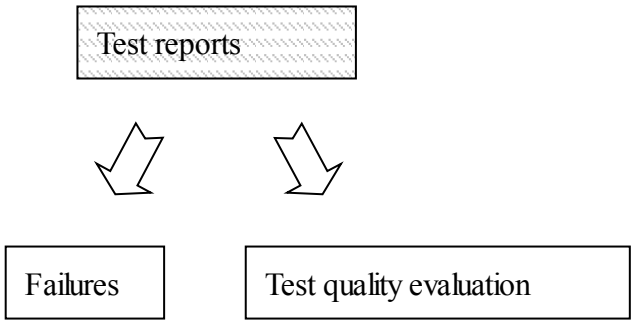
The main features of UniTESK technology are as follows:

- 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 which leaves out of account features of the 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 for a selected criterion of test coverage. This scenario aims to achieve maximum coverage according to a selected criterion. Widely used test scripts are analogous to test scenarios. However, UniTESK test scenarios give a possibility to improve greatly quality of testing under 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 requirements to testing quality 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 sy-tem. 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.	

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.	
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.

1.2 UniTESK implementation in CTESK

CTESK implements UnitTESK 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 be aware of 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, specification type library, 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 creat-ing, copying, comparing, and destroying data of these types. It also contains a set of predefined specification data types.

Generators of textual and graphical test reports generate easy-to-analyze representations of test execution trace.

2 User manual

There is a review of the SeC language usage features in this chapter.

- In "General information" section the differences between SeC and C is cited, the concepts and constructions which extension of C 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* is described. Also the concept of *allowable* and *specification* data types is introduced, the working with existing specification data types and rules for creation of new data types are considered in detail. The mechanism of *invariants* for data types and variables is described.
- In "*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 "*Test 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.

3 General information

SeC completely supports ANSI C standard. Additionally, *specification data types, types and variables with invariants, coverages* 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`.

4 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 *preconditions* 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 catchers*. Unlike *specification functions*, *coverage criteria* can not be specified for *deferred reactions*.

4.1 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);
```

SeC supports three constructions using for inclusion in specification function body : a precondition (which may be omitted), coverage criteria (which may be either in any or no amount) and a postcondition (which is necessar-ily 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 { ... }
}
```

In a general case, auxiliary code may be used within the body of a specification function between the described blocks.

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.Usually this code is using for definition of variables which are common for precondition, coverages and postcondition. Full description is in semantic requirements to specification functions.

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*, execut-ing of a testing interaction and synchronizing of model and implementation states in the *media-tor*, 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).

4.2 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 post-condition (which is necessarily single).

Precondition checks possibility of appearing of a reaction in a given state. Precondition is exe-cuted 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 { ... }
}
```

In a general case, extra code may be used in the body of deferred reaction between described blocks. 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. Usually this code is using for definition of variables which are common for precondition, coverages and postcondition. Full description is in semantic requirements to deferred reactions.

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

4.3 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 al-tered.

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 precondi-tion 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 opera-tor `@` 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;
  }
}
```

4.4 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)) …
```

4.5 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 specifica-tion is registered.

Deferred reaction postcondition describes constraints, which the results of interaction should sat-isfy 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 key-word (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.

```
/* List — library specification type*/
specification void f( List* l ) {
  int j;
  post {
    int i;
    Object* pre_item;
    for( i = 0, j = 0
       ; i < @size_List(l) /* allowed */
       ; i++, j++
       )
```

```
      {
        /* not allowed: i is not defined outside the postcondition */
        pre_item = @get_List(l, i);
        /* not allowed: j has the only one unknown value outside the
postcondition */
        pre_item = @get_List(l, j);
        pre_item = get_List(@l, j); /* allowed */
        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.

4.6 Data Types

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

1. Boolean type bool for presentation of logical expressions, and true and false constants.
2. Specification types, that integrate the types of C with basic operations with the data of the types: creating, copying, comparing, destroying.
3. 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.

Types which are allowed for the arguments and return values of specification functions, deffered reactions and mediator functions, for iteration variables and scenario functions of scenario state variables and also for global variables using in the above-mentioned cases are called SeC allowable types.

Special demands are maked for the allowable types. Many types of C-language doesn't provide this demands. For example, it's impossible to compute the array size using its pointer, correspondingly it's impossible to copy the type's value.Specification types are using for this constraints overcoming . Also specification types are necessary ones during use of test system support library CTESK (See also “CTESK test system support library”).

4.7 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:

1. Arithmetical types (`int, char, double,` ...).
2. Enumerated types (`enum`), the range of which, unlike C, is limited to their constants and may not contain an arbitrary integer value.
3. 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.
4. Non-typed pointer (`void`*). Values of the type are interpreted simply as an address of a certain memory cell.
5. Functional pointer. Values of the type are interpreted simply as an address of a certain function.
6. 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.
7. 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*.

4.8 Specification types

The acceptable data types are required to meet the demands which are not ensured by many types of the C language. It is impossible, for example, to calculate the size of array by pointer to it, thereafter, it is impossible to copy value of data type. Specification types are used for overcoming this constraints. CTESK test system support library use is also required specification types (See also CTESK test system support library).

Specification data type combines C data type with basic operations for dealing with data of this type: creating, copying, comparing, destroying.

Values of specification data types are always located in a dynamic memory and are available only by specification references – pointer of appropriate type, which is either null or point to allocated or initialized memory. It means that there can be no declaration of variables or parameters of the specification types (not pointers to them) and also no direct use of specification type in structures, unions or arrays determination.

If specification reference is omitted in a specification data type definition, it is given zero value automatically.

Specification references are dereferencing like pointers in C – whit the use of dereferencing operators * and ->. The result of reference dereferencing is a l-value data type of which is the same as the data type on the base of which specification data type was defined (namely with the base data type in specification data type definition) or with the type of specification structure field (See also Creating new specification types).

Zero references can't be dereferenced (lead to run-time error).

Specification references returned by function call without assigning returned references to variables can't be dereferenced (lead to memory leak or run-time error).

```
/* List – library specification type */
List* l = create_List(&type_Integer);
/* Integer – library specification type */

Integer* spec_i;
int i;

append_List(l,create_Integer(1));

i = *get_List(l,0); /* not allowed */
spec_i = get_List(l,0);
i = *spec_i; /* i is 1 */

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

IntPair* ip = create(&type_IntPair, 1, 1);
int ai = create(&type_IntPair, 1, 1)->a; /* not allowed */

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

Address arithmetic on references or their indexing is not allowed.
```

Comparison of addresses in references with C operators == and != is allowed.

Comparison of addresses in references returned by function call without assigning returned references to variables is not allowed (lead to memory leak).

```
List* l = create_List(&type_Integer);
Integer* spec_i
int i;

append_List(l,create_Integer(1));
spec_i = get_List(l,0);
if (get_List(l,0)!= NULL) /* not allowed */
if (spec_i != NULL)
i = *spec_i; /* i is 1 */
```

In SeC the built-in specification type Object is defined that is incomplete specification type. It is abstract basic data type for all specification data types. The reference type Object* is applied by analogy with void*. The reference to any specification type may be cast to the reference to Object and vice versa. If, in case of an inverse cast, reference type of value is not compatible with the data type to which the cast is performed, then the system behavior is not defined.

Data types of specification references can only be cast to pointers to void or Object types, 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 (about subtypes see also Type invariant).

Management of the memory, to which specification references refer, is automated through the mechanism of reference counting with tracking of cyclic relations. When specification references are used in the assignment statements, operations of passing references as the function arguments, returning a reference from the function, exit a reference from the visibility scope, reference counters vary automatically. The memory that has been allocated for the value of a specification data type is automatically deallocated as soon 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, as in such cases automatic altering of reference counters is not supported.

The functions that implement basic operations on the values of specification data types are located in the library of specification data types of CTESK (See also "Library of specification data types").

4.8.1 Creating of specification data type value

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

As the first parameter, the function receives a pointer to a specification data type descriptor. The descriptor of the specification type is a constant the name of which consist 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 function of initialization of specification type.

The function allocates memory for the value of the specification type, zeroizes it, invokes the type initialization function, passing values of the type initialization parameters in a list of the va_list* type to it 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 the specification representation of the C built-in type int and of the string specification type, respectively. When a reference of the Integer* type is created, an integer value,

which will be stored according to the reference, should be passed to the function create. When a reference of the `String*` type is created, a normal string of C must be passed.

The usage of function create can cause errors because of the usage of not typified argument list.... To except such errors it is recommended to create function `create_name_of_specifiction_type` for each specification data type and to call `create` out of it.

```
specification typedef struct {int a; int b;} IntPair ={};
IntPair* create_IntPair(int a, int b)
{
  return create(&type_IntPair, a, b);
}
```

If you use create out of function the name of which has `create_` prefix, the compiler gives out warning:

```
warning: call create() out of create_... function
```

4.8.2 Copying the specification type value

```
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`*.

4.8.3 Cloning of the specification type value

```
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.

4.8.4 Comparing the specification type values

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

When the values of the references passed are equal, the function returns a zero value. Otherwise, the

function returns a nonzero 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, not subtypes of the same type, and the type of one reference is not the subtype of the second reference's type (see also Type invariant), then the function returns a nonzero value. If one of the references is zero, and the other is not, a nonzero value will be returned. If both references are zero, then zero will be returned.

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

4.8.5 Comparing the specification type values to equality

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

The function returns either the true value, in case the values of the references passed are equival, 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 references are zero true will be returned.

```
SpecificationType* ref1 = create(&type_SpecificationType);
SpecificationType* ref2 = create(&type_SpecificationType);
...
if (equals(ref1,ref2)) {/* values are equival */
...
} else {/* values are not equival*/
...
}
```

4.8.6 String form of specification type value

```
String* toString(Object* ref)
```

The function returns the reference to the String type value - the specification form of a string type.

```
SpecificationType* ref = create(&type_SpecificationType);
String* str;
...
/* conversion *ref into string form*/
str = toString(ref);
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 the one hand, free cannot be invoked for the returned pointer, and on the other hand, the pointer may not be used after destroying the value of the passed reference.

4.8.7 Creating XML-form of specification type value

```
String *to_XML_MyType( Object *ref )
```

The function returns specification line which contain XML-form of *ref* parameter.

4.8.8 Creating new specification types

Specification types are declared using 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
, .to_XML = pointer_to _XML_conversion_function
, .enumerate = pointer_to_enumerate_function
, .destroy = pointer_to_destroy_function
};
```

The specification type must be declared or defined in each translation unit before use.

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

In 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, with elements of the above-defined types;
- 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. 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 is initialized. The list of 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*);
```

Using this pointer the function of copying the value of the given specification type is invoked from the library functions copy and clone. The function of copying specification type is invoked if the references passed to the function copy or clone are nonzero. 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*);
```

Using this field the function of comparing the values of the given specification type is invoked from the library functions compare and equals. The function of comparing specification type is called if the references passed to the functions compare or equals are nonzero 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). 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*);
```

Using the field the function of constructing a string presentation of the given specification type is invoked from the library function toString. The stringifying function is invoked if the reference transferred to toString is nonzero.

Field enumerate has the type Enumerate:

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

Using 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*);
```

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

If the basic type in the definition of a specification type is a SeC allowable type 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. If default functions implement all needed functionality for all basic operations an empty initialization is used in specification type definition.

```
specification typedef struct {int x; int y;} Point = {};
Point *pt2, *pt1 = create(&type_Point, 1, 2);
/*in result : pt1->x == 1, pt1->y == 2*/
```

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.

To create data structure with complex topology, e.g. defining recursive structures,it is recommended to use specification references only.

```
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 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, it will be more difficult to correct management of the dynamic memory.

```
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.

4.8.9 Default implementation of basic operations of specification types

If the basic type in the definition of a specification type is a SeC allowable type 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. If default functions implement all needed functionality for all basic operations an empty initialization is used in specification type definition.

4.8.10 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.

4.8.11 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 nonzero 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.

4.8.12 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.

4.8.13 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.

4.8.14 Function of creating XML-form by default

Function of creating XML-form by default returns XML-form of basic type value.

4.8.15 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.

4.8.16 Function deallocating resources by default

The function of deallocating resources by default:

- does not act provided that 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.

4.8.17 Definition of own functions for the specification type basics operations

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.

4.8.18 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* - references to the values of the library specification type Integer, which is a specification representation of the built-in int type of C. 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 has two parameters of initialization for such structure specification type: 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 - 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.

4.8.19 Function of copying specification type

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

The function has no return value. It 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, because the function of copying by default 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. 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.

4.8.20 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 nonzero value otherwise. A nonzero 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 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 lengths 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`.

4.8.21 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, which should contain a string presentation of the specification type, that corresponds to the value of 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 `IntegerSeq` is defined. Definition of the function is necessary, because function of stringifying by default 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.

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

4.8.22 Function of creating XML-form specification type

```
String* name_of_XML-form_creating_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, which should contain an XML-form of the specification type, passed within the single parameter of the function.

If the data type don't have the inner structure we needed in, then a conversion function should be written like `to_XML_Integer()`:

```
String *to_XML_MyType( MyType *mt )
{
  return to_XML_spec("MyType", to_string_MyType(mt));
}
```

If the inner structure of the data type is important, then a conversion function could be written like library type function Set (set):

```
String *to_XML_MyType( MyType *mt )
{
  String *res = format_String("<object kind=\"spec\" type=\"MyType\"
text=\"Header\">\n" [, ...]);
  foreach(Object obj: nested obects, which inherit Object)
  {
    res = concat_String( res, toXML( r(obj) ) );
  }
  foreach(data: nested obects with kind = Simple, Struct, Pointer,
Array)
  {
    res = concat_String( res, ts_to_XML_sectype(&typeDesc, &data) );
  }
  res = concat_String(res, create_String("</object>\n"));
  return res;
}
```

or like library type function Map (mapping):

```
String *to_XML_MyType( MyType *mt )
{
  String *res
    = format_String
      ( "<object kind=\"spec\" type=\"MyType\" text=\"Header\">\n
      [, ...]
      );

foreach( Object obj: nested obects, which inherit Object )
{
  res = concat_String
        ( res
        , format_String
          ( "<object kind=\"spec\" "
```

```
            "type=\" MyElement\" "
            "text=\"element header\">"
          [, ...]
          )
        );
    res = concat_String( res, toXML( r(obj) ) );
    res = concat_String( res, create_String("</object>\n"));
  }

  foreach( data: nested obects with kind = Simple, Struct, Pointer,
Array )
  {
    res = concat_String( res
                       , format_String
                         ( "<object kind=\"simple\" "
                           "type=\" MyType.Element\" "
                           "text=\"element\"/>"
                         [, ...]
                       )
  }
  res = concat_String(res, create_String("</object>\n"));
  return res;
}
```

Combining the templates of generation from the above-listed examples, one can create any treetype structure of objects.

4.8.23 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, because the function of enumerating references by default 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`.

The pointer to the function `enumerate_IntegerSeq` initializes the field *`enumerate`* in definition of the type `IntegerSeq`.

4.8.24 Function deallocating resources of specification type

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

The function has no return value. It has one parameter of the type void* that represents a pointer to the memory location, where data of the specification type is stored. The function must deallocate only additional memory, allocated within the function of initialization the given specifica-tion 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, because the function of deallocating resources by default 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.

4.9 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.

4.9.1 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.

4.9.2 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.

5 Coverages

Coverage criteria are used for estimation of achieved test complete.

Full testing of program requires checking of correctness of its behavior in all situations. The total number of situations is usually too large for full test. Coverage criterion is choosing to reduce the number of test situations. According to coverage criterion all test situations are divided into classes, called *coverage elements*.

Hypothesis of testing is put forward. Any error in a program, which becomes apparent in situation referred to the certain class, becomes apparent in any other situations from this class. That is, if hypothesis of testing is accepted, it is enough to check correctness of program behavior in one situation from each class for full testing. If the coverage criterion is selected appropriately then test hypothesis is correct for all errors. In this case reached the degree of completeness of testing corresponds to the percentage of covered elements for given coverage criterion.

In the software industry coverage criteria based on the structure of source code of program are used, in particular:

- coverage of operators;
- coverage of graph branches of the thread of execution;
- coverage of graph path of the thread of execution;
- coverage of conditions.

During black-box testing source code of program may be unavailable. In this case similar coverage criteria based on the specification structure are applied.

During program testing again given requirements set coverage criterion may be based on the definitions of situations used in the requirements. That is, for the requirement “if the **condition C** is fulfilled then **requirement R** must be fulfilled ”, the **condition C** defines coverage elements.

Such coverage may be sufficiently rough - It will be enough to check **requirement R** once in situation satisfying **condition C**. If it isn't enough coverage elements can be divided additionally and requirement is considered as tested fully only in case of its checking under each qualified condition.

In CTESK 2.8 coverages are realized as set of elements, attributes if which are name and optional text description. Each coverage element corresponds to the certain state of system under test.

Coverages are specified using CTESK 2.8 construction, in which information necessary for coverage identification and work with them and their elements concentrates. CTESK 2.8 allows to create coverages of different types and requires compliance with the certain rules for specifying of coverages of each type.

Basic operations of interaction with coverages elements are:

- access to an element of coverage (for example, the calculation of achieved element);
- compare element with another one (for example, achieved element with an expected one);

- entering of information to trace about coverage element achievement

5.1 Different types of coverages

In SeC coverage is defined by name, set of coverage elements and not obligatory *computation function* of current element. Coverage element can have identifier or text representation, or both of them. Using system state given by parameters and global variables, computation function computes coverage element to which this state applies.

Coverages are specified using SeC construction, in which information necessary for coverage identification and work with them and their elements concentrates. SeC allows to create coverages of different types and requires compliance with the certain rules for specifying of coverages of each type.

Type of coverages are differed the following way:

- By belonging to the specification functions:
  - global coverage isn't tied to the specification function, it is defined and declare on the compilation module level;
  - local coverage is tied to the certain specification function. Prototype and definition of the specification function contain declarations and definitions of the local coverages, respectively.
- By elements representation and computing method of achieved element:
  - If the author of the test immediate specifies the collection of the elements for coverage and marks on his own the achieved element then this coverage is a *enumerable* one.
  - If the author of the test immediate specifies the collection of the elements for coverage and defines the computation function of achieved element then this coverage is a *computable* one. Computation function is a indispensable part of such coverage.
    - If the author of the test defines some enum-type as the collection of the coverage elements then computation function is generated on basis of the structure of this type and the coverage is called enum-coverage.
- By the position in the hierarchy:
  - primary coverage, set of elements of which was created independently of other coverages;
  - derived coverage, set of elements of which is combination of elements of other coverages. Coverages, elements of which were used in creation of derived coverage, are called basic with respect to it.

Global coverages can be enumerable, computable and enum-coverages. Local coverages are always computable.

5.1.1 Enumerable coverage

Enumerable coverage is coverage which doesn't have elements computation function. Thereby, accessibility of the elements of the enumerable coverage is described by the author of the test. Enumerable coverage is specifies only by set of elements in possession of names and not obligatory string description.

```
// declaration of enumerable coverage
```

```
extern coverage ConstCovD =
{
  C1 = "First",
  C2, // string description of this elements on default: "C2"
  C3 = "Last"
};

// definition of the previously declared coverage
coverage ConstCovD;

// definition of not previously declared coverage
coverage ConstCovND =
{
  C1 = "ND1",
  C2 = "ND2",
  C3
};
```

It is possible to get an element of the enumeration coverage only by specification this element using coverage name and element name (possibly complex name for derived coverage), separated by a dot.

Explicit specification of coverage element. There is possibility to explicit specify necessary coverage element at any moment using coverage name and element name (possibly complex name for derived coverage) separated by a dot.

```
// drop to trace indicated element of enumeration coverage
trace ( Cov.C1 );

// drop to trace indicated element of product-coverage
trace ( ProdCov.C1.C2 );
```

5.1.2 Computable coverage

Computable coverage is used for realization of certain algorithms for computing achieved coverage element depending on some parameters. Computable coverage is coverage accessibility of the elements of which can be compute using algorithm defined by author of the test.

Computable coverage besides attributes general for all coverages – name and set of elements, has explicit function of computation of elements. Computation function returns coverage element and receives parameters, which can characterize state of the system under test or mark branch of functionality as argument.

```
// definition of not previously declared computable coverage
coverage AuxCoverage ( int i )
{
    /*
     * element name can be absent,
     * if it doesn't require access from program -
     * numerical value and description drop to trace.
     */
    if ( i <= 0 ) return { "Incomprehensible number" };
    if ( i > 3 ) return { MNOGO, "Very large number" };

    /*
     * element description also can be absent,
     * in this case it equals to name
```

```
     */
    return { MALO };
}
```

It is possible to get an element of the computable coverage by specification of necessary element (the same as for enumerable coverage) or by computation of achieved elements..

Explicit specification of coverage element. There is possibility to explicit specify necessary coverage element at any moment using coverage name and element name (possibly complex name for derived coverage) separated by a dot.

```
// drop to trace indicated element of enumeration coverage
trace ( Cov.C1 );

// drop to trace indicated element of product-coverage
trace ( ProdCov.C1.C2 );
```

Computation of achieved coverage element. Computation of coverage element is possible only for computable coverage.

```
// check of achieved element of computable coverage
if ( CalcCov.POZ == CalcCov (i) );
{ ... }
```

5.1.3 Enum-coverage

Enum-coverage are used for easy creation of coverages based on enumerable types.

Coverage can be defined on the basis of an enumerated type (`enum`). Elements of such coverage

correspond to enumeration type constants and computation function is determined automatically. This function has only on parameter of using enumeration type. Corresponding element of coverage returns for each value of enumeration type.

```
typedef enum { RED, GREEN, BLUE } Color;

/*
 * declaration of coverage based on enumerable type,
 * with computable function on default
 */
extern coverage enum ColorCoverage( Color color );

/*
 * definition of previously declared coverage
 * on basis of enumerable type
 */
coverage ColorCoverage;
```

5.1.4 Derived coverage

Derived coverages are used when reuse of elements and functions of coverages computation are needed.

Coverages, elements and computation function of which are given directly, are called *primary coverage.*

Coverages built on basic of other coverages which are basic in this context called *derived* coverage. Definition of derived coverage must describe method of its building on basis of the basic coverages;

primary or derived coverage can be basic coverage.

Coverage order is number of basic coverages, elements combination of which are elements of given coverage. Coverage order of any primary coverage is 1. Coverage order of derived coverage

is equal to sum of orders of all basic coverages.

For any coverage it's possible to built list of primary coverages on which derived coverage is based. Length of this list is equal to coverage order. This list is formed from list containing only given coverage in the following way: cyclically passing from the beginning to the end, substitute coverages for basic coverages (products become sequences of multiplied coverages). Eventually only primary coverages remain in list, number of which is equal to order of initial coverage.

There are two ways to create derived coverage:

- change of coverage domain (inheritance of elements set and computation functions);
- coverage product (combination of elements of basic coverages and computation functions).

Access to elements of derived coverage depends on its type.

Computable coverage is result of change of coverage domain and product of computable coverage.

Element which is part of it can be gotten by explicit indication of necessary element or by computation of achieved element by giving parameter to computation function. The product of an enumerable coverage on enumerable one is enumerable coverage. Element of such coverage can be gotten by only explicit indication of this element. Thus, access to the elements of derived coverage is not principle differ from access to the elements of the primary coverage.

Operation of getting of coverage element component is specific for derived coverages. This operation is available for coverages of order higher 1 – i.e. for product-coverage derived from them: then the element of given coverage is combination of elements of other coverages and it's possible to get its component. Element components of other coverages are forbidden to get.

5.1.5 Product-coverage

Product-coverages are used when sets of elements of several coverages are necessary to be combined. Elements of product-coverages are combination of elements of basic coverages, every of which is called coverage product element component. Coverage product element is considered as achieved one, if all its components are achieved simultaneously.

```
// definition of product-coverage for enumerable coverage
coverage ProdCovNF = ConstCovND * ConstCovD;
// definition of product-coverage for computable coverage
coverage ProdCovF (Color color, char ch, int in)
 = AlphaCoverage ( ch ) * ColorCoverage (color) * AuxCoverage ( in );
```

Operation of getting of coverage element component is available for coverages of order higher then one – i.e. for product-coverages or derived coverages: then element of given coverage is combination of elements of other coverages and it's possible to get its component. Components of elements of other coverages are forbidden to get.

Getting of coverage element component

```
// AlphaCoverage and AuxCoverage – computable coverages

// ColorCoverage - enum-coverage based on Color type

// definition of product-coverage of computable coverage
```

```
coverage ProdCovF (Color color, char ch, int in)
 = ColorCoverage (color) * AlphaCoverage ( ch ) * AuxCoverage ( in );

// drop to trace of component of coverage element
trace ( ProdCovF( i, (char) i, i )[1] );
```

5.1.6 Change of coverage domain

Change of coverage domain allows to use function of computation of basic coverage in derived coverage indicated only variable which will be parameter of this function.

Change of domain is applicable to only computable coverages; gotten derived coverage is computable also. Such derived coverage use function of computation of basic coverage specified expressions used as factual parameters. Parameter of derived coverage and variables visible for it can be used in these expressions

```
// basic computable coverage
coverage AlphaCoverage( char c )
{
  if ( 'a' <= c && c <= 'z' ) return LOWER;
  if ( 'A' <= c && c <= 'Z' ) return UPPER;
  /*
   * if execution flow reaches this point,
   * then coverage isn't complete.
   */
}

// variable, which will be parameter of new coverage
extern char charState;

// ( void ) said about availability of computation function for this
coverage
extern coverage StateCoverage ( void ) = AlphaCoverage( charState );
coverage StateCoverage;
```

5.1.7 Local coverage

It makes sense to define coverage as local, if it is used by singular specification function. When states of system under test, which identify coverage elements, achieved within the bounds of one specification function, it's convenient to make this coverage as local coverage. Local coverage is coverage which defined within some specification function; this function is called enclosing to this coverage.

Coverages defined within specification function are called local as contrasted to coverages, declared on global level. The structure of the specification function prototype includes local variables declaration, belonging to its, and the body of specification function includes declaration of this coverages.

The intensional part of local coverage declaration is coverage name. Local coverages must be computable, i.e. they can be initialized by computation function or coverages product. The set of parameters of computation function of local coverage coincide with set of parameter of specification function.

The scope of local coverages is block containing their definition. Local coverages names should not coincide with names of specification function arguments and its names, and also with names of

other local coverages.

```
specification foo (int a, int b, char h)
{
  coverage aCov
  {
    if (a > 0)
      return {a_poz, "a is pozitive"};
    if ((a < 0) && (b > 0))
      return {a_b_oppoz, "a and b are oppozite"};
    if ((a == 0) && (h > a))
      return {a_null_proof, "a is null, approved."};
  }
  post { ... }
}
```

Access to the elements of local coverages can be obtain from postcondition or computable function of local coverages defined after given one.

It is sufficient to specify the coverage name: it will indicate element achieved in the process of execution of function under the test. Also for local coverage the operation of getting coverage element is available, but computation of coverage element is forbidden.

```
coverage CharCmd (char c)
{
  if (c == 'e')
    return {exit, "exit command entered"};
  else if (c == 'h')
    return {help, "help command entered"};
  else
    return {nocmd, "no command or unknown"};
}

specification foo (int a, int b, char h)
{
  coverage hCov = CharCmd (h);
  coverage aCov
  {
    if (hCov == hCov.exit)
      return {abort, "Program terminated"};
    else if (hCov == hCov.help)
      return {pause, "Program standby"};
    else
    {
      if (a > 0)
        return {a_poz, "a is pozitive"};
      if ((a < 0) && (b > 0))
        return {a_b_oppoz, "a and b are oppozite"};
      if ((a == 0) && (h > a))
        return {a_null_proof, "a is null, approved."};
    }
  }
  post
  {
    if (aCov != aCov.a_null_proof)
    {
      //... }
    }
  }
}
```

5.2 Operations on coverages elements

Operations on coverages elements implement basic opportunities of control of testing and reflecting its results in the report.

Basic operations of coverage elements interaction are:

1. getting coverage element (for example, computation of achieved element);
2. comparison with other elements (for example, achieved element with expected one);
3. entering of information about coverage element achievement to report.

By combining these operation it's possible to control testing subject to achievement one element or the other and enter information about achieved elements to report.

It should be noted that only first operation from list depends on signs of the coverage type(computable function, belonging to specification function): getting, computation of coverage element.

Comparison and tracing operators require only rightly gotten coverages elements as arguments and don't depend on coverages declaration and definition.

The only especially specified SeC combination of basic operations is operation of coverage elements iteration, which becomes easier implementation of bypass of all necessary state of system under the test, i.e. some coverage elements.

5.2.1 Getting of coverage element

Access to coverage elements is used for entering information about achieved element and comparison elements against each other to testing report.

Different forms of access to element are provided for different types of global coverages.

1. *Explicit indication of coverage element*. There is opportunity to exact indicate necessary coverage element at any moment, using coverage name and element name (possibly complex name for derived coverage), separated by point.

```
// drop to trace indicated element of enumerable coverage
trace ( Cov.C1 );

// drop to trace indicated element of product-coverage
trace ( ProdCov.C1.C2 );
```

2. *Computation of achieved coverage element*. Computation of coverage element is possible only for computable coverages.

```
// checking of achieved element of computable coverage
if ( CalcCov.POZ == CalcCov (i) )
```

Access to elements of local coverage can be obtained from postcondition or from computable function of local coverages defined after given one. Suffice it to indicate coverage name: it will designate element achieved in the process of execution of function under the test.

Also operation of getting of coverage element is available, but computation of coverage element is forbidden.

```
coverage CharCmd (char c)
{
  if (c == 'e')
    return {exit, "exit command entered"};
  else if (c == 'h')
    return {help, "help command enterd"};
  else
    return {nocmd, "no command or unknown"};
}

specification foo (int a, int b, char h)
{
  coverage hCov = CharCmd (h);
  coverage aCov
  {
    if (hCov == hCov.exit)
      return {abort, "Program terminated"};
    else if (hCov == hCov.help)
      return {pause, "Program standby"};
    else
    {
      if (a > 0)
        return {a_poz, "a is pozitive"};
      if ((a < 0) && (b > 0))
        return {a_b_oppoz, "a and b are oppozite"};
      if ((a == 0) && (h > a))
        return {a_null_proof, "a is null, approved."};
    }
  }
  post
  {
    if (aCov != aCov.a_null_proof)
    {
      //...
    }
  }
}
```

5.2.2 Coverage element iteration

Coverage element iteration is contraction helping to bypass all branches of functionality describing by coverage. Parameters of iteration are name of invoked function and coverage name. Coverage elements bypass of which is not desirable for some reason can be sieved using filter.

`iterate coverage` operator of coverage elements iteration is similar with iterate operation of iteration: value of iteration variable is global with respect to scenario containing operator.

```
iterate coverage
        ( f.PointCoverage : f.PointCoverage[0] != IntCoverage.ZERO )
{
   //...

}
```

In contrast to `iterate` operator construction of control of iteration variable is absent: in coverages iteration accounting of processed coverage elements is realized by system under the test.

5.2.3 Entering of information about coverage element to report

Operations on coverage elements realize opportunities of control of testing and mapping its results to report.

Parameter of `trace` predefined function, which is tool of gathering of information about coverage elements, must be the coverage element specified by one of following methods:

- explicit indicated by user;
- getting as result of activities of function of coverage computation;
- getting as result of activities of operation of getting of coverage element.

```
// drop to trace of indicated element of enumerable coverage
trace ( Cov.C1 );

// drop to trace of indicated element of product-coverage
trace ( ProdCov.C1.C2 );
```

5.2.4 Storage of coverage elements in variables of CoverageElement type

SeC allows to store coverage elements in variables of `CoverageElement` special type.

Declaration of variable-element of coverage

```
coverage IntCoverage zero = IntCoverage( 0 );
```

After `coverage` keyword coverage name is specified, elements of which can contain variable declared in such manner

Declaration of variable-element of coverage can contain initializer: expression returning element of indicated coverage. In other cases, these variables conform to the rules of using of variables of C.

6 Mediators

Mediators are intended for binding specification to implementation of the system under test, or to specification of another level of abstraction.

Mediators do task of synchronization of specification data model state with the system under test state.

Conversion of model representation of a test action into implementation representation and conversion of implementation representation of reaction into model representation task matters only for stimulus mediators.

In SeC language mediators are implemented as mediator functions and catchers.

In mediator functions code blocks which are responsible for state synchronization and conversion implementation and model representation are mark out.

6.1 Mediator function

Mediator functions are implementation of mediators. Each mediator function binds specification function or deferred reaction to a part of implementation of the system under test or to its specification of another level of abstraction.

Mediator functions are marked by mediator for SeC keywords. An unique identifier—name of

the mediator function—must reside between these words. Each mediator function corresponds to a specification function or deferred reaction. Signature and access constraints of this function or reaction must be specified in declarations and definition of the mediator function.

Mediator function can contain:

1. Call 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).
2. *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*.
3. Auxiliary code before first or after the second special block. Auxiliary 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 bind to the specification function (or deferred reaction) using 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);
```

6.1.1 Call-block of mediator function

Call blocks of mediator specification functions are intended for implement behavior, described in corresponding specification functions, by means of impact on the system under test.

In call-block testing interaction with data conversion from model into implementation presentation and back is implemented.

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.

These instructions represent the body of the function, which has the same parameters and type of result as the appropriate *specification function.*

Mediator for function of selection of stack element

```
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));
  }
  //...
}
```

6.1.2 State-block of mediator function

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(stack_impl->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.

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*.

6.2 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.

7 Test scenarios

> Test scenarios define source data for tests building. Each test is a sequence of test actions, designed to solve some testing problem. Usually such problem is formulated as testing of a system under test behavior by performing test actions through a set of interface functions, until succeeding a given level of coverage in accordance with criteria of specification coverage.

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.

7.1 Creation of test scenario and its call parameters

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 as compared with dfsm allow 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 transition at all.

The ndfsm test engine does not intended for testing systems with deferred 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 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.

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

-t file_name

Send tracing to the file with specified name.

-nt

Disables tracing.

--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 appear (for ndfsm only).

-uend

Execute testing until complete despite errors.

--find-first-series-only, -ffso

Find only first success series.

--trace-format

Format of drop frame to trace.

--disabled-actions

Ignored scenario function list.

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`.

```
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: :

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

Three extra fields for scenarios with deferred reaction are mandatory.

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 model state is specified.

7.1.1 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 indicates whether initialization has been successful:

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

Usually function of initialization do next actions:

- initialization of the system under test;
- initialization of the specification data model;
- initialization of scenario data;
- setting up mediators for specification functions and reactions used in the given test scenario;

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);
  ...
}
```

7.1.2 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));
}
```

7.1.3 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);
}
```

7.1.4 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);
```

If model state presented as variable of nonspecification type or several variables, new specification type included all necessary data is need to be defined. .

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

7.1.5 Function of saving state stationarity

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

The function has no parameters and returns logical value that indicate whether the current model state is stationarity or not (The state is stationarity state if no deferred reactions are expected in this state).

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

Example:

```
List* modelList;
int   modelInt;
specification typedef struct {
```

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

7.1.6 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:

```
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;
}
```

7.2 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. Enumeration of possibilities of specification function argument is convenient to do use iteration statement.

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;
}
```

7.2.1 Iteration statement

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 (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.

As invocation of a scenario functions is executed in each new model state, then life time iteration variable goes beyond the scope of the execute time of scenario function in which it was declared. Each model state of the system must be correspond to own copy iteration variable to provide efficiency of increment. 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 a value of 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.

Iteration statement by coverage is specially intended to enumeration of all elements of any coverage.

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.

7.2.2 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.

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

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

8 Additional facilities

8.1 String and XML representations of non-specification types.

CTESK toolkit allows to define user functions of forming string and XML representation not only for specification types but also for any types of C, built using typedef construction, and also for types with invariants. Format and rules of forming XML representation values the same as specification types.

User functions of forming string and XML type representation specify in the following way.

Let there be declaration of type `T`

```
typedef struct { int x; int y } T;
```

For redefining of standard method of string representation construction is necessary to define function

```
String* to_String_T( T* );
```

For redefining of standard method of XML representation construction is necessary to define function

```
String* to_XML_T( T* );
```

It's important keep in mind that given functions will be used for tracing not only T type objects but also:

- "pointer to T" object type (any level):

```
specification void f( T *t );
```

- arrays, element of which are T or pointer to T:

```
specification void f( T t[10] );
```

- type-invariant objects, built on T type bases:

```
invariant typedef T InvT;

specification void f( InvT *it );
```

In addition, user function of non specification types representation will be invoked during tracing of type fields values, base for specification types:

```
specification typedef struct { String *str; T t; } S;
specification void f( S *s );
```

But user functions will **not** invoked for non invariant types, built using `typedef` construction "over" the types, for which these functions are not defined:

```
typedef T T2;
specification void f( T2 *t2 );
```

During tracing of t2 value of to_string_T and to_XML_T functions will **not** be used.

9 SeC Semantic

SeC *is* the extension of C programming language, for test development on the base of formal specifications.There are special constructions, introduced to describe requirements to a system under test and other components of the test system in a compact and convenient manner in SeC. This makes test development maximally comfortable, and allows reduction of training costs for specialists who are already be aware of C.

SeC language introduce the specification data types (see *Specification data types*), data types and global variables invariants, test scenario *(*see *Test scenario)* and tree kinds of functions: specification (see Specification functions, Deferred reactions), mediator(see *Mediator functions*) and scenario(see *Scenario functions*)functions. These types, invariants and functions are defined in specification files with **.sec** 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, 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`.

9.1 Specification

Specifications represent a formal description of requirements to a system under test in form of data invariants and specifications of a system under test's behavior. Specifications can use both data of a system under test and it's own specification data model for description of requirements.

9.1.1 Specification functions

Specification functions are intended for describing behavior of the system under test, experiencing external actions throw a part of its interface.

Specification function defines:

- constraints of access to data;
- precondition;
- local coverage criteria;
- postcondition;

Specification functions are declared and defined in specification files and are marked by `specification` SeC keyword. They can contain the following elements:

- Description of access constraints of the specification function to global variables and parameters.
- Description of local coverages (only inside the functions prototypes).
- Precondition, describing when behavior of the system under test is defined.
- Local coverage criteria, describing partition of the system under test behavior into functional branches, when interaction is fulfilling throw a part of interface described by the specification function.
- Postcondition, describing constraints of results of the system under test functioning, described by the specification function.
- Auxiliary SeC code outside precondition, coverage criteria, and postcondition.

Specification functions are called in the same way as the usual functions. In the invocation point of a specification function the following is performed in the specified order during a test run: check for data type invariants for expressions with `reads` or `updates` access, check for variables with `reads` or `updates` access, check for precondition, determining coverage elements for values of passed argument, call for mediator set for this specification function, check for immutability of expressions with `reads` access, check for data type invariants for expressions with `writes` or `updates` access, check for variables with `writes` or `updates` access, check for postcondition .

Syntax

(*declaration_specifiers*)?
"specification"
(*declaration_specifiers*)?
declarator
(*declaration*)*
compound_statement
;

Semantic rules

1. Names of specification functions belong to the same namespace as names of usual C functions.
2. Specification function must be defined exactly once within all translation units (`translation_unit`).
3. Not allowed to add any entity with the names which concur with the name of this specification function in declaration or specification of specification function
4. For every global variable and parameter (or their parts), used in a specification function, all declarations and declaration of the function must contain appropriate access constraints (`se_access_description`).
5. In all declarations and declaration of the function the access constraints must be the same.
6. Compound statement of a specification function must contain exactly one postcondition (`se_post_block_statement`) after coverage criteria and precondition blocks, if they present.
7. Compound statement of a specification function can contain no more than one precondition (`se_pre_block_statemen`) before coverage criteria (it they present) and postcondition blocks; after precondition must be followed by compound statement, or first coverage criterion, or postcondition.
8. Compound statement of a specification function can contain several local coverage criteria (`se_coverage_block_statement with initializer se_coverage_derivation_initializer or se_coverage_function_initializer`) following one after another, after precondition (if it present) and before postcondition; after last coverage criterion must be followed by compound statement or postcondition.
9. There can be no other constructions between `se_coverage_statement` constructions.
10. Constructions `se_pre_block_statement`, `se_coverage_statement` and `se_post_block_statement` may be putted in syntax nodes `block_item` (code blocks are limited by curly brackets).
11. With a glance of auxiliary code the specification function's body must have the following structure:

```
{
  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
}
```

12. Any one of auxiliary code's blocks may be absent.
13. On conditions that a couple auxiliary_code_2_1 and auxiliary_code_2_2 or auxiliary_code_3_1 and auxiliary_code_3_2 is absent, it is allowed don't writing the curly brackets which consist the missing couple of an auxiliary code.
14. Specification function must have no side effects:
    14.1. values of global variables and data, passed by reference, must not change;
    14.2. dynamic storage, allocated in a specification function, must be freed at the same nesting level (in the same compound statement):
        14.2.1. storage, allocated in precondition, coverage criteria, or postcondition, must be freed in the same block;
        14.2.2. storage, allocated in the beginning of specification function, must be freed in its ending after precondition, coverage criteria, postcondition, and compound statements containing coverage criteria and postcondition;
        14.2.3. storage, allocated in compound statement after precondition, must be freed in the end of this compound statement after coverage criteria, postcondition, and compound statement containing postcondition;
        14.2.4. storage, allocated in compound statement before postcondition, must be freed in the end of this compound statement after postcondition.

9.1.2 Deferred reactions

Deferred reactions are declared and defined in specification files as functions without parameters, marked by `reactions` SeC keyword, and returning pointers to specifications data types. They can contain the following elements:

- Description of access constraints of the deferred reaction to global variables.
- Precondition, describing when appearance of such reactions is possible.
- Postcondition, describing constraints of global variables that must be satisfied after occurrence of this deferred reaction.
- Auxiliary SeC code outside precondition and postcondition.

Syntax

```
( declaration_specifiers )?
"reaction"
( declaration_specifiers )?
declarator
( declaration )*
compound_statement
;
```

Semantic rules

1. Names of deferred reactions belong to the same namespace as names of usual C functions.
2. Deferred reactions must be defined exactly once within all translation units (translation_unit).
3. Deferred reactions must not contain parameters which type of return value may be only a pointer on specification type.
4. Not allowed to add any entity with the names which concur with the name of this specification function in declaration or specification of deferred reactions.
5. For every global variable and parameter (or their parts), used in reactions, all declarations and declaration of the function must contain appropriate access constraints (se_access_description).
6. In all declarations and declaration of the reactions the access constraints must be the same.
7. Compound statement of a deferred reaction must contain exactly one postcondition (se_post_block_statement) after precondition block, if it presents.
8. Compound statement of a deferred reaction can contain no more than one precondition (se_pre_block_statemen) before postcondition blocks; after precondition must be followed by compound statement or postcondition.
9. With a glance of auxiliary code the deferred reaction's body must have the following structure:

```
{
auxiliary_code_1_1
pre { ... }
  { auxiliary_code_2_1
    post { ... }
    auxiliary_code_2_2
  }
  auxiliary_code_1_2
}
```

10. Any one of auxiliary code's blocks may be absent.
11. On conditions that a couple auxiliary_code_2_1 and auxiliary_code_2_2 is absent, it is allowed don't writing the curly brackets which consist it.
12. Deferred reaction must have no side effects:
    - 12.1. values of global variables must not change;
    - 12.2. dynamic storage, allocated in a reaction, must be freed at the same nesting level (in the same compound statement):
      - 12.2.1. storage, allocated in precondition, or postcondition, must be freed in the same block;
      - 12.2.2. storage, allocated in the beginning of reaction, must be freed in its ending after precondition, postcondition, and compound statements containing postcondition;
      - 12.2.3. storage, allocated in compound statement after precondition, must be freed in the end of this compound statement after postcondition.

9.1.3 Access constraints

Access constraints are described after specification function or deferred reaction signature. Modifier `reads` is used to indicate read-only access, writes—write access, and `updates`—update access. Access modifier affects all identifiers listed after it until next modifier or beginning of function or reaction body. Aliases for constrained expressions can be defined in access constraints.

If an expression has update access (updates modifier), it possesses prevalue before post keyword, i.e. value before interaction with system under test, described by the given specification function, or value before occurrence of the given reaction. After post keyword the expression possesses postvalue, i.e. value after interaction with system under test or after occurrence of the reaction; prevalue after post keyword can be accessed via `@` SeC operator.

Syntax

```
se_access_description ::= se_access_specifier se_access
                              ( "," se_access )*
                          ;

se_access_specifier ::= "reads" | "writes" | "updates" ;

se_access ::= ( se_access_alias )? assignment_expr ;
```

Semantic rules

1. In access constraints for a function or reaction the same expression cannot be used several times with different access modifiers.
2. If an expression has write access (`writes` modifier), it cannot be used in function or reaction before `post` keyword.
3. Parameters of a function cannot be used as expressions with a write access(`writes` modifier).
4. If an expression has read-only access (`reads` modifier), its value is accessible everywhere inside function or reaction, and cannot change.

9.1.4 Aliases

Aliases are defined by a mere assignment in access constraints. In a specification function or deferred reaction aliases can be used in the same way as local variables.

Syntax

```
se_access_alias ::= <ID> "=" ;
```

Semantic rules

1. Alias identifier cannot coincide with parameters identifiers, local coverages names and must be unique within access constraints of specification function or deferred reaction.

9.1.5 Precondition

Precondition of a deferred reaction describes when appearance of such a reaction is possible. During test run precondition of deferred reaction is checked every time when the reaction occurs.

Violation of precondition indicates inconsistency between behavior of the system under test and its specification.

Precondition in SeC language is a set of instructions that syntactically and semantically equals to a function body with the same parameters as the specification function (deferred reactions have no parameters) and returning a `boole` an value. These instructions must be enclosed in curly braces and marked by `pre` keyword.

Syntax

```
se_pre_block_statement ::= "pre" compound_statement ;
```

Semantic rules

1. Specification function or deferred reaction can contain no more than one precondition, defined before coverage criteria (if they present) and postcondition.
2. Precondition can be omitted, that is equal to precondition, always returning true.
3. Precondition must have no side effects, i.e. it cannot change values of global variables and data, passed by reference.
4. Instructions in precondition must be syntactically and semantically equal to a function body with the same signature as the specification function or deferred reaction and returning a boolean value.
5. Precondition cannot use expressions with write access (i.e. defined in access constraints with writes specifier).

9.1.6 Postcondition

Postcondition of specification function is intended for description of constraints of results of the system under test functioning during interactions with it through a part of the interface, described by the specification function. During test run postcondition is checked every time after appropriate interaction with the system under test. Violation of postcondition indicates inconsistency between behavior of the system under test and its specification.

Postcondition of deferred reaction is intended for description of constraints of values of the reaction and global variables after occurrence of this reaction. Violation of postcondition after occurrence of the reaction and its mediator completion indicates inconsistency between behavior of the system under test and its specification.

Postcondition in SeC language is a set of instructions that syntactically and semantically equals to a function body with the same parameters as the specification function (deferred reactions have no parameters) and returning a boolean value. These instructions must be enclosed in curly braces and marked by post keyword.

Syntax

```
se_post_block_statement ::= "post" compound_statement ;
```

Semantic rules

1. Specification function or deferred reaction must have exactly one postcondition.
2. Postcondition must be defined after precondition (if it presents) and after the last local

coverage criterion (if coverage criteria present).

3. Instructions in postcondition must be syntactically and semantically equal to a function body with the same signature as the specification function or deferred reaction and returning a boolean value.
4. Postcondition must have no side effects, i.e. it cannot change values of global variables and data, passed by reference.
5. Postcondition and code after postcondition can use the following additional constructions:
   5.1. Preexpressions with @ operator (see Preexpressions);
   5.2. Value returned by the specification function (or value of the occurred reaction) is accessible via identifier of this function (or reaction), with the exception of void functions.
   5.3. Access to elements of local coverage which were computed early (If local coverages are defined in a specification function).
   5.4. Pseudovariable timestamp of TimeInterval type contains start and finish time marks for invocation of mediator of the specification function, or time marks supplied on deferred reaction registration.

9.1.7 Pre-expressions

Pre-expressions are used for specification of constraints of the system under test state before and after test action, or before and after occurrence of deferred reaction.

Syntax

```
"@" cast_expr ;


unary_expr ::=   postfix_expr
               | "++" unary_expr
               | "--" unary_expr
               | unary_operator cast_expr
               | "sizeof" unary_expr
               | "sizeof" "(" type_name ")"
               | gcc_extension_specifier cast_expr
           ;

unary_operator ::= "&" | "*" | "+" | "-" | "~" | "!" | "@" ;

cast_expr ::=   unary_expr
              | "(" type_name ")" cast_expr
              ;
```

Semantic rules

1. `@` operator can be used only after `post` keyword in the thread of execution.
2. Preexpression that is used in a postcondition, must be computable before this postcondition in the thread of execution.
3. Preexpression must not be embedded into each other.

9.2 Specification data types

Specification data types are defined by usual C-like typedef construction marked by `specification` SeC keyword. This construction can contain several declarations with initializers. If initializer is omitted, it declares new specification data type, otherwise it defines new specification data type with its own name.

Values of specification data types are located in a dynamic storage with automatic management. It maintains reference counters for each specification data type object and automatically destroys objects when there are no references to them.

Specification extension of C contains built-in incomplete specification data type Object. It is the base data type of all specification objects but no objects can be of this type. Type of reference to Object (i. e. Object*) is used in the same way as void*. Reference to any specification data type can be casted to reference to Object and vice versa. Note that behavior is not defined when object, referenced as Object*, is casted to incompatible data type.

Syntax

```
declaration ::= (  (( declaration_specifiers )?
                    "specification"
                    ( declaration_specifiers )?
                    "typedef"
                    ( declaration_specifiers )?
                   )
                 | (( declaration_specifiers )?
                    "typedef"
                    ( declaration_specifiers )?
                    "specification"
                    ( declaration_specifiers )?
                   )
                )
                ( init_declarator  ( "," init_declarator )* )?
                ";"
              ;
```

Semantic rules

1. Specification data types cannot be local.
2. Names of specification data types belong to the same namespace as typedef-names.
3. Definition of specification data type with a given name (declarator with initializer) can occur only once in all translation units (`translation_unit`).
4. Initializer in definition of specification data type must look like the following: `= { .<field(1)> = <expr>, .<field(2)> = <expr>, ... }`. Inside the curly braces there must a list (possibly empty) of constructions like `.<field(i)> = <expr>`, separated by commas. `<field(i)>` can possesses one of the values: `init`, copy, compare, `to_string`, `enumerate`, or `destroy`. `<expr>` must have appropriate data type according to the field specified:

```
/* Object initializer type
   ( object initialization    by pointer 'ref') */

   typedef void (*Init)( void* ref, va_list* arg_list );

         /* Object copier type (copy    <data> from 'src' to 'dst')
```

```
*/

typedef void (*Copy)( void* src, void* dst );

      /* Object comparer type (compare <data> of 'left' and
'right') */

typedef int (*Compare)( void* left, void* right );

/* Object stringifier type (make string representation of
<data>) */

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

typedef String* (*to_XML)( void* ref );

/* Subobjects enumerator type (enumerate objects belonging to
the given one) */

typedef void (*Enumerate)
  ( void* obj, void (*callback)( void* ref, void* par ),
    void* par
  );



/* Object destructor type (free resources allocated by object)
*/

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

Detailed information on functions to be used in an initializer can be found in "Library of specification data types" section.

5. If field initializer is omitted in a specification data type definition, appropriate default function is used.

   In default functions pointer to any data type (except specification, functional, and `void` data types) are treated as pointer to a single value of this data type. It means that all functions should be provided by the user when base data type is a pointer to several values (examples are array or character string).

6. The following data types are prohibited to be used as base types for specification data types:

   6.1. specification, functional, and incomplete;

   6.2. unions, arrays, and structures, containing above-listed data types.

   The following data types are prohibited too if one or more field initializers are omitted in a specification data type definition:

   6.3. unions and arrays of variable length;

   6.4. structures and arrays, containing all above-listed data typed

   6.5. pointers to unions, arrays and other structures.

7. If specification data type declaration contains keyword `invariant`, an invariant must be declared for this data type.

8. All base types are mentioned in the declarations and specification data type definition must be compatible against each other.

9. If at least one specification data type declaration contains `invariant` keyword, all declarations and definition must contain this keyword.
10. Specification data types can be used only by reference, like incomplete data types of C. The following are prohibited:
    10.1. declarations of variables of specification data types;
    10.2. unions, structures, and arrays, containing fields or elements of specification data types; and so on.

9.3 Invariants of types

Invariants of data types are declared by C-like typedef construction, marked by `invariant` SeC keyword. This construction defines new data types names, like usual typedef. But as distinct from typedef, range of defined data type not equals to range of the base data type, its a subrange of base data type range. Thus invariant typedef defines not a synonym for base data type expression, but a new data type with its own range.

Constraints of base data type range are described in a compound statement (compound_statement) that syntactically and semantically equals to a function body without side effects, returning a boolean value, and having one parameter of:

- defined subtype, if the base type is usual C data type,
- pointer to defined subtype, if the base type is specification data type.

Compound statement of invariant is marked by `invariant` modifier followed by the formal parameter of appropriate type in parentheses. Type of returning value is fixed and thus not indicated directly.

Invariant of data type can be checked for an expression of appropriate data type. Expression of invariant call consists of `invariant` keyword followed by an expression to check in parenthesis. Invariant call evaluates to true if a value of the expression to test satisfies invariant constraints, and to false otherwise.

Syntax

```
declaration ::=    (( declaration_specifiers )?
                      "invariant"
                      ( declaration_specifiers )?
                      "typedef"
                      ( declaration_specifiers )?
                    )
                  | (( declaration_specifiers )?
                      "typedef"
                      ( declaration_specifiers )?
                      "invariant"
                      ( declaration_specifiers )?
                    )
                  ( init_declarator  ( "," init_declarator )* )?
                  ";"
               ;

"invariant" "(" parameter_declaration ")" compound_statement

"invariant" "(" assignment_expr ")"
```

Semantic rules

1. Data type invariant cannot be defined for a local data type.
2. Data type must be defined by typedef construction before definition or call of this data type invariant, and its declaration specifiers (declaration_specifiers) must include invariant SeC specifier (se_declaration_specifiers).
3. If at least one declaration of data type in a translation unit (translation_unit) contains invariant SeC specifier (se_declaration_specifiers), all declarations and definition of this

data type in all translation units must contains invariant specifier, and definition of appropriate data type invariant must occurs once in all translation units (translation_unit).

4. Specification of this type invariant must occurs once in all translation units (translation_unit) which are gathered in one system.
5. During the type with invariant definition using of functional types and void as a base data type are prohibited.

If base data type in definition of data type with invariant is a specification type, parameter in invariant definition is declared as pointer to defined data type. User can be sure that inside invariant body this pointer is not NULL. For invariants of other data types (including pointer to specification type) parameter in invariant definition is declared as defined data type.

6. If in the capacity of base data type with invariant is used a specification type,then the parameter in the expression of invariant call must have a type which conforms a specification reference. Otherwise there is the expression of the type which invariant is checking in the expression of type invariant call in parenthesis.
7. Data type invariant call expression must contains in parenthesis an l-value expression .

9.4 Variable invariant

If a global variable cannot possess all values of its data type range, the range should be constrained by variable invariant. For that all declarations of such variables must be marked by invariant SeC keyword to indicate that the declared variables have constrained range.

Range constraints are described in a compound statement (compound_statement) that syntactically and semantically equals to a function body without side effects, without parameters, and returning a boolean value.

Compound statement of invariant is marked by invariant modifier followed by the variable identifier in parentheses. Type of returning value is fixed and thus not indicated directly.

Invariant of a variable can be checked for this variable value. If a variable has a type with a determined invariant, type invariant will be checked before the checking of the variable invariant .

Expression of invariant call consists of invariant keyword followed by variable name in parenthesis. Invariant call evaluates to true if a value of the variable satisfies invariant constraints, and to false otherwise.

Syntax

```
( declaration_specifiers )? "invariant" ( declaration_specifiers )?
( init_declarator  ( "," init_declarator )* )? ";"
;

"invariant" "(" <ID> ")" compound_statement;
"invariant" "(" <ID> ")"
```

Sematic rules

1. Variable invariants can be defined only for global variables.
2. Variable must be declared before definition or call of this variable invariant, and its declaration specifiers (declaration_specifiers) must include invariant SeC specifier (se_declaration_specifier).
3. If definition or at least one declaration of variable contains invariant SeC specifier (se_declaration_specifier), all declarations and definition of this variable in all translation units (translation_unit) must contains invariant specifier, and definition of this variable invariant must occurs once in all translation units.
4. Variable invariant definition and call expression must contain in parenthesis the name of the appropriate variable as the only parameter.
5. Invariant body must syntactically and semantically be equal to a function body without side effects, without parameters, and returning a boolean value.

9.5 Test scenario

In SeC language test scenario is defined in the same way as global variable which specifiers contain `scenario` keyword. Type of test scenario specifies test engine and is indicated by an identifier, defined in `typedef` construction. The following corresponds to each test engine:

- data type of information, necessary for test building, which is the base type in `typedef` construction that defines test scenario type,
- function to run the test built by this test engine.

Test scenario is initialized in its definition by a value, appropriate for the given test engine.

Test is run by a function call construction, where the name of a function is the name of the test scenario, and parameters are semantically equal to parameters of standard function `main(int argc, char** argv)`.Test run construction evaluates to `true`, if the test completed correctly and no errors were found during testing, and to `false` otherwise.

Syntax

```
( decl_specifiers )?
"scenario"
( decl_specifiers )?
( init_declarator  ( "," init_declarator )* )? ";"
;
```

Semantic rules

1. Names of test scenarios belong to the same namespace as names of usual global variable of C language.
2. Test scenario must be defines exactly once within all translation units (translation_unit).
3. Test scenario cannot be defined locally.
4. Type of test scenario must be specified by an identifier defined by typedef construction.
5. Identifier of test scenario type is a name of a test engine.
6. Initializer data type must be compatible with test scenario type.
7. If scenario type is a structure, it is possible to use dereferencing syntax of C99 standard in spite of actually used C standard.
8. Test scenario call looks like a call of a function with two parameters of int and char** types respectively. Second parameter must point to an array of character strings, terminated by zero character; the last element of this array (and only last) must be a null pointer. First parameter must be equal to size of the array without last element.
9. Test scenario call evaluates to boolean value.
10. Full definition of a test engine requires definition of a function to run tests built by the engine. This function must have the following signature.

```
bool start_<test_engine>( int argc
                        , char** argv
                        , <test_engine>* td
                        )
```

9.6 Scenario functions

In SeC language scenario function is specified as a function without parameters, returning a boolean value, and marked by scenario keyword. Scenario functions can perform additional checks on basis of results of system under test functions execution. Scenario function must evaluate to true, if the system under test behaves correctly, and to false otherwise. Test system automatically takes into account postcondition checks of executed specification functions or occurred deferred reactions, so scenario functions should not consider them.

Syntax

```
( decl_specifiers )?
"scenario"
( decl_specifiers )?
declarator
( declaration )*
compound_statement ;
```

Semantic rules

1. Names of scenario functions belong to the same namespace as names of usual functions of C language.
2. Scenario function must be defined exactly once within all translation units (`translation_unit`).
3. Scenario functions must have no parameters and must return a boolean value.
4. Scenario functions cannot be invoked directly.
5. Scenario functions can contain iteration statement, iteration statement by coverages, state variables.

9.6.1 Iteration statement

Two types of iteration statements can be used in scenario functions: the usual iterator and the iteration statement by coverage elements. The description of the latter strongly connects with terminology of coverages, therefore it is taken out to the relevant part.

Iteration statement starts with `iterate` keyword followed by the following list in parenthesis, separated by semicolon:

- Declaration and initialization of iteration variable;
- Controlling expression;
- Iteration expression;
- Filter expression;

All parts except first are not mandatory and can be omitted. Note that filter expression without iteration expression can lead to infinite looping. Declaration are finished by the iteration body.

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;
  ...
}
```

Iteration variable are not local variables, they are a kind of state variables. Their values are remembered in a special data structure, associated with the current generalized model state. These values become accessible as the model falls within the same generalized state.

Syntax

```
se_iteration_statement ::= "iterate"
                           "("
                           declaration
                           ( expression )?
                           ";"
                           ( expression )?
                           ";"
                           ( expression )?
                           ")"
                           statement
                       ;
```

Semantic rules

1. Iteration statements can be used only in scenario functions.
2. Controlling expression and filter expression must have bool data type, if they present.
3. One and only one iteration variable can be declared in the start declaration.
4. Iteration variable cannot be of incomplete or local data type and must be initialized.

9.6.2 State variables

State variables are intended to store data, associated with generalized model state. Values of such variables become accessible as the model falls within the same generalized state. Declaration of a state variable starts with `stable` modifier followed by declaration of local variables. The following code:

```
operator_1;

stable int i = 1;

operator_2;
```

equals to the following:

```
operator_1;

iterate(int i = 1; false ;;)
{
    operator_2;
}
```

Syntax

```
( declaration_specifiers )?
```

```
"stable"
( declaration_specifiers )?
( init_declarator  ( "," init_declarator )* )? ";" ;
```

Semantic rules

1. State variables can be used only in scenario functions.
2. State variables cannot be of incomplete or local data type and must be initialized.

9.7 Coverages

Complete information about a coverage (accessible within the bounds of the test) can be broken up:

1. Name of a coverage
2. Elements of a coverage
3. For computable coverages - function of coverage's element computation.

Specification of these coverage's characteristics can take place inside the compilation module in accordance with the rules: using the different kinds of specifications and declarations. Inside the compilation module the full information about each coverage must be presented for the correct work of operation of access to the coverage and his elements.

Each coverage can belong to the specification function or to be independent. That way, coverages are divided into local and global ones:

- Global coverages aren't attached to the specification functions. Their visibility scope is the compilation module. They are declared and specified on a global basis.
- Each local coverage is attached to the specification function; they are considered the specified on this function's layer and they have the visibility scope which is a interval between own specification and the end of function's body. Specifications and declarations of local coverages are contained in the prototype and determining a specification function;

Also the coverages are divided by means of setting elements and the method of computing an achieved element:

- If the author of the test immediate specifies the collection of the elements for coverage and marks on his own the achieved element then this coverage is a *enumerable* one.
- If the author of the test immediate specifies the collection of the elements for coverage and defines the function of computation the achieved element then this coverage is a *computable* one. Computation function is a indispensable part of such coverage.
- If the author of the test defines some enum-type in the capacity of the collection of the coverage's elements then Computation function is generated on basis of this type's structure and the coverage is called *enum-coverage*.

With the operations on the coverages' elements capabilities of using information about coverage's elements are realized for controlling of the test's course and recording data in the report.

9.8 Declarations and specifications of coverages

There are five kinds of coverages declarations and specifications:

1. shortcut declaration;
2. full declarations;
3. declarations of the primary computable coverages;
4. specification of the primary computable coverages;
5. shortcut specification.

The following kinds of constructions are the coverages specifications:

- full declaration, which doesn't contain `extern`**;**
- specification of the primary computable coverages;
- shortcut specification.

All other kinds of constructions are the coverages declarations. Declarations and specifications of the same coverage in the context of one compilation module must follow one be one in such a way that a quantity of known information about a coverage increases strictly with each declaration (each coverage has only one specification).References to the semantic rules of location for the concrete declaration type can be found in the end of the rule set which are general for all declarations and specifications.

Syntax

```
se_coverage_declaration ::= ( "extern" )? "coverage" ( "enum" )? <ID>
                                ( "(" ( parameter_type_list )? ")" )?
                                (   se_coverage_initializer
                                  | ";"
                                )
                            ;

se_coverage_initializer ::=   se_coverage_elements_initializer
                            | se_coverage_derivation_initializer
                            | se_coverage_function_initializer
                            ;

statement ::=   ...
              | se_coverage_statement
          ;

se_coverage_statement ::= se_coverage_declaration ;
```

Semantic rules

1. General requirements to coverages declarations and specifications.
   1.1. The names of global coverages enter into the file visibility scope.
   1.2. The names of local coverages have the visibility scope, which is limited by specification function much as the visibility scope of structure fields is limited by the structure.
   1.3. For every coverage within the one translation_unit may be no more then one

declaration or specification for every kind. (for example, it can't be two complete declarations for the same coverage).

1.4. Names of coverages elements have the visibility scope,which is limited the coverage much as he visibility scope of structure fields is limited by the structure.

1.5. As a whole the dependent files which compose one program must be exactly one specification for every declared coverage.

2. se_coverage_declaration requirements.

2.1. First construction identifier se_coverage_declaration sets the coverage's name.

2.2. Construction se_coverage_declaration can be nested direct in

2.2.1. translation_unit;

2.2.2. se_coverage_statement.

3. `se_coverage_declaration` of global coverages requirements:

3.1. If there isn't `extern`, then *`se_coverage_declaration`* is a coverage specification always.

3.2. If declaration or specification of global coverage has parentheses then set a computable or enum- coverage in it.

3.3. If `se_coverage_declaration`, which set a global coverage has enum, then enum- coverage is specified.

4. Requirements to `se_coverage_declaration` of local coverages:

4.1. Construction `se_coverage_statement` can be present only at the definition of specification function's body between pre- and postcondition.

4.2. extern specificator is forbidden.

4.3. Enum is forbidden (can't be local coverages).

4.4. Parentheses are forbidden (list of parameters for local coverages is predeclared by embedding specification function).

4.5. Only such kinds of initializers are permitted: `se_coverage_derivation_initializer` or `se_coverage_function_initializer`.

4.6. `se_coverage_declaration` which nested in s `se_coverage_statement` always must be a coverage specification.

5. General requirements to `se_local_coverage_description`.

Syntax

```
direct_declarator ::=   // ...
                      | direct_declarator
                        "("
                        parameter_type_list
                        ")"
                        ( se_access_description )*
                        ( se_local_coverage_description )*
                      | direct_declarator
                        "("
                        ( <ID> ( "," <ID> )* )?
                        ")"
```

```
                        ( se_access_description )*
                        ( se_local_coverage_description )*
                    ;

se_local_coverage_description ::= "coverage"
                                    se_local_coverage
                                    (
                                      ","
                                      se_local_coverage
                                    )*
                                ;

se_local_coverage ::= <ID>
                      (    se_coverage_elements_initializer
                         | se_coverage_derivation_initializer
                      )?
```

5.1. Identifier in `se_local_coverage` sets a name of local coverage.

5.2. Construction `se_local_coverage_description` can be present at `external_declaration` only and if only embedding `external_declaration` is a prototype of specification function.

5.3. If `translation_unit` has some prototypes of the same specification function, then only one of them can contain the local coverage description.

5.4. `se_local_coverage` is a coverage declaration always.

6. General requirements to the global computable coverage (including enum-coverage)

6.1. Types and names of the coverage parameters must satisfy the requirements to the types and names of C-function declarations or specifications parameters depending on whether `se_coverage_declaration` is a coverage declarations or specifications.

6.2. Parameters' names must not coincide with the name of declarations or specifications coverage.

6.3. Under the `translation_unit` lists of parameters all declarations and specifications of the same global computable coverage(including enum-coverage) must be equal. That is must coincide:

6.3.1. quantity of parameters;

6.3.2. types of parameters;

6.3.3. names of parameters.

7. General requirements to local coverages.

7.1. Local coverage is a coverage with the function of elements computation. The collection of coverage parameters coincide with the collection of embedding specification function parameters.

7.2. Names of all local coverages which are declared and specified for one specification function must be different.

7.3. Names of local coverages mustn't coincide with:

7.3.1. name of embedding specification function;

7.3.2. names of its parameters;

7.3.3. alias from the access constraints.

7.4. More precise to foregoing requirements: under the definition of specification function local coverages are parts of visibility scope corresponding to the block which contains the coverage definitions.

7.5. For each local coverage which is announced in local coverage description the definition of the same specification function must contain the coverage definition with the same name.

7.6. There are local coverages must be defined in the same way they were defined in local coverages description of prototypes this specification function in the description of a specification function.

9.8.1 Shortcut declaration

Shortcut coverage allows to declare the coverage having set only its name.

Shortcut definition of global coverage is used for using in one file of a compilation module the coverage's definition which is in the another file. Shortcut specifications of local coverage must be parts of a prototype of an embedding specification function. A coverage can't be defined in a shortcut declaration (initializer `is invalid`).

Similarly to declarations of variables in C language shortcut declaration is specified by extern keyword for the global coverages; also for the global coverages enum keyword is invalid. These constrains are set by requirements to `se_local_coverage_description for the local coverages.`

Shortcut declaration of a global coverage.

```
extern coverage C( int x, int y );
```

Shortcut declaration of a local coverage.

```
specification void f( int x ) coverage C;
```

Semantic rules

1. No one declaration or specification of the same coverage mustn't precede the shortcut declaration of the coverage.
2. The shortcut declaration of the coverage can precede any other form of the same coverage declaration or specification except that shortcut declaration.

9.8.2 Full declaration

Full declaration is destined for specify all necessary information about a coverage – name, elements and probably computation function.

Full declaration is the syntactical structure giving all necessary information about the described coverage.

1. If the full declaration hasn't extern specifier then full declaration is a coverage's specification .
2. If the full declaration of a local coverage is present in specification function's definition it is a specification, if it is present in prototype then it's declaration .

Full declaration are allowable one for following kind of coverages:

1. enumerable coverage;
2. derived coverage;
3. enum- coverage's.

Semantic rules

1. Before full coverage declaration The translation_unit text can contain only shortcut declaration of the same coverage.
2. If the full declaration of the global coverage hasn't extern,then this declaration is a coverage's specif.

9.8.3 Full declaration of enumerable coverage

The full declaration of an enumerable coverage sets its name and also the names and the textual description of its elements.

The full declaration of a coverage is governed by the specification and declaration syntax.

Since enumerable coverages can be only the global ones full specification of an enumerable coverage is set by the construction `se_coverage_declaration` only.

The construction `se_coverage_declaration` is a full declaration of an enumerable coverage on conditions that:

1. There isn't enum;
2. There aren't parentheses;
3. There is an initializer in the form of `se_coverage_elements_initializer`.

Full declaration of enumerable coverage

```
extern coverage C
  = { C1 = "first element", C2, C3 = "third element" };
```

Syntax

```
se_coverage_elements_initializer ::= "="
                                     "{"
                                     se_coverage_element_declaration
                                     ( ","
se_coverage_element_declaration )*
                                     "}"
                                     ";"
                                   ;
se_coverage_element_declaration ::= <ID>
                                    ( "=" <STRING_LITERAL> )?
                                  ;
```

Optional string literals set the textual presentation of the corresponded elements. If string literal is omitted, identifier of coverage element is used as textual representation.

Semantic rules

1. The identifier in `se_coverage_element_declaration` set the names of coverage's elements.

2. Names of coverage's elements can be different.

9.8.4 Full declaration of enum – coverage

Full declaration of enum – coverage sets primary coverage is constructed on basis of the enumerable type with the computation function on default.

Construction se_coverage_declaration is a full declaration of an enum- coverage on conditions that:

1. There is enum;
2. There are parentheses;
3. There isn't an initializer.

Full declaration of enum – coverage

```
extern coverage enum ColorsCoverage( enum Colors col );
```

Semantic rules

1. The coverage must have only one parameter whose type is a enumerable one.
2. The names of coverage's elements coincide with the names of this enumerable type's constants.

9.8.5 Full declaration of derived coverage

Full declarations of derived coverages set derived enumerable and computable coverages.

Construction `se_coverage_declaration` is a full declaration of global derived coverage on conditions that:

1. There isn't enum.
2. There is an initializer in the form of `se_coverage_derivation_initializer`.

Full declaration of a global product-coverage

```
extern coverage PointCoverage( int x, int y )
    = IntCoverage( x ) * IntCoverage( y );
```

Construction `se_local_coverage` or `se_coverage_statement` is a full specification of local derived coverage on conditions that there is an initializer of the form `se_coverage_derivation_initializer`.

Full declaration of local coverage by se_coverage_statement instruction

```
// declaration

specification void f( int x, int y )

coverage PointCoverage = IntCoverage( x ) * IntCoverage( y );
```

Full declaration of a local coverage by se_local_coverage instruction

```
specification void f( int x, int y )
{
   pre{ ... };
   coverage PointCoverage( int x, int y )
          = IntCoverage( x ) * IntCoverage( y );
   post{ ... };
}
```

Syntax

```
se_coverage_derivation_initializer ::= "="
                        se_base_coverage
                        ( "*" se_base_coverage )*
                        ( se_coverage_filter )?
                    ;

se_base_coverage ::= se_coverage_name
                        ( "("
                          ( assignment_expr ( "," assignment_expr )* )?
                          ")"
                        )?
                    ;

se_coverage_filter ::= ":" expression ;
```

All coverages are mentioned in `se_base_coverage` are called base coverages of current declaration (specification) coverage. The initializer of the current type defines how the element of declaration (specification) coverage is built on basis of the base coverage's elements.

Semantic rules

1. If a coverage is the coverage with the function of computation elements then the coverages which are specified in the list `se_base_coverage` must be coverages with the computation function.(Otherwise the specification coverage will be noncomputable one).
2. In a different way none of coverages are specified in the `se_base_coverage` list musn't be a computable coverage.
3. `se_base_coverage` mustn't point at the coverage about which only the name is known (there is nothing above the compilation module text except a shortcut definition)
4. If `se_coverage_derivation_initializer` is present at a local coverage declaration or specification and `se_coverage_name` from `se_base_coverage` points at a local coverage which is refer to the same function like the declaration (specification) coverage, then `se_base_coverage` mustn't have parentheses. (`se_coverage_name` is interpret as an element of local coverage wich is computed with the same values of specification function's parameters).
5. Otherwise if `se_coverage_name` from `se_base_coverage` points at computable coverage, then
   5.1. `se_base_coverage` must have arguments of the call.
   5.2. `se_base_coverage` subordinates to the rules of coverage's element computation.
6. `se_coverage_name` from `se_base_coverage` mustn't poin at declaration

(specification) coverage. (Thereby when the coverage depends on oneself the situation is forbidden)

7. If `se_coverage_derivation_initializer` contains exactly one `se_base_coverage` and a primary coverage is used in `se_base_coverage` then the derived coverage has the same elements like the primary based coverage.

9.8.6 Primary computable coverage declaration

Primary computable coverage declaration sets:

1. coverage name;
2. types and names of a computation function;
3. names and textual presentations of the elements.

Global coverages: se_coverage_declaration is a global primary computable coverage declaration on conditions that:

1. There is extern.
2. There are parentheses.
3. There is an initializer in the form of se_coverage_elements_initializer.

Declaration of global primary computable coverage

```
extern coverage IntCoverage( int x )
       = { NEGATIVE = "negative"
         , ZERO = "zero"
         , POSITIVE = "positive"
         };
```

Local coverages: `se_local_coverage` is a local primary coverage declaration if an initializer in the form of `se_coverage_elements_initializer` is.

Declaration of local primary computable coverage

```
specification void f( int x )
      coverage ArgCoverage = { NEGATIVE = "negative"
                             , ZERO = "zero"
                             , POSITIVE = "positive"
                             };
```

Semantic rules

1. Before the primary computable coverage declaration the text translation_unit can contain only the shortcut declaration of the same coverage.
2. All aforesaid requirements must be executed which cover to `se_coverage_elements_initializer`.

9.8.7 Primary computable coverage definition

A primary computable coverage definition sets:

1. coverage name;
2. names and textual presentations of the elements;

3. algorithm for elements calculation.

Definition of global primary computable coverage

```
coverage IntCoverage( int x )
{
   if( x < 0 )
      return { NEGATIVE, "negative" };
   else
   if( x == 0 )
      return { ZERO, "zero" };
   else
      return { POSITIVE, "positive" };
}
```

Local coverages: `se_coverage_declaration` is a local primary coverage definition if an initializer in the form of `se_coverage_function_initializer`.

Definition of local primary computable coverage

```
specification void sqrt( int x )
{
   pre{ ... }
   coverage ArgCoverage
   {
      if( x < 0 )
         return;
      else
      if( x == 0 )
         return { ZERO, "zero" };
      else
         return { POSITIVE, "positive" };
   }
   post{ ... }
}
```

Semantic rules

1. Global coverages: `se_coverage_declaration` is a global primary computable coverage definition on conditions that:
   1.1. There isn't extern;
   1.2. There isn't enum;
   1.3. There are parentheses.
   1.4. There is an initializer in the form of `se_coverage_function_initializer`.
2. Before the primary computable coverage specification the text `translation_unit` can contain:
   2.1. primary computable coverage declaration;
   2.2. shortcut declaration of this coverage.
3. Initializer se_coverage_function_initializer.

Syntax

```
se_coverage_function_initializer ::= compound_statement ;

se_return_expression ::=    expression
                          | se_coverage_element_return_expression
                       ;

se_coverage_element_return_expression ::= "{"
                       ( <ID> ( "," <STRING_LITERAL> )?
                        |<STRING_LITERAL>
                       )
                       "}"
```

3.1. Only following forms of return construction are acceptable in the `compound_statement`:

3.1.1. the expression is absence.It means that this branch of agency the computation function doesn't define an element.

3.1.2. `se_return_expression` is of the form `se_coverage_element_return_expression`.

3.2. Even though one “nonempty” return construction must be inside `compound_statement`.

3.3. If present coverage declaration were above the translation_unit text then

3.3.1. `se_coverage_element_return_expression` must contain only the element's identifier (thereby not allowed to redefine the elements string representation);

3.3.2. This identifier sets the name of a coverage's element.

3.3.3. For each declaration of coverage element the definition must contain `se_coverage_element_return_expression` which defines the element with the same name.

3.3.4. For each define identifier from `se_coverage_element_return_expression` the definition must contain the coverage's element with the same name.

3.4. Otherwise if the first `se_coverage_element_return_expression` element is:

3.4.1. identifier, then:

3.4.1.1. it sets the name of coverage's element. If a string literal follows after the identifier, then it set this element's textual representation.

3.4.1.2. Only one `se_coverage_element_return_expression` among all the `se_coverage_element_return_expression` from `compound_statement` which define one and the same coverage's element can contain this element's textual representation;

3.4.1.3. If a string literal is in none of the `se_coverage_element_return_expression` which define one and the same coverage's element then this coverage's element textual representation coincides with its name.

3.4.2. String literal, then define nameless coverage's element. All nameless

coverage's elements are considered different without distinction of coincidence or noncoincidence of them descriptions. There is no way to refer to such element using the operation of coverage's element derivation. String literal sets textual representation of this nameless coverage's element.

9.8.8 Shortened definition

The construction se_coverage_declaration is a shortened definition of a coverage on conditions that:

1. There isn't extern;
2. There isn't enum;
3. There aren't parentheses.
4. There isn't an initializer.

Global coverage shortened definition

```
extern coverage enum C( enum E e );

...

coverage C; // shortcut specification
```

Local coverage shortened definition

```
specification void f( int x, int y )
     coverage PointCoverage = IntCoverage( x ) * IntCoverage( y );
// ...

specification void f( int x, int y )
{
   pre{ ... }
   coverage PointCoverage; // shortcut specification

  post{ ... }
}
```

Semantic rules

Above the shortcut specification the `translation_unit` text can contain:

1. full declaration;
2. shortcut declaration.

9.8.9 Common rules of coverage access

Global coverage access is realized after the first declaration by the name(see examples of the full declarations of derived coverages).

Local coverage access allows three different contexts.

1. Outside a prototype or a function definition the local coverage access produces using the chain of <name_of_function> “.” <name_of_coverage> terminals.

2. Inside a prototype of a specification function local coverage can be called both by name and by qualification name of the coverage.
3. Inside a definition of a specification function in the definitions of local coverages the earlier defined coverages of the same function can be called also by name or by qualification name, as a local coverage's name has visibility scope limited by a specification function body.

Access to the local coverage outside the function prototype or definition

```
coverage C( int x, int y ) = f.ArgCoverage( x ) * f.ArgCoverage( y );
```

Access to the local coverage inside the function prototype

```
specification void f( int x, int y )
       coverage X_Coverage
       coverage Y_Coverage
       coverage MutualCoverage = X_Coverage * f.Y_Coverage
     ;
```

Access to the local coverage inside the function definition

```
specification void f( int x, int y )
{
   coverage X_Coverage ... ;
   coverage Y_Coverage ... ;
   coverage MutualCoverage = f.X_Coverage * f.Y_Coverage
   post{ ... }
}
```

Semantic rules

1. The construction `se_coverage_name`, composed of an identifier, point to a coverage, if a dereferencing using C language rules gives us a coverage with the same name.
2. The construction `se_coverage_name` of the form <ID> "." <ID> point to local coverages if:
   2.1. The first identifier refer to a name of a specification function;
   2.2. to this function by the moment of the appearance of considered `se_coverage_name` in code `translation_unit` defined the coverage, the name of which is the same as the second identifier name.

9.9 Rules of performing operations on coverage elements

Variety of different types of coverages needs foresight of different contexts of performing operations on coverages elements. So lets give some definitions to state the rules of access to coverage elements.

Coverages, elements and computation functions of which are defined directly, are called *primary.*

Coverages are called *derived* if they are built on basis of other coverages, which are *basic* in this context. The definition of a derived coverage should describe a method of it's building on the basis of basic coverages; basic coverage can be both primary and derived coverage.

Coverage order is the number of primary coverages, combinations of elements of which are the elements of this coverage. Coverage order of any primary coverage is 1. Coverage order of derived coverage is the sum of orders of all its basic coverages.

For any coverage can be created the list of primary coverages. The derived coverage is based on them finally. The length of this list is equal to the coverage order. This list is formed from the list having only a current coverage in the following way :to substitute the basic coverages instead of the coverages when pass cyclically from the beginning to the end of list (products become orders of multiply coverages).Eventually only primary coverages will left over in the list, their quantity will be equal to the order of the initial coverage.

It is evident that the same lists can be get for different coverages using this algorithm. Coverages are called comparable if the list of primary based coverages is equal for them.

Operations on coverage elements are performed on special type objects - CoverageElement in the context of sintax.For the operations giving the coverage element, a belonging processed or return elements to the coverage is tracked unlike common C -language expressions where only the type of thier result is tracked.

9.9.1 Getting of a constant coverage element

Syntax

This expression syntactically looks like an access to the element of a structure type object in C-language.

```
postfix-expression ::=   // ...
                       | postfix-expression
                         "."
                         identifier
                       | // ...
                   ;
```

Elements of primary coverages are numbered explicit on the basis of the coverage declaration. Explicit numbering allows to use expressions of getting elements wherever the constant expression is necessary for example into case- marks.

For getting a coverage element need to use the special kind of such expression.

```
identifier ( "." identifier )+
```

If the chain of identifiers begins with `se_coverage_name` which point to a coverage then whole the chain of identifiers is viewed as an expression of coverage element getting. The value of this expression has the `CoverageElement` type.

Getting of a constant primary coverage element

```
IntCoverage.POSITIVE
```

The similar expression for the derived coverage's element builds inclusive of its order and a list of primary coverages.

Coverage order is the number of primary coverages, combinations of elements of which are the elements of this coverage. Coverage order of any primary coverage is 1. Coverage order of derived coverage is the sum of orders of all its basic coverages.

For any coverage can be created the list of primary coverages. The derived coverage is based on them finally. The length of this list is equal to the coverage order. This list is formed from the list having only a current coverage in the following way: to substitute the basic coverages instead of the coverages when

pass cyclically from the beginning to the end of list (products become orders of multiply coverages).Eventually only primary coverages will left over in the list, their quantity will be equal to the order of the initial coverage.

As every derived coverage element has exactly one element from every coverage of this list and the sequence order of coverages is unique then the getting of a constant derived coverage element is a constant expression.

Getting of a constant element of the second order coverage

```
f.PointCoverage.ZERO.ZERO
```

Semantic rules

Lets `se_coverage_name` points at the *Cov* coverage(it doesn't matter wheter it has the elements computation function or not)

1. The expression of coverage element getting must be viewed as integer constant expression.
2. If *Cov* is a primary coverage, then :
   2.1. after the `se_coverage_name` point must be present exactly one identifier;
   2.2. this identifier must coincide with the name of an *Cov* coverage element.
3. Otherwise *Cov* coverage is a derived coverage of order N. Following items must be executed:
   3.1. Number of identifiers must be equal to N after the `se_coverage_name`.
   3.2. Every i-th (1<=i<=N) identifier standing after se_coverage_name must coincide with an element of i-th primary coverage from the list which is build using the foregoing method.
4. The result of the whole expression:
   4.1. has the `CoverageElement` type;
   4.2. is a element of *Cov* coverage.

9.9.2 Getting of a component of a derived coverage element

Formative element of any basic primary coverage can be gotten for the element of the derived coverage of order bigger then one

Coverage order is the number of primary coverages, combinations of elements of which are the

elements of this coverage. Coverage order of any primary coverage is 1. Coverage order of derived coverage is the sum of orders of all its basic coverages.

For any coverage can be created the list of primary coverages. The derived coverage is based on them finally. The length of this list is equal to the coverage order. This list is formed from the list having only a current coverage in the following way :to substitute the basic coverages instead of the coverages when pass cyclically from the beginning to the end of list (products become orders of multiply coverages).Eventually only primary coverages will left over in the list, their quantity will be equal to the order of the initial coverage.

Syntax

For getting the component of the derived coverage element the expression which is syntax coincide with the getting array element expression is used.

```
postfix-expression "[" expression "]"
```

If the `CoverageElement` expression is to the left of an opening bracket then all of it is considered as the expression of getting of a component of a derived coverage element.

Getting of a component of a coverage element

```
coverage IntCoverage neg = f.PointCoverage.ZERO[0];
```

Semantic rules

Lets the result of the left part expression is an coverage element Cov_k

1. Coverage *Cov* must be derived coverage of order bigger then one.
2. Expression in square brackets must be an integer constant.
3. The value of this constant must be above or equal null and smaller then the coverage order *Cov*.
4. Result of the whole expression:
   - 4.1. has the `CoverageElement` type;
   - 4.2. is the element of k-th coverage in the list of primary based coverages.

9.9.3 Coverage element computation

Syntax

This expression syntactically looks like an function call in C-language.

```
postfix_expression "(" ( argument_expression-list )? ")"
```

Special view of this expression needs to use for coverage element computation **:**

```
( identifier "." )? identifier "(" ( argument_expression_list )? ")"
```

If the left part of the expression of function call is `se_coverage_name` which is pointing at a coverage then the whole expression is the expression of coverage element computation.

Global coverage element computation

```
IntCoverage( 10 )
```

Local coverage element computation

```
f.PointCoverage( -1, 1 )
```

Semantic rules

1. `se_coverage_name` must point at the computable coverage.
2. Types of expressions-arguments must be consistent with the types of parameters in coverage declaration according the same rules as the types of arguments function call and parameters types from its declaration.
3. Result of the whole expression:
   3.1. has the `CoverageElement` type;
   3.2. is the element of the coverage and `se_coverage_name` points at this coverage.

9.9.4 Expression of a tracing of a coverage element

Expression of a tracing of a coverage element is intended for putting a coverage element is reached in testing process on the report.

Syntax

```
se_primary_expr ::=    ...
                     | se_trace_expression
                ;

se_trace_expression ::= "trace"
            "("
            assignment_expr
            ")"
          ;
```

tracing of a coverage element

```
trace( f.PointCoverage( -10, 10 ) );
```

Semantic rules

1. An argument must have `CoverageElement` type.
2. Result of the whole expression has the void type.

9.9.5 CoverageElement type

`CoverageElement` type is intended for the storing of information about individual coverage elements. The every `CoverageElement` type expression connects with the coverage whose element is the result of this expression.

Semantic rules

1. `CoverageElement` type objects can be used in the following contexts:
   1.1. As a controlling expression in `switch` operator.

1.2. As a case-mark expression in `switch` operator.

1.3. As a parameter inside the parentheses trace-expression.

1.4. In the left and right parts of the equality comparison.

2. `CoverageElement` type is not compatible with any other types.

3. If inside the switch operator the `CoverageElement` type expression of C1 coverage is used then for any case-mark referring to current operator the following conditions must hold:

   3.1. The expression inside the case-mark must be the constant `CoverageElement` type expression. Just one kind of such expressions is expressions of getting of a constant coverage element. Let the expression gives an element of C2 coverage.

   3.2. The C1 and C2 coverages must be comparable among themselves.

   3.3. For any two case-mark whose expressions have `CoverageElement` type this expressions must give different coverage elements. (i.e. Mustn't the same order of identifiers after the first dot in the expressions of getting of a coverage element).

4. (the expanshion of the requirements from 6.5.9 c99 standart) If one of the operands of operator == (or !==) has `CoverageElement` type and belongs to C1 coverage, then:

   4.1. The other operand has `CoverageElement` type also.

   4.2. If the other operand belongs to C2 coverage, then coverages C1 and C2 must be comparable among themselves.

9.9.6 Coverage elements iteration

Coverage elements iteration is a construction which helps to bypass all the functionality branches set by coverage. It is a specific improvement of a simple iteration operator.

Iteration parameters are the name of the called specification function and the coverage name. Coverage elements, which shouldn't be bypassed by any reason, can be sieved using the filter.

Syntax

```
se_iterate_coverage_statement ::= "iterate"
                                  "coverage"
                                  (    "("
                                       se_coverage_name
                                       ( se_coverage_filter )?
                                       ")"
                                     | se_coverage_name
                                  )
                                  statement
                              ;
```

Construction `se_coverage_name` in the iteration headline indicate the coverage, elemnts of which will be looked through as the value of the iteration variable. Uninterestion by any reason coverage elements values can be sieved by an optional filter.

Local coverage iteration with filtration

```
iterate coverage (    f.PointCoverage : f.PointCoverage[0]
                   != IntCoverage.ZERO
```

```
                    )
{
    ...
}
```

Semantic rules

1. Construction `se_iterate_coverage_statement` can be placed only inside of a scenario method definition.
2. There can't be nested `se_iterate_coverage_statement`, which iterate the elements of the same coverage.
3. All the above-stated requirements must be hold by `se_coverage_filter` (see Construction `se_coverage_filter`).

Coverage elements filter construction

Filter-expression makes it possible to sieve the basic coverage elements combinations which are not members of a resulting derived coverage. Filter is a predicate of the coverage element. If on any specific element filter-expression returns 0,it means that the element sieves. In case of using `se_coverage_filter` in the initializer of derived coverages elements of a resulting coverage are sieved and in case of using the filter in the coverage elements iteration – coverage elements on which the iteration is proceeded.

```
coverage NoZeroCoverage = IntCoverage( x )
    : NoZeroCoverage != IntCoverage.ZERO
```

Additional rules of writing filter-expressions are stated in the section Appeal to the earlier calculated coverage element.

Semantic rules

1. If there is a coverage filter-expression, its result must be of scalar type.
2. There can be only such operations inside a filter-expression:
   - 2.1. appeal to the earlier calculated coverage element;
   - 2.2. getting a constant coverage element;
   - 2.3. getting a component of coverage element where there are expressions from subparagraphs 2.1 and 2.2 as the left part;
   - 2.4. comparison of expressions of `CoverageElement type`;
   - 2.5. logical expressions of C language.

9.9.7 Appeal to the earlier calculated coverage element

In some contexts an earlier calculated coverage element is defined. Such an element can be referred to using either name_expr, or field_expr equivalent to se_coverage_name and pointed to the appropriate coverage.

Such contexts are:

- filter-expression in the initiator of derived coverage;
- filter-expression in the coverage elements iteration;
- body of coverage elements iteration;

- post-condition of a specification function.

Inside `se_iterate_coverage_statement` the current element of the iterated coverage is known (current value of iteration variable). Inside a post-condition elements of all the local coverages of embedding function reached with the current parameters are known.

Using the earlier calculated coverage element in a body of coverages iteration

```
iterate coverage IntCoverage
{
   if( IntCoverage == IntCoverage.ZERO )
      f( 0 );
   else
   if( IntCoverage == IntCoverage.POSITIVE )
      f( 10 );
   else
      f( -10 );
}
```

Global coverage iteration (there is an interpretation of name_expr C in the comments).

```
iterate coverage (  C
                  :    C    // element
                    != C.C1 // coverage
                 )
{
   trace( C ); // element
}
```

Local coverage iteration(there is an interpretation of field_expr f.C in the comments).

```
iterate coverage (   f.C
                   :    f.C    // element
                     != f.C.C2 // coverage
                 )
{
   if( f.C( 4 ) // coverage
       == f.C   // element
     )
   {...}
}
```

Semantic rules

1. In some contexts `name_expr` and `field_expr`, equivalent to `se_coverage_name` and pointed to a specific coverage, indicate an element of the coverage:
   1.1. Coverage declared (defined) in a filter-expression of initializer of derived coverage.
   1.2. Coverage iterated in a filter-expression of coverage elements iteration headline.
   1.3. Coverage iterated in a coverage elments iteration body.
   1.4. Local coverage in a post-condition of embedding specification function.
2. Meanwhile situations are mapped out where `se_coverage_name` is a coverage name in:
   2.1. an expression of coverage element calculation;

2.2. an expression of getting a constant element of the coverage.

9.9.8 Coverage element-variable declaration

Coverage elements calculated during a test run can be saved in a variable of `CoverageElement` type. Such variables are declared using a special specification type — `se_coverage_type_specifier`.

```
se_coverage_type_specificer ::= "coverage" se_coverage_name ;
```

Coverage element-variable declaration

```
coverage IntCoverage zero = IntCoverage( 0 );
```

Semantic rules

1. Variables declared in such manner has type `CoverageElement`.
2. Values of these variables are coverage elements specified in `se_coverage_type_specifier`.
3. If declaration (definition) of coverage element-variable has an initializer, then:
   3.1. it must be a single expression of `CoverageElement type`;
   3.2. a result of initializing expression must be an element of the coverage comparable to the coverage specified in se_`coverage_name`.
4. `Aside from that coverage elements-variables conform all the rules of C language semantics.`

9.10 Mediator function

Mediator functions are marked by `mediator for` SeC keywords. An unique identifier—name of the mediator function—must reside between these words. Each mediator function corresponds to a specification function or deferred reaction. Signature and access constraints of this function or reaction must be specified in declarations and definition of the mediator function.

Mediator function can contain:

1. Call block (marked by `call` keyword), implementing behavior, described in the corresponding specification function, by means of performing test action.
2. State block (marked by `state` keyword), implementing synchronization of specification data model state with the system under test state after performed test action or occurred deferred reaction.
3. Auxiliary code before first or after the last named block.

```
( declaration_specifiers )?
"mediator" <ID> "for"
( declaration_specifiers )?
declarator
( declaration )*
compound_statement
;
```

Semantic rules

1. Mediator function names belong to the same namespace as names of usual C functions.
2. Mediator function must be defined exactly once within all translation units (`translation_unit`).
3. Specification function or deferred reaction must be declared before declaration or definition of corresponding mediator function.
4. Declaration specifiers (`declaration_specifiers`) and declarators (`declarator`) of all declarations and definition of a mediator function must contain signature and access constraints of the corresponding specification function or deferred reaction.
5. Mediator of a specification function (`compound_statement`) must contain:
   - auxiliary C-code;
   - call block;
   - state block (may be omitted);
   - auxiliary C-code;
6. Mediator of a deferred reaction (`compound_statement`) must contain:
   - auxiliary C-code;
   - state block;
   - auxiliary C-code;

9.11 Semantic of call block of mediator function

Call block in SeC language is a set of instructions that syntactically and semantically equals to a function body with the same signature and return type as the corresponding specification function. These instructions must be enclosed in curly braces and marked by `call` keyword.

```
se_call_block_statement ::= "call" compound_statement ;
```

Mediator for the function of the selection of stack element

```
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));
  }
  //...
}
```

Semantic rules

1. Call block is the mandatory block of mediators of specification functions.
2. Call block is not permitted in mediators of deferred reactions.
3. Instructions in call block must be syntactically and semantically equal to a function body with the same signature and return type as the corresponding specification function.
4. Call of specification functions and reactions is prohibited in call-block.

9.12 State block of mediator function

State block in SeC language is a set of instructions that syntactically and semantically equals to a function body with the same signature as the corresponding specification function, without return value. These instructions must be enclosed in curly braces and marked by `state` keyword.

Syntax

```
se_state_block_statement ::= "state" compound_statement ;
```

Mediator for the function of element addition to the stack

```
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(stack_impl->elems[--k]))
                    );
  }
}
```

Semantic rules

1. Instructions in state block must be syntactically and semantically equal to a function body with the same signature as the corresponding specification function or deferred reaction, without return value.
2. Value returned by the specification function (or value of the occurred reaction) is accessible via identifier of this function (or reaction), with the exception of `void` functions.
3. Pseudovariable timestamp of TimeInterval type contains start and finish time marks for invocation of call block of this mediator function, or time marks supplied on deferred reaction registration.

9.13 String and XML- view of non-specification types

There are SeC translator output warnings when checking the user's functions of creation string and XML- view of non-specification types in this part.

In time of the declaration process or function definition (the name's prefix of this function is to_string_ or to_XML_) the translator works in a certain way:

1. Compute the suffix <type's_name> - remainder of the function's name.
2. If <type's_name> equals to one of the strings spec, Default, Subtype, then such function is ignored as these are service functions from the library.
3. Also the function is ignored if <type's_name> is a name of specification type.
4. If the type of returned value is not equal to String*, the function is ignored and return the warning:

   return type of '<function's complete name>' is not 'String *'
5. If the quantity of parameters aren't equal to 1, then such function is ignored and return the warning:

   size of parameters list in '<function's complete name>' is not one
6. If the type of parameter isn't <type's_name>, such function is ignored and return the warning:

   parameter of '<function's complete name>' is not '<type's_name> *'

If the concerned function wasn't ignored using the one of above mentioned reason then for the type <type's_name> user function of forming string and XML- representation is set(it depends on a prefix).

10 CTESK test system support library

CTESK includes a support library for tests being developed. The library provides an interface for interaction with the test system as well as a set of additional data types and functions. Header files of the library are located in the include directory of CTESK distributive.

This section describes a part of the library interface intended for test developers use. Another part of the interface defined in header files is intended for generated components of CTESK test system only.

10.1 Base services of the test system

Base services provided by the test system are used via constructions of specification extension of C language.

Base services of CTESK test system included in the support library, consist of a set of data types and functions defining test system time model, and of a small set of system functions of SeC language.

10.1.1 System functions

The following are system functions of SeC language:

- setBadVerdict

  `setBadVerdict` function sets negative verdict of the current mediator call.

- assertion

  `assertion` function examines the value of the specified expression and terminates execution of the application if it evaluates to 0.

setBadVerdict

`setBadVerdict` function sets negative verdict of the current mediator call.

```
void setBadVerdict( const char* msg );
```

Parameters

Msg

Comment describing the reason of negative mediator verdict. This comment appears in test trace and can be used for simplifying test results analysis.

The parameter can possess `NULL` value. No comment appears in test trace in this case.

Additional information

`setBadVerdict` function sets negative verdict of the current mediator call. Mediator must inform test system by calling this function if it cannot perform its task due to some reason.

`setBadVerdict` function can be invoked several times during the execution of one mediator.

`setBadVerdict` function can be invoked out of mediator call. In this case a comment appears in test trace as a user message without other side effects.

Header file: ts/ts.h

Assertion

`assertion` function examines the value of the specified expression and terminates execution of the application if it evaluates to 0.

```
void assertion( int expr, const char* format, ... );
```

Parameters

`expr`

Expression that should not be equal to 0. Application is terminated if this condition is violated.

`format`

Format string for the message about violation of the condition. Message is constructed from the format string and additional parameters in the same way as by `printf` function from C standard library.

Additional information

assertion function examines the specified condition and terminates execution of the application if the condition is violated.

In the case of violation, the specified message appears either in test trace (if the function is invoked within test scenario) or in stderr stream otherwise. Test trace is closed correctly after the message and the application is terminated by exit(1) system call.

Any abnormal test scenario termination must be performed via assertion call, otherwise test trace integrity is not guaranteed.

Header file: utils/assertion.h

10.1.2 Time model

CTESK test system supports three modes of time handling:

- Without accounting time,
- Linear time model,
- Distributed time model.

To define time points, the test system uses *time marks*. Time mark is an abstract value that can either be associated with the real time in some way, or be used just to put time points to an order.

Each time mark belongs to a *frame of time*. All time marks within the same time frame are put in linear order. Time marks of different time frames are not ordered.

In linear time mode, all time marks are considered to belong to the only time frame. Thus all time marks are put in linear order.

In distributed time mode, time marks can belong to different time frames. This mode is the most general, but at the price of the least efficient managing algorithms.

Time model managing in CTESK is performed by the following functions:

- setTSTimeModel
- getTSTimeModel

Time marks are defined with the following data types, constants, and functions:

Data types:

- LinearTimeMark
- TimeFrameOfReferenceID

- TimeMark
- TimeInterval

Constants:

- systemTimeFrameOfReferenceID
- minTimeMark
- maxTimeMark

Functins

- getTimeFrameOfReferenceID
- setSystemTimeFrameOfReferenceName
- createTimeMark
- createDistributedTimeMark
- createTimeInterval
- getCurrentTimeMark
- setDefaultCurrentTimeMarkFunction

setTSTimeModel

`setTSTimeModel` function changes the mode of time handling by CTESK test system.

```
TSTimeModel setTSTimeModel( TSTimeModel time_model );
```

Parameters

time_model

New time handling mode for the test system.

Return value

Previous time handling mode.

Additional information

When dfsm test engine is used, the test system by default:

works without accounting time, if at least one of the saveModelState, restoreModelState, or isStationaryState test scenario fields are not defined or initialized by a `NULL` pointer,

uses linear time model, if all of the saveModelState, restoreModelState, and isStationaryState test scenario fields are defined by non-`NULL` pointers.

Time handling mode can be changed in scenario initialization function.

Header file: ts/timemark.h

GetTSTimeModel

`getTSTimeModel` function returns current time handling mode of CTESK test system.

```
TSTimeModel getTSTimeModel( void );
```

Return value

Current time handling mode of CTESK test system.

Header file: ts/timemark.h

LinearTimeMark

`LinearTimeMark` data type is used for identification of time marks within a time frame.

```
typedef unsigned long LinearTimeMark;
```

Additional information

Values of this data type can represent any characteristic of time points within a time frame. For example, number of seconds (or milliseconds) from the given moment.

If one value of `LinerTimeMark` data type is greater than another value, the time point described by the former time mark is guaranteed to be later than the time point described by the latter time mark. If two values are equal, positional relationship of appropriate time points is unknown: time points can either coincide or not.

Header file: ts/timemark.h

TimeFrameOfReferenceID

TimeFrameOfReferenceID type defines identifiers of time frames.

```
typedef int TimeFrameOfReferenceID;
```

Additional information

The test system is functioning in a dedicated time frame with predefined identifier systemTimeFrameOfReferenceID. In a linear time mode this is the only permitted identifier of time frame.

Other identifiers can be defined in distributed time mode by getTimeFrameOfReferenceID function. The function returns an identifier of a time frame by its name. Two calls to this function with the same name produces the same identifier. Call to this function with a NULL pointer returns an unique time frame identifier, that is guaranteed not to be returned twice.

Header file: ts/timemark.h

TimeMark

`TimeMark` structure defines the type of time mark, used in the test system.

```
typedef struct TimeMark TimeMark;

struct TimeMark {
  TimeFrameOfReferenceID frame;
  LinearTimeMark timemark;
};
```

Additional information

Time mark is characterized by an identifier of time frame and a value indicating the time point within this time frame.

One time mark is less than another time mark when and only when both time marks belong to the same time frame and the value of `timemark` field of the former time mark is less than the value of that field of the latter time mark.

Header file: ts/timemark.h

TimeInterval

`TimeInterval` data type defines time interval between the given two time marks.

```
typedef struct TimeInterval TimeInterval;

struct TimeInterval { TimeMark minMark; TimeMark maxMark; };
```

Additional information

TimeInterval data type defines time interval between minMark and maxMark time marks. Both time marks are required to belong to the same time frame and minMark must be less or equal to maxMark. Boundary time marks are included in the interval.

For indicating minimal or maximal possible time mark, special constants minTimeMark and maxTimeMark must be used.

Header file: ts/timemark.h

systemTimeFrameOfReferenceID

constant defines an identifier of the time frame dedicated for the test system.

```
extern const TimeFrameOfReferenceID systemTimeFrameOfReferenceID;
```

Additional information

Identifier of the time frame dedicated to the test system, can be assigned a symbolic name by setSystemTimeFrameOfReferenceName function. If a name was assigned to the dedicated time frame of the test system, each call to getTimeFrameOfReferenceID with this name returns *systemTimeFrameOfReferenceID*.

Header file: ts/timemark.h

minTimeMark

constant is guaranteed to be less than any other time mark of any time frame.

```
extern const TimeMark minTimeMark;
```

Additional information

minTimeMark constant is a dedicated value of TimeMark data type that is guaranteed to be less than any other time mark regardless of its time frame. minTimeMark constant is equal to itself only.

Header file: ts/timemark.h

maxTimeMark

constant is guaranteed to be greater than any other time mark of any time frame.

```
extern const TimeMark maxTimeMark;
```

Additional information

`maxTimeMark` constant is a dedicated value of **`TimeMark`** data type that is guaranteed to be greater than any other time mark regardless of its time frame. `maxTimeMark` constant is equal to itself only.

Header file: ts/timemark.h

getTimeFrameOfReferenceID

Function returns an identifier of a time frame that corresponds to the specified name.

```
TimeFrameOfReferenceID getTimeFrameOfReferenceID( const char* name );
```

Parameters

name

Name of the time frame.

The parameter can be a `NULL` pointer. If so, the function returns an unique time frame identifier that is guaranteed not to be returned twice.

Return value

Identifier of the time frame with the given name. Repeated calls to this function with the same name evaluates to the same identifier.

Additional information

`getTimeFrameOfReferenceID` function returns an identifier of a time frame by its name. Two calls to this function with the same name produces the same identifier. Call to this function with a `NULL` pointer returns an unique time frame identifier.

`getTimeFrameOfReferenceID` function can be invoked only in distributed time model mode.

Usually each computer has its own time frame. In this case the network name of the computer can be used as the name of its time frame.

For uniformity of handling time frames identifiers, a symbolic name can be assigned to the predefined identifier by setSystemTimeFrameOfReferenceName function.

If a name was assigned to the dedicated time frame of the test system, each call to getTimeFrameOfReferenceID with this name returns systemTimeFrameOfReferenceID.

Header file: ts/timemark.h

setSystemTimeFrameOfReferenceName

Function sets the name of the time frame dedicated for the test system.

```
bool setSystemTimeFrameOfReferenceName( const char* name );
```

Parameters

name

Name of the time frame dedicated for the test system.

The parameter cannot be a `NULL` pointer.

Return value

The function evaluates to `false`, if the given name was already used to identify other time frame, and to `true` otherwise.

Additional information

`setSystemTimeFrameOfReferenceName` function sets the name of the time frame dedicated for the test system. All subsequent calls to getTimeFrameOfReferenceID function returns systemTimeFrameOfReferenceID .

Time frame dedicated to the test system can have several names simultaneously.

Header file: ts/timemark.h

createTimeMark

Function creates a time mark in the time frame dedicated for the test system.

```
TimeMark createTimeMark( LinearTimeMark timemark );
```

Parameters

timemark

Mark of a time point within the time frame dedicated for the test system.

Return value

The function returns a time mark within the time frame dedicated for the test system, identified by `timemark` internal mark.

Additional information

`createTimeMark` function creates a time mark in the time frame identified by systemTimeFrameOfReferenceID and with `timemark` internal mark.

Header file: ts/timemark.h

createDistributedTimeMark

Function creates a time mark in the specified time frame.

```
TimeMark createDistributedTimeMark
         ( TimeFrameOfReferenceID frame, LinearTimeMark timemark );
```

Parameters

frame

Time frame identifier of a time mark being created.

timemark

Mark of a time point within the `frame` time frame.

Return value

The function returns a time mark in the time frame identified by `frame` and with `timemark` internal

mark.

Additional information

Header file: ts/timemark.h

createTimeInterval

Function creates a time interval with the specified bounds.

```
TimeInterval createTimeInterval( TimeMark minMark
                               , TimeMark maxMark
                               );
```

Parameters

minMark

Time mark of the lower bound of the interval.

maxMark

Time mark of the upper bound of the interval.

Return value

The function returns a time interval with the specified bounds.

Additional information

`createTimeInterval` function creates a time interval between `minMark` and `maxMark` time marks. Both time marks must belong to the same time frame and `minMark` must be less than `maxMark`. Boundary time marks are included to the interval.

For indicating minimal or maximal possible time mark, special constants minTimeMark and maxTimeMark must be used.

Header file: ts/timemark.h

getCurrentTimeMark

function returns a time mark corresponding to the current moment of time within the given process.

```
TimeMark getCurrentTimeMark( void );
```

Return value

The function returns a time mark corresponding to the current moment of time within the given process.

Additional information

The function returns a time mark corresponding to the current moment of time within the given process. The function is used by the test system for automatic evaluation of a time interval, containing a specification function call.

By default a current time mark belongs to the time frame dedicated for the test system. Mark within the time frame is calculated as number of seconds since 00:00:00, January 1, 1970 (as value of `time` system function).

This behavior can be redefined by setDefaultCurrentTimeMarkFunction function.

Header file: ts/timemark.h

setDefaultCurrentTimeMarkFunction

Function set up the user-defined function, evaluating time mark for the current moment of time within the given process.

```
GetCurrentTimeMarkFuncType
setDefaultCurrentTimeMarkFunction
( GetCurrentTimeMarkFuncType new_func );
```

Parameters

new_func

Pointer to a function to use for time mark evaluation of the current moment of time within the given process. The parameter must be not `NULL`.

Return value

The function returns a pointer to the previously used current time mark evaluating function.

Additional information

`setDefaultCurrentTimeMarkFunction` function set up the user-defined function, evaluating time mark for the current moment of time within the given process. All subsequent calls of getCurrentTimeMarkfunction return time marks, evaluated by that function.

The user-defined function is used for automatic evaluation of a time interval, containing a specification function call.

Header file: ts/timemark.h

11 Standart test engines

CTESK 2.8 includes two test engines: dfsm and ndfsm. They allow to test a wide class of software — from simple systems without internal state to distributes systems with asynchronous interfaces.

11.1 Dfsm

The dfsm test engine is based on traversal of finite state machine. Finite state machine, used for test building, is defined explicitly by defining function to evaluate current scenario state and a set of test actions.

During test run the dfsm applies the test actions that can change scenario state. The 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.

The dfsm test engine 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.

A set of test actions is defined by scenario functions.

The dfsm data type of test scenario is used in initialization of a test scenario, based on the dfsm test engine. Fields of this type are described on page «Fields of data types of test scenario»

Additional parameters of the dfsm test engine can be tuned by the following functions:

- setFinishMode
- setDeferredReactionsMode
- setWTime
- setFindFirstSeriesOnly
- setFindFirstSeriesOnlyBound

All functions, which can set or return test engine parameters are enumerates on page «Test engine service function».

A list of parameters is passed to the test scenario on its invocation. The dfsm test engine has several standard parameters affected its behavior. Standard parameters must precede user parameters. The dfsm test engine processes standard parameters and passes the rest of parameters to the scenario initialization function.

11.2 Ndfsm

The ndfsm test engine, comparing with dfsm, 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.

A set of test actions is defined by scenario functions.

The ndfsm data type of test scenario is used in initialization of a test scenario, based on the ndfsm test engine. Fields of this type are described on page «Fields of data types of test scenario».

Additional parameters of the test engine can be tuned by the following functions:

- setFinishMode
- setDeferredReactionsMode
- setWTime
- setFindFirstSeriesOnly
- `setFindFirstSeriesOnlyBound`

All functions, which can set or return test engine parameters are enumerates on page «`Test engine service function`».

A list of parameters is passed to the test scenario on its invocation. The test engine has the same standard parameters affected its behavior. Standard parameters must precede user parameters. The test engine processes standard parameters and passes the rest of parameters to the scenario initialization function.

11.3 Fields of data types of test scenario

The dfsm data type of test scenario is used in initialization of a test scenario, based on the dfsm test engine. Correspondingly, the ndfsm data type of test scenario is used in initialization of a test scenario, based on the ndfsm test engine. This data type is a structure; set of dfsm fields includes set of nfdsm fields. These fields are in both structures:

- *init* (type PtrInit)
- *finish* (type PtrInit)
- *getState* (type PtrGetState)
- *actions*

The only mandatory field is *actions*, which contains array of scenario functions, defining test actions. dfsm structure includes additional fields:

- *saveModelState* (type PtrSaveModelState)
- *restoreModelState* (type PtrRestoreModelState)
- *isStationaryState* (type PtrIsStationaryState)
- *observeState* (type PtrObserveState)
- *init*
  init field contains a pointer to test scenario initialization function.
- *finish*
  finish field contains a pointer to test scenario finalization function.
- ***getState***
- *getState* field contains a pointer to a function, evaluating current test scenario state.
- ***actions***
  actions field contains an array of scenario functions, ended with a NULL pointer.
- ***saveModelState***
  saveModelState field contains a pointer to function for saving specification data model state.
- ***restoreModelState***
  restoreModelState field contains a pointer to function, restoring specification data model state.
- ***isStationaryState***
  isStationaryState field contains a pointer to function for checking model state stationarity.
- ***observeState***
  observeState field contains a pointer to function for synchronization of model state with state of the system under test after stabilization time.

init

init field contains a pointer to test scenario initialization function.

```
PtrInit init;
```

Additional information

Initialization function takes an array of parameters, semantically similar to argc and argv parameters of main standard function. It can use them for test scenario tuning.

Normally initialization function performs the following actions:

- initialization of the system under test,
- initialization of specification data model,
- initialization of scenario data,
- setting mediators for specification functions and reactions used in this test scenario.

In the case of testing of the systems with deferred reactions, the function is additionally us

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

Initialization function returns a boolean value. It must evaluate to true if initialization completed successfully, and to false otherwise. In the latter case test scenario is terminated, and finalization function is not invoked.

Initialization function can contain specification functions calls, performing initialization of the system under test, specification or scenario. If deferred reactions support mode is set a test engine makes serialization of all stimuli sent in the initialization function call and all reactions received.

init field can be initialized by a NULL pointer or can be not initialized at all. It equivalents to an empty initialization function returning true.

Header file: ts/dfsm.h, ts/ndfsm.h

finish

finish field contains a pointer to test scenario finalization function.

```
PtrFinish finish;
```

Additional information

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

Normally scenario finalization function performs the following actions:

- freeing resources of the system under test, allocated by the test scenario,
- freeing resources of specification data model,
- freeing resources of the test scenario.

Finalization function can contain specification functions call, realized freeing resources of the system under test, freeing resources of specification data model or resources of the scenario. If deferred reactions support mode is set a test engine makes serialization of all stimuli sent in the

initialization function call and all received reactions.

finish field can be initialized by a NULL pointer or can be not initialized at all. It this case no actions to scenario completion execute.

Header file: ts/dfsm.h, ts/ndfsm.h

getState

getState field contains a pointer to a function, evaluating current test scenario state.

```
PtrGetState getState;
```

Additional information

Evaluation test scenario state function has no parameters and returns an object of a specification data type.

It is important to take into account determinancy constraint of dfsm test engine or presence of deterministic strongly connected complete spanning submachine of ndfsm test engine when evaluating test scenario state.

getState field can be initialized by a NULL pointer or can be not initialized at all. The dfsm and ndfsm test engines consider this as scenario with a single state.

Header file: ts/dfsm.h, ts/ndfsm.h

actions

actions field contains an array of scenario functions, ended with a NULL pointer.

```
ScenarioFunctionID actions[];
```

Additional information

actions field contains an array of scenario functions, ended with a NULL pointer.

actions field is mandatory for initialization. Last element of the array must be a NULL pointer.

Header file: ts/dfsm.h, ts/ndfsm.h

saveModelState

saveModelState field contains a pointer to function for saving specification data model state.

```
PtrSaveModelState saveModelState;
```

Additional information

Function for saving specification data model state has no parameters and returns an object of a specification data type. The object must contain the whole state of specification data model. This object is used then by a function for restoring specification data model state to completely restore the state.

saveModelState field can be initialized by a NULL pointer or can be not initialized at all. It

makes testing with deferred reactions impossible.

Header file: ts/dfsm.h, ts/ndfsm.h

`restoreModelState`

`restoreModelState` field contains a pointer to function, restoring specification data model state.

```
PtrRestoreModelState restoreModelState;
```

Additional information

Function for restoring specification data model state takes an object of a specification data type and restores the state of specification data model using this object. The object is guaranteed to be previously constructed by a `function for saving specification data model state`. The function for restoring state does not return a value.

`restoreModelState` field can be initialized by a NULL pointer or can be not initialized at all. It makes testing with deferred reactions impossible.

Header file: ts/dfsm.h, ts/ndfsm.h

`IsStationaryState`

`isStationaryState` field contains a pointer to function for checking model state stationarity.

```
PtrIsStationaryState isStationaryState;
```

Additional information

Function for checking model state stationarity has no parameter and evaluates to true, if current model state is stationary, and to false otherwise.

Model state is called stationary, if the target system that meets the model cannot initiate interaction in this state.

`isStationaryState` field can be initialized by a NULL pointer or can be not initialized at all. It makes testing with deferred reactions impossible.

Header file: ts/dfsm.h, ts/ndfsm.h

`ObserveState`

observeState field contains a pointer to function for synchronization of model state with state of the system under test after stabilization time.

```
PtrObserveState observeState;
```

Additional information

Model state synchronization function has no parameters and no return value. It is invoked in the end of stabilization time after the test system initiates next test action and the target system comes to a stationary state. Synchronization function can invoke one or more specification functions that read state of the system under test but not change it. Interactions initiated during synchronization counts in serialization process as well as previous test actions.

observeState field can be initialized by a NULL pointer or can be not initialized at all. No synchronization is performed in this case**.**

Header file: ts/dfsm.h, ts/ndfsm.h

11.4 Data types used by test engine

Data types in the following list are intended for storing values of test scenario fields and references of user service function.

- **FinishMode**
  FinishMode enumeration data type defines possible modes of dfsm test engine finalizing.
- **PtrFinish**
  PtrFinish field specifies test scenario finalization function type.
- **PtrGetState**
  PtrGetState data type specifies type of function, evaluating current test scenario state.
- **PtrInit**
  PtrInit data type specifies test scenario initialization function type.
- **PtrIsStationaryState**
  PtrIsStationaryState data type specifies type of a function for checking model state stationarity.
- **PtrObserveState**
  PtrObserveState data type specifies type of a function which synchronizes model state with the state of system under test on the expiry of stabilization time.
- **PtrRestoreModelState**
  PtrRestoreModelState data type specifies type of a function to restore specification data model state.
- **PtrSaveModelState**
  PtrSaveModelState data type specifies type of function, returning specification data model state.

FinishMode

FinishMode enumeration data type defines possible modes of dfsm test engine finalizing.

```
typedef enum { UNTIL_ERROR, UNTIL_END } FinishMode;
```

Additional information

FinishMode enumeration data type defines possible modes of test engine finalizing.

First mode - UNTIL_ERROR indicates that testing is finished immediately after first error detection.

Second mode – UNTIL_END indicates that testing is continued after detection of non-critical error and is finished just on reaching desired coverage criteria.

By default test engines are operating in UNTIL_ERROR mode.

Header file: ts/engine.h

PtrFinish

PtrFinish field specifies test scenario finalization function type.

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

Additional information

Scenario finalization function has no parameters and no return value. It intended for freeing resources after scenario completion.

Header file: ts/engine.h

PtrGetState

`PtrGetState` data type specifies type of function, evaluating current test scenario state.

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

Additional information

Evaluation test scenario state function has no parameters and returns an object of a specification type.

Header file: ts/engine.h

PtrInit

`PtrInit` data type specifies test scenario initialization function type.

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

Additional information

Initialization function takes an array of parameters, semantically similar to argc and argv parameters of main standard function, and returns a boolean value. It evaluates to true if initialization completed successfully, and to false otherwise.

Header file: ts/engine.h

PtrIsStationaryState

`PtrIsStationaryState` data type specifies type of a function for checking model state stationarity.

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

Additional information

Function for checking model state stationarity has no parameters and returning a boolean value.

Header file: ts/engine.h

PtrObserveState

`PtrObserveState` data type specifies type of function for synchronization of model state with state of the system under test after stabilization time.

```
typedef void (*PtrObserveState)( void );
```

Additional information

Model state synchronization function has no parameters and no return value.

Header file: ts/engine.h

PtrRestoreModelState

PtrRestoreModelState data type specifies type of a function to restore specification data model state.

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

Additional information

The function takes an object of a specification data type and restores the state of specification data model using this object. The function does not return a value.

Header file: ts/engine.h

PtrSaveModelState

PtrSaveModelState data type specifies type of function, returning specification data model state.

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

Additional information

Function for saving specification data model state has no parameters and returns an object of a specification data type.

Header file: ts/engine.h

11.5 Test engine service function

Test engine can be ruled by functions from following list:

- areDeferredReactionsEnabled
  `areDeferredReactionsEnabled` function returns current deferred reactions support mode of test engine.
- getFindFirstSeriesOnlyBound
  `getFindFirstSeriesOnlyBound` function returns current value of `FindFirstSeriesOnlyBound`.
- getFinishMode
  `getFinishMode` function returns test engine current finalization mode.
- getWTime
  `getWTime` function returns waiting period of a test engine for a target system stabilization.
- isFindFirstSeriesOnly
  `isFindFirstSeriesOnly` function returns current value of `FindFirstSeriesOnly property of a test system.`
- setDeferredReactionsMode
  `setDeferredReactionsMode` function sets deferred reactions support mode of test engine.
- setFindFirstSeriesOnly
  `setFindFirstSeriesOnly` function sets `FindFirstSeriesOnly property of a test system.`
- setFindFirstSeriesOnlyBound
  `setFindFirstSeriesOnlyBound` function sets `FindFirstSeriesOnlyBound property of a test system.`
- setFinishMode
  `setFinishMode` sets value of test engine finalization mode.
- setWTime
  `setWTime` function sets waiting period of a test engine for a target system to stabilize.

areDeferredReactionsEnabled

areDeferredReactionsEnabled function returns current deferred reactions support mode of test engine.

```
bool areDeferredReactionsEnabled( void );
```

finish_mode

New mode of test scenario finalization.

Return value

Current deferred reactions support mode of test engine.

Additional information

setDeferredReactionsMode is intended for change deferred reactions support mode.

Header file: ts/engine.h

getFindFirstSeriesOnlyBound

getFindFirstSeriesOnlyBound function returns current value of FindFirstSeriesOnlyBound property.

```
int getFindFirstSeriesOnlyBound( void );
```

Return value

Current value of FindFirstSeriesOnlyBound property.

Additional information

setFindFirstSeriesOnlyBound function can be used to change value of the FindFirstSeriesOnlyBound property.

Header file:ts/engine.h

getFinishMode

getFinishMode function returns test engine current finalization mode.

```
FinishMode getFinishMode( void );
```

Return value

Current test engine finalization mode.

Additional information

getFinishMode function returns current test engine finalization mode.

setFinishMode function can be used to change the value of current finalization mode.

Header file: ts/engine.h

getWTime

The function returns waiting period of a test engine for a target system stabilization.

```
time_t getWTime( void );
```

Return value

Waiting period of a test engine for a target system stabilization.

Additional information

getWTime function returns waiting period of a test engine for a target system stabilization.

setWTime function can be used to change waiting period for the target system stabilization.

Header file: ts/engine.h

isFindFirstSeriesOnly

isFindFirstSeriesOnly function returns current value of FindFirstSeriesOnly property of a test system.

```
bool setFindFirstSeriesOnly( void );
```

Return value

Current value of FindFirstSeriesOnly property.

Additional information

setFindFirstSeriesOnly function can be used to change value of the FindFirstSeriesOnly property.

Header file: ts/engine.h

setDeferredReactionsMode

setDeferredReactionsMode function sets deferred reactions support mode of test engine.

```
bool setDeferredReactionsMode( bool enable );
```

Parameters

enable

If this parameters is true, support for deferred reactions is turned on, otherwise off.

Return value

The function returns previous value of support mode for deferred reactions of test engine.

Additional information

setDeferredReactionsMode function sets deferred reactions support mode of test engine. Support for deferred reactions cannot be turned on if some of the saveModelState, restoreModelState, or isStationaryState fields of test scenario are not defined or are initialized by a NULL pointer. If all of these fields are initialized by a non-NULL pointer, mode for deferred reactions of test engine is turn on default.

Deferred reactions support mode can be changed only within test scenario initialization function.

areDeferredReactionsEnabled function can be used to access current value of deferred reactions support mode.

Header file: ts/engine.h

setFindFirstSeriesOnly

setFindFirstSeriesOnly function sets FindFirstSeriesOnly property of a test system.

```
bool setFindFirstSeriesOnly( bool new_value );
```

Paremeters

new_value

Value of FindFirstSeriesOnly property of the test system.

Return value

The function returns previous value of FindFirstSeriesOnly property of the test system.

Additional information

`setFindFirstSeriesOnly` function sets `FindFirstSeriesOnly` property of the testing system. During serialization, if the property is false, the test system constructs all possible sequences of interactions and checks whether they all lead to the same specification data model state. In other words, it checks for determinancy of the model. When it is known (for some reason) that all allowable sequences of interactions lead to the same state, it is possible to set `FindFirstSeriesOnly` property to true, thus optimizing test system operation. For example, when the model consists the only stationary state, the indicated above condition is certainly satisfied and it is possible to set `FindFirstSeriesOnly` property to true.

By default the property is false.

`FindFirstSeriesOnly` property can be changed only during test scenario operation, including test scenario initialization function. Changes of this property before test scenario start will not affect its behavior.

`isFindFirstSeriesOnly` function can be used to get current value of the property.

Header file: ts/engine.h

`SetFindFirstSeriesOnlyBound`

Function sets `FindFirstSeriesOnlyBound` property of a test system.

```
int setFindFirstSeriesOnlyBound( int bound );
```

Parameters

bound

Value of `FindFirstSeriesOnlyBound` property of the test system.

Return value

The function returns previous value of `FindFirstSeriesOnlyBound` property of the test system.

Additional information

During serialization, if the property `FindFirstSeriesOnlyBound` is zero, the test system constructs all possible sequences of interactions and checks whether they all lead to the same specification data model state.

During serialization, if the property `FindFirstSeriesOnlyBound` is positive and the number

of interactions is less than `FindFirstSeriesOnlyBound`, the test system constructs all possible sequences of interactions and checks whether they all lead to the same specification data model state. If the number of interactions is greater than or equal to `FindFirstSeriesOnlyBound`, the test system considers the only possible sequence of interations.

By default the property is 0.

`setFindFirstSeriesOnlyBound(0)` call is equal to `setFindFirstSeriesOnly(false)` call.

`setFindFirstSeriesOnlyBound(1)` call is equal to `setFindFirstSeriesOnly(true)` call.

FindFirstSeriesOnlyBound property can be changed only during test scenario operation, including test scenario initialization function. Changes of this property before test scenario start will not affect its behavior.

getFindFirstSeriesOnlyBound function can be used to get current value of the property.

Header file: ts/engine.h

`setFinishMode`

`setFinishMode` function sets test engine finalization mode.

```
FinishMode setFinishMode( FinishMode finish_mode );
```

Parameters

`finish_mode`

New test engine finalization mode.

Return value

Previous test engine finalization mode.

Additional information

`SetFinishMode` function sets test engine finalization mode. By default test engine is operating in UNTIL_ERROR mode.

Finalization mode can be changed at any moment during test system operation. Mode change affects only following errors and does not affect previous ones.

getFinishMode function can be used to access the value of current finalization mode.

Header file: ts/engine.h

`setWTime`

`setWTime` function sets waiting period of a test engine for a target system to stabilize.

```
time_t setWTime(time_t secs);
```

Parameters

secs

Waiting period for the target system stabilization, in seconds. Value of the parameter must be non-negative integer number.

Return value

The function returns previous value of waiting period for the target system stabilization.

Additional information

setWTime function sets waiting period for the target system stabilization. The test engine waits specified time after each test action for all information about deferred reactions to be gathered and the target system to stabilize.

By default waiting period is equal to 0.

Waiting period ca be changed only within initialization function. getWTime function can be used to get current value of waiting period.

Header file: ts/engine.h

11.6 Standard parameters of test scenario

A list of parameters is passed to the test scenario on its invocation. The test engine has several standard parameters affected its behavior. Standard parameters must precede user parameters. The test engine processes standard parameters and passes the rest of parameters to the scenario initialization.

In CTESK 2.8 test engine supports the standard parameters, given in list.

- **--ffso**
  --find-first-series-only
  Indicate to fine only first successful series.
- **-nt**
  --no-trace
  Disables tracing.
- **-t**
  Send tracing to file with specifies name.
- **-tc**
  Send tracing to the console.
- **--trace-accidental**
  Turns tracing of information about uncertain transitions on.
- **-tt**
  Send tracing to the file with unique name compose from scenario name and current time.
- **-uend**
  Indicate to execute test to end in spite of errors.
- **-uerr**
  Indicate to execute test before first error appear. (by default).
- **--trace-format**
  Indicate format of data drop to trace.
- **--disabled-actions**
  Indicate file name with list of scenario functions, which will not be invoked.

--find-first-series-only
-ffso

Indicate to fine only first successful series.

Additional information

`--find-first-series-only` standard parameter (-ffso for short) sets true value of FindFirstSeriesOnly property. It means that during serialization the test system does not construct all possible sequences of interactions and checks whether they all lead to the same specification data model state, but uses the only allowable sequence of interactions.

The value of FindFirstSeriesOnly property can be changed by the setFindFirstSeriesOnly function at test scenario initialization and during its operation.

By default the property if false.

-nt
--no-trace

Disables tracing

Additional information

`--no-trace` standard parameter (`-nt` for short) disables tracing. This parameter can not be used with '`-t`', '`-tt`' or '`-tc`' standard parameters.

-t

Send tracing to file with specifies name.

Additional information

'`-t` <file-name>' standard parameter adds the specified file to the set of devices for receiving test trace. If file name is not specified or the file cannot be opened for write, test scenario is abnormally terminated.

-tc

Send tracing to the console.

Additional information

`-tc` standard parameter adds console to the list of devices for receiving test trace. Console means standard output stream of the process the test system operates in.

--trace-accidental

Turns tracing of information about uncertain transitions on.

Additional information

'`--trace-accidental`' standard parameter turns tracing of information about uncertain transitions on. The tracing information about uncertain transitions can be changed by the `setTraceAccidental` function at test scenario initialization and during its operation. By default the tracing information about uncertain transitions is turned off.

-tt

Send tracing to the file with unique name compose from scenario name and current time.

Additional information

`-tt` standard parameter adds a file with the automatically generated name `<scenario_name>-YYYY-MM-DD--HH-MM-SS.utt` to the list of devices for receiving test trace. If the file with that name cannot be opened for write, test scenario is abnormally terminated.

`-tt` parameter is default parameter: test launch without command line parameters is equivalent test launch with `-tt` parameter.

-uend

Indicate to execute test to end in spite of errors.

Additional information

'-uend' standard parameter sets the value of scenario finalization mode. The value can be changed by setFinishMode at test scenario initialization and during its operation.

If scenario parameters contain several standard parameters for scenario initialization mode, only last of them will be taken into account by the test engine.

-uerr

Indicate to execute test before first error appear. (by default).

Additional information

-uerr standard parameter sets the value of scenario finalization mode. The value can be changed by setFinishMode at test scenario initialization and during its operation.

If scenario parameters contain several standard parameters for scenario initialization mode, only last of them will be taken into account by the test engine.

The ndfsm test engine allows to set the value of the -uerr standard parameter, which stands for the maximum permitted number of the errors. The format -uerr=number_of_errors is used for this purpose.

--trace-format

Indicate format of data drop to trace.

Additional information

--trace-format standard parameter sets format, in which complex data type data is dropped to trace. This parameter influences on dropping to trace view (in the sequel to the report) of values of iteration variables values of scenario, scenario state, values of method invocation parameters and return values of these methods. Two values of parameter are acceptable:

--trace-format=xml (on default) and –trace-format=string. In the first case trace saving information about complex data types(if these types were used in the scenario) which transform into expand tree in report is turn out. In the second case trace of smaller volume, demanding less resources for report generation is turn out.

--disabled-actions

Indicate file name with list of scenario functions, which will not be invoked.

Additional information

--disabled-actions=file_name standard parameter value specifies file name with list of scenario functions names, which are not invoke by test engine. File must exist and contains list of scenario functions names separated by a line feed.

12 Tracing services

Tracer of CTESK test system provides a possibility to store information about test process for its subsequent analysis. For that all components of the test system automatically traces information about their work in a special format. Then report generator uses the test trace for testing results analysis and to building different kinds of reports.

For a user the tracer provides tracing control interface and message tracing interface.

12.1 Tracing control

Tracing control functions are divided into two classes: trace saving control functions and trace

contents control functions.

Trace saving control functions are operates as follows. Test trace can be simultaneously saved to several devices. CTESK 2.8 supports two kinds of devices—console (as standard output stream of the process the test system operates within) and file.

The tracer saves test trace to devices included in the set of devices for trace saving. To add a device in the set, functions from addTraceTo... group are used. If the specified device already belongs to the set, its entry counter is incremented.

To remove a device from the set of devices for trace saving, functions from removeTraceTo...

group are used. If the specified device was added to set several times, the operation just decrements its entry counter. The device will be actually removed from the set only when its entry counter becomes equal to zero.

The set of devices for trace saving can be changed when no one test scenario is running. In particular, the set cannot be changed within test scenario initialization function.

Trace saving control functions:

- addTraceToConsole
- removeTraceToConsole
- addTraceToFile
- removeTraceToFile

Trace contents control functions specify set of the tracer's messages and their format. In CTESK 2.8 next trace contents control function are available trace contents control function is:

- setTraceAccidental

setTraceUserEnv

Function to set trace character encoding is:

- setTraceEncoding

addTraceToConsole

Function adds the console to the set of devices for trace saving.

```
void addTraceToConsole( void );
```

Additional information

addTraceToConsole function adds the console to the set of devices for trace saving. If the console already belongs to the set, its entry counter is incremented.

Console is the standard output stream of the process the test system operates within.

addTraceToConsole function can be invoked only when no one test scenario is running.

Header file: ts/c_tracer.h

removeTraceToConsole

Function removes the console from the set of devices for trace saving.

```
void removeTraceToConsole( void );
```

Additional information

removeTraceToConsole function removes the console from the set of devices for trace saving. If its entry counter is greater than unit, the counter is decremented and console will not be actually removed from the set. Console is the standard output stream of the process the test system operates within.

removeTraceToConsole function can be invoked only when no one test scenario is running.

Header file: ts/c_tracer.h

addTraceToFile

Function adds the specified file to the set of devices for trace saving.

```
bool addTraceToFile( const char* name );
```

Parameters

name

Name of the file to be added to the set of devices for trace saving.

The parameter cannot be a NULL pointer.

Return value

The function returns true if the file was added successfully, and false otherwise. A reason for a false result, for example, can be impossibility to open a file with the specified name to write.

Additional information

`addTraceToFile` function adds the specified file to the set of devices for trace saving. If the file already belongs to the set, its entry counter is incremented.

The function return false if the file cannot be opened to write. `addTraceToFile` function can be invoked only when no one test scenario is running.

Header file: ts/c_tracer.h

removeTraceToFile

Function removes the specified file from the set of devices for trace saving.

```
bool removeTraceToFile( const char* name );
```

Parameters

name

Name of the file to be removed from the set of devices for trace saving.

The parameter cannot be a NULL pointer.

Return value

The function returns false if a file with the specified name does not belong to the set of devices for trace saving, and true in case of successful file deletion.

Additional information

`removeTraceToFile` function removes the specified file from the set of devices for trace saving. If its entry counter is greater than unit, the counter is decremented and the file will not be actually removed from the set.

`removeTraceToFile` function can be invoked only when no one test scenario is running.

Header file: ts/c_tracer.h

`setTraceAccidental`

`setTraceAccidental` function turns tracing of information about uncertain transitions on or off.

Uncertain transitions are transitions corresponding to those scenario functions calls, which does not produce specification calls.

```
bool setTraceAccidental( bool enable);
```

Parameters

`enable`

The function turns uncertain transitions tracing on if the parameter equals to true, and off otherwise.

Return value

The function returns previous value of the property of uncertain transitions tracing.

Additional information

By default the tracer does not save information about uncertain transitions, therefore `setTraceAccidental` function

can be used when it necessary. `setTraceAccidental` function can be invoked only when no one test scenario is running.

Header file: ts/c_tracer.h

`setTraceUserEnv`

`SetTraceUserEnv` sets value of user environment variables. Values of these variables with information about system drop in the beginning of the trace.

```
void setTraceUserEnv( const char* name, const char* value);
```

Parameters

`name`

Name of the user environment variable.

Value

Value of variable.

Additional information

Value of variables, which are set by `setTraceUserEnv` function, can describe, for example, remote system parameters, which test can not define automatically. They are seen in report, and also can be used for detected errors identification in known errors DB.

Function must be invoked before scenario.

Header file: ts/c_tracer.h

`setTraceEncoding`

`setTraceEncoding` function sets trace character encoding.

```
void setTraceAccidental( const char* encoding);
```

Parameters

encoding

The identifier of the trace character encoding.

Additional information

`setTraceEncoding` function sets trace character encoding. By default, character encoding of the trace is UTF-8.

Setting of trace character encoding is necessary for correct representation in reports of local string, used in the capacity of functionality branches names, subsystems names or user messages.

Function must be invoked before scenario.

Header file: ts/c_tracer.h

12.2 Message tracing

CTESK 2.8 includes the only kind of tracer messages, available to the user. These messages are called user messages. They play an auxiliary part and are used mainly for manual test trace analysis.

But some error reports show user messages to simplify analysis.

The following functions are used for tracing user messages:

- traceUserInfo
- traceFormattedUserInfo

traceUserInfo

`traceUserInfo` function saves the user message in the test trace.

```
void addTraceToConsole( const char* info );
```

Parameters

info

Pointer to a character string, followed by a zero character, which contains a user message.

Additional information

`traceUserInfo` function saves the user message in the test trace. User messages play an auxiliary part and are used mainly for manual test trace analysis. But some error reports show user messages to simplify analysis.

Header file: ts/c_tracer.h

traceFormattedUserInfo

Function saves the formatted user message in the test trace.

```
void addTraceToConsole( const char* format, ... );
```

Parameters

format

Pointer to a character string, followed by a zero character, which contains a format of a user message. The string can contain any of 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.

Additional information

`traceFormattedUserInfo` function formats and saves user message in the test trace. User messages play an auxiliary part and are used mainly for manual test trace analysis. But some error reports show user messages to simplify analysis.

Header file: ts/c_tracer.h

13 Deferred reactions registration services

CTESK test system supports testing of systems with deferred reactions. Systems with deferred reactions are systems which can participate in several interactions simultaneously or can initiate interactions with their environment themselves.

One of the important task when testing systems with deferred reactions is gaining all necessary

information about interactions with the target system. This information is requested by CTESK test system to check whether the target system behavior conforms to its specification, because these are the interactions which reflect behavior of the target system with deferred reactions.

The test system automatically registers all interactions initiated by calls of specification functions within the test system process. All other interactions must be registered by the test developer in a special test system component—interactions registrar.

Each interaction with the target system is characterized by the channel, in which it occurs. The test system uses identifiers of interaction channels to identify them.

For father convenience of deferred reactions registration, the test system provides `catcher functions registering service.`

13.1 Interaction channels

Each interaction with the target system is characterized by the channel, in which it occurs. All interactions within the same channel are linearly ordered. Thus the test system assumes that within the same channel the first registered interaction has occurred earlier than the second one.

Identifiers of interaction channels, used for identifying channels within the test system, has `ChannelID` data type.

There are two predefined constants of this data type:

- *`WrongChannel`*
- *`UniqueChannel`*

To allocate a channel identifier and then free it, the following functions are intended:

- getChannelID
- releaseChannelID

`ChannelID`

`ChannelID` data type is used for identification of interaction channels within the test system.

```
typedef long ChannelID;
```

Additional information

`ChannelID` data type is used for identification of interaction channels within the test system. There are two constants of this data type: *`WrongChannel`* and *`UniqueChannel`*.

`getChannelID` function returns a newly allocated channel identifier. When the identifier becomes unnecessary, it can be freed by `releaseChannelID` function.

A channel identifier is correct when it is equal to UniqueChannel constant, or it was returned by `getChannelID` function and is not equal to `WrongChannel`.

Header file: ts/register.h

`WrongChannel`

WrongChannel constant indicates an incorrect interactions channel identifier.

```
extern const ChannelID WrongChannel;
```

Additional information

WrongChannel constant indicates an incorrect interactions channel identifier. This constant is playing an auxiliary part, for example, `getChannelID` function returns the constant if it cannot allocate new channel identifier.

Header file: ts/register.h

`UniqueChannel`

`UniqueChannel constant indicates an unique channel. Only one interaction`

can occur in a unique channel, and other interactions cannot occur in it channel in principle.

```
extern const ChannelID UniqueChannel;
```

Additional information

`UniqueChannel` constant indicates an unique channel. Only one interaction can occur in a unique channel, and other interactions cannot occur in it channel in principle. This constant is frequently used when interaction channels have no sense for the target system modeling.

Header file: ts/register.h

`getChannelID`

`getChannelID` function returns a newly allocated interactions channel identifier.

```
ChannelID getChannelID( void );
```

Return value

Function returns a newly allocated interactions channel identifier if it can, or *`WrongChannel`* otherwise.

Additional information

No longer necessary channel identifier can be freed by `releaseChannelID` function.

Header file: ts/register.h

`releaseChannelID`

`releaseChannelID` function frees the specified interactions channel identifier.

```
void getChannelID( ChannelID chid );
```

Parameters

`chid`

Channel identifier to be freed.

The parameter must be a channel identifier, returned previously by getChannelID function.

Additional information

Header file: ts/register.h

13.2 Interactions registrar

The test system automatically registers all interactions initiated by means of specification functions calls within the test system process. Interactions are considered to occur in a channel specified by StimulusChannel property. To control the property the following functions are intended:

- setStimulusChannel
- getStimulusChannel

By default the property is equal to `UniqueChannel`.

All other interactions must be registered by the test developer in the interaction registrar by means of the following functions:

- registerReaction
- registerReactionWithTimeMark
- registerReactionWithTimeInterval
- registerWrongReaction
- registerStimulusWithTimeInterval

setStimulusChannel

`setStimulusChannel` function set the value of StimulusChannel property.

```
ChannelID setChannelID( ChannelID chid );
```

Parameters

Chid

Interactions channel identifier to be used by the test system when automatically registering interactions.

The parameter must be a correct channel identifier.

Return value

The function returns previous value of `StimulusChannel` property.

Additional information

`StimulusChannel` property contains a channel identifier to be used by the test system for identifying a channel for interactions, initiated by means of specification function calls within the test system process. By default this property is equal to *UniqueChannel*.

To access current value of `StimulusChannel` property, `getStimulusChannel` function is intended.

Header file: ts/register.h

getStimulusChannel

`getStimulusChannel` function returns the value of `StimulusChannel` property.

```
ChannelID getChannelID( void );
```

Return value

The function returns the value of StimulusChannel property.

Additional information

`StimulusChannel` property contains a channel identifier to be used by the test system for identifying a channel for interactions, initiated by means of specification function calls within the test system process. By default this property is equal to *`UniqueChannel`*.

To change the value of `StimulusChannel` property, `setStimulusChannel` function is intended.

Header file: ts/register.h

registerReaction

`registerReaction` function is intended for registration of reactions, received from the target system.

```
void registerReaction( ChannelID chid, const char* name,
SpecificationID reactionID, Object* data );
```

Parameters

`chid`

Interactions channel identifier to be used by the test system when automatically registering interactions. The parameter must be a correct channel identifier.

`name`

Name of the reaction. Used only for tracing.

The parameter can be a NULL pointer. If so, name of the interaction is considered to be equal to the reaction name reactionID.

`reactionID`

Reaction identifier of the registered interaction.

`data`

Data from the target system in model representation.

Data type of the parameter must coincide with data type of reactionID reaction return value.

Additional information

`T`he main properties of interaction are reaction name `reactionID` and data data, received during the interaction. Data type of the received data must coincide with data type of the reaction return value.

Time marks, time intervals, and interactions channels are used by the test system for ordering registered interactions.

If data received from the target system cannot be converted to model representation, the test system

must be informed about receiving incorrect reaction by means of registerWrongReaction function.

Header file: ts/register.h

registerReactionWithTimeMark

registerReactionWithTimeMark function is intended for registration of reactions, received from the target system, specifying time mark of occurrence moment.

```
void registerReactionWithTimeMark( ChannelID chid, const char* name,
SpecificationID reactionID, Object* data, TimeMark mark );
```

Parameters

chid

Interactions channel identifier to be used by the test system when automatically registering interactions. The parameter must be a correct channel identifier.

name

Name of the reaction. Used only for tracing.

The parameter can be a NULL pointer. If so, name of the interaction is considered to be equal to the reaction name reactionID.

reactionID

Reaction identifier of the registered interaction.

data

Data from the target system in model representation.

Data type of the parameter must coincide with data type of reactionID reaction return value.

mark

Time mark of occurrence moment.

Additional information

The main properties of interaction are reaction name reactionID and data data, received during the interaction. Data type of the received data must coincide with data type of the reaction return value.

Time marks, time intervals, and interactions channels are used by the test system for ordering registered interactions.

If data received from the target system cannot be converted to model representation, the test system must be informed about receiving incorrect reaction by means of registerWrongReaction function.

Header file: ts/register.h

registerReactionWithTimeInterval

registerReactionWithTimeInterval function is intended for registration of reactions, received from the target system, specifying time interval of its occurrence.

```
void registerReactionWithTimeInterval
     ( ChannelID chid
     , const char* name
     , SpecificationID reactionID
     , Object* data
     , TimeInterval interval
     );
```

Parameters

chid

Interactions channel identifier to be used by the test system when automatically registering interactions. The parameter must be a correct channel identifier.

name

Name of the reaction. Used only for tracing.

The parameter can be a NULL pointer. If so, name of the interaction is considered to be equal to the reaction name reactionID.

reactionID

Reaction identifier of the registered interaction.

data

Data from the target system in model representation.

Data type of the parameter must coincide with data type of reactionID reaction return value.

interval

Time interval of interaction occurrence. The interaction is considered to be occurred somewhere within the interval, not occupied the whole interval.

Additional information

The main properties of interaction are reaction name reactionID and data data, received during the interaction. Data type of the received data must coincide with data type of the reaction return value.

Time marks, time intervals, and interactions channels are used by the test system for ordering registered interactions.

If data received from the target system cannot be converted to model representation, the test system must be informed about receiving incorrect reaction by means of registerWrongReaction function.

Header file: ts/register.h

registerWrongReaction

registerWrongReaction function is intended to notify the test system about receiving incorrect reaction, that cannot be converted to model representation. Reaction is an interaction initiated by the target system.

```
void registerWrongReaction( const char* info );
```

Parameters

info

Description of incorrect reaction, used when analyzing test results.

The parameter can be a NULL pointer.

Additional information

`registerWrongReaction` function is intended to notify the test system about receiving incorrect reaction, that cannot be converted to model representation. Reaction is an interaction initiated by the target system.

After registering incorrect reaction, the test system terminates analysis of current test action with negative verdict.

Header file: ts/register.h

registerStimulusWithTimeInterval

`registerStimulusWithTimeInterval` function is intended for registering stimulus that was not registered automatically by the test system. Stimulus is an interaction with the target system initiated by the test.

```
void registerReactionWithTimeInterval
     ( ChannelID chid
     , const char* name
     , SpecificationID stimulusID
     , TimeInterval interval
     , ...
     );
```

Parameters

chid

Interactions channel identifier to be used by the test system when automatically registering interactions. The parameter must be a correct channel identifier.

name

Name of the reaction. Used only for tracing.

The parameter can be a NULL pointer. If so, name of the interaction is considered to be equal to the reaction name reactionID.

stimulusID

Identifier of the specification function registered interaction corresponds to.

interval

Time interval of interaction occurrence. The interaction is considered to be occurred

somewhere within the interval, not occupied the whole interval.

arguments

Additional arguments strictly in the following order:

1. List of specification function parameters values before its invocation.
2. List of specification function parameters values after its invocation.

3. Value returned by the specification function (if data type of result value is not void).

Additional information

`registerStimulusWithTimeInterval` function is intended for registering stimulus that was not registered automatically by the test system. All stimuli, initiated by means of specification functions calls within the test system process, are registered automatically. Thus the only stimuli to be registered manually, are those initiated outside the test system process or by other means than specification function call.

The main properties of interaction are reaction name `reactionID` and data, passed via additional arguments.

Time marks, time intervals, and interactions channels are used by the test system for ordering registered interactions.

If data received from the target system cannot be converted to model representation, the test system must be informed about incorrect behavior during interaction by means of `registerWrongReaction` function within the test system main process.

Header file: ts/register.h

13.3 Catcher functions registering service

For the convenience of registering deferred reactions, the test system provides catcher functions registering service. The service is organized as follows. Special catcher functions are registered in the test system, to be invoked after stabilization period of each test action. Till the stabilization period the target system must initiate all requested reactions and come to a stable state. Catcher functions must then gather all information about received reactions and register them in interaction registrar.

To register catcher functions the following functions are intended:

- registerReactionCatcher
- unregisterReactionCatcher
- unregisterReactionCatchers

Catcher function is parameter of each of these functions, ReactionCatcherFuncType is type of these functions.

ReactionCatcherFuncType

ReactionCatcherFuncType data type is used for registration of catcher functions in the test system. .

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

Additional information

Header file: ts/timemark.h

registerReactionCatcher

RegisterReactionCatcher function registers the catcher function in the test system along with its auxiliary data.

```
void registerReactionCatcher(
        ReactionCatcherFuncType catcher,
        void*                   par
                            );
```

Parameters

catcher

Pointer to a catcher function.

The parameter must not be a NULL pointer.

par

Auxiliary data of the function being registe. The parameter can be a NULL pointer.

Additional information

registerReactionCatcher function registers the catcher function in the test system along with

its auxiliary data.

When the test system invokes the catcher function, it passes the auxiliary data to it as its parameter. The sane catcher function can be registered in the test system several times with different data. If so, the function will be invoked appropriate number of times with different parameter value.

Header file: ts/register.h

`unregisterReactionCatcher`

unregisterReactionCatcher function removes a record of the specified catcher function with the specified auxiliary data from the test system.

```
bool unregisterReactionCatcher(
            ReactionCatcherFuncType catcher,
            void*                   par
                        );
```

Parameters

`catcher`

Pointer to a catcher function.

The parameter must not be a NULL pointer.

`par`

Auxiliary data of the function being registe. The parameter can be a NULL pointer.

Return value

The function returns false if the specified function with the specified auxiliary data was not registered before, and true otherwise.

Additional information

`unregisterReactionCatcher` function removes a record of the specified catcher function with the specified auxiliary data from the test system.

To remove all registration records about the specified catcher function,`unregisterReactionCatchers` function is intended.

Header file: ts/register.h

`UnregisterReactionCatchers`

`unregisterReactionCatchers` function removes all records of the specified catcher function from the test system.

```
bool unregisterReactionCatchers( ReactionCatcherFuncType catcher );
```

Parameters

`catcher`

Pointer to a catcher function.

`The parameter must not be a NULL pointer.`

Return value

The function returns false if the specified function was not registered before, and true otherwise.

Additional information

`unregisterReactionCatchers` function removes all records of the specified catcher function from the test system.

To remove a records about the catcher function with specific auxiliary data, `unregisterReactionCatcher` function is intended.

Header file: ts/register.h

14 Library of specification data types

Library of specification data types contains standard functions for dealing with specification data types (creation, copying, comparing, stringifying) as well as predefined specification data types for standard data types of C language (`char`, `short`, `int`, `long`, `float`, `double`, `void`*, strings `char`*), for complex numbers, for data type with a single value, and for container data types (list, set, map).

14.1 Standard functions

Standard functions can be used for any specification references. The result of function execution depends on specification reference data type. For example, result of comparing two specification references of different data types is always negative, and result of comparing two specification references of the same data type is defined by comparing function, specified in the data type definition.

Data type of specification reference is defined by a pointer to *specification data type descriptor*. Descriptor constant always has the name, consisting of the name of specification data type and a prefix `type_`:

```
const Type type_specification_data_type_name;
```

- **Function of Creating references**
  The function creates object of the passed type and returns a pointer to it.
- **Function of getting reference's type**
  The function identifies the type of the specification reference and returns a descriptor of this type.
- **Function of copying values by references**
  The function copies the content of the object by reference *src* to the content of the object by reference *dst*.
- **Function of cloning object**
  The function helps to get a new object with the same content – to clone the object.
- **Function of comparing values by references**
  The function compares object passed to it by specification references returning an integer result.
- **Function of detecting equivalence of values by references**
  The function compares objects passed to it by specification references returning a logical value.
- **Function of building a string representation of value by reference**
  The function returns a pointer to a `String` type value – a string representation of specification type.
- **Function of building XML representation of value by reference|outline**
  The function returns a pointer to a String type value – XML representation of specification type.

14.1.1 Function of Creating references

The function creates object of the passed type and returns a pointer to it.

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

Parameters

type

A pointers to specification data type descriptor.

...

Data type initialization parameters

Return value

The function returns a specification reference to the created object.

Additional information

Initialization parameters must meet the type: parameters for all predefined data types are described in the relevant part, and the parameters for user-defined data types are specified in the data type definition.

```
Integer* ref = create(&type_Integer, 28); // ref → 28
```

In the above example a reference to library type `Integer` (specification analogue of `int` data type) is created and initialized.

The usage of function `create` can cause errors because of the usage of not typified argument list.... To except such errors it is recommended to create function `create_name_of_specifiction_type` for each specification data type and to call create out of it.

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

IntPair* create_IntPair(int a, int b)
{
  return create(&type_IntPair, a, b);
}
```

If you use `create` out of function the name of which has `create_` prefix, the compiler gives out warning:

```
warning: call create() out of create_... function
```

14.1.2 Function of getting reference's type

The function identifies the type of the specification reference and returns a descriptor of this type.

```
const Type *type( Object* ref )
```

Parameters

ref

A specification reference to specification type object

Return value

The function returns a pointer to the descriptor constant of the specification data type, referenced by *ref* pointer.

```
Integer* ref = create(&type_Integer, 28);

if (type(ref) == &type_Integer) // true
```

14.1.3 Function of copying values by references

The function copies the content of the object by reference src to the content of the object by reference *dst*.

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

Parameters

src

Reference to the object for coping.

dst

Reference to the object in which to copy.

Additional information

References must be of the same data type, i.e. they must have equal data type descriptors. Otherwise the application will be terminated in run time with an error message.

```
Integer* ref1 = create(&type_Integer, 28); // ref1 → 28

Integer* ref2 = create(&type_Integer, 47); // ref2 → 47

copy(ref1,ref2); // ref1 → 28, ref2 → 28
```

In the above example references *ref1* and *ref2* are referencing to different values of `Integer` specification data type after their initialization. After invocation of `copy()` function, value, referenced by *ref2* pointer, become equal to value, referenced by *ref1* pointer.

14.1.4 Function of cloning object

The function helps to get a new object with the same content – to clone the object.

```
Object* clone( Object* ref )
```

Parameters

ref

A reference to the object for cloning.

Additional information

The function allocates memory for data type value, referenced by *ref* pointer, initializes the allocated memory by a value, equal to value referenced by *ref* pointer, and returns a pointer to the allocated and initialized memory.

```
Integer* ref1 = create(&type_Integer, 28);

String* ref2 = clone(ref1);
```

ref1 is initialized to value `28`. Values, referenced by *ref1* and *ref2* pointers, become equal after invocation of `clone()` function.

14.1.5 Function of comparing values by references

The function compares object passed to it by specification references returning an integer result.

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

Parameters

left

A specification reference to the first compared object.

right

A specification reference to the second compared object.

Return value

The function returns zero if the values, referenced by the given references, are equal.

If the values are not equal, the function returns a nonzero value, that can be interpreted differently depending on the data type. For example, for `String` library data type, the result has the same meaning as `strcmp()` function for `char*` C data type.

If the parameters have incomparable data types (i.e. the references have different data types and data type of one reference is not a subtype of another (see Invariants of data types)), the function returns nonzero value.

If one of the references is `NULL`, and the other is not, the function returns a nonzero value.

If both references are `NULL`, the function returns zero.

```
if (!compare(ref1,ref2)) {/* values are equal */
...
}
else {/* values are not equal */

...
}
```

14.1.6 Function of detecting equivalence of values by references

The function compares objects passed to it by specification references returning a logical value.

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

Parameters

self

A specification reference to the first comparing object.

ref

A specification reference to the second comparing object.

Return value

The function returns `true` if the values, referenced by the given references, are equal, and `false` otherwise.

If the parameters has different data type, the function returns `false`.

If one of the references is `NULL`, and the other is not, the function returns `false`.

If the both references are `NULL`, the function returns true.

```
if (equals(ref1,ref2)) {/* values are equal */

...
}
else {/* values are not equal */
...
}
```

14.1.7 Function of building a string representation of value by reference

The function returns a pointer to a String type value – a string representation of specification type.

```
String* toString(Object* ref)
```

Parameters

ref

A reference to the specification type object.

Return value

The function returns a pointer to the value of `String` data type—specification representation of character string data type.

```
Integer* ref = create(&type_Integer, 28); // ref → 28

String* str = toString(ref); // str → "28"

printf("*ref == '%s'\n", toCharArray_String(str));
```

14.1.8 Function of building XML representation of value by reference

The function returns a pointer to a String type value – XML representation of specification type.

```
String* toXML(Object* ref)
```

Parameters

ref

A reference to the specification type object.

Return value

The function returns a reference to the String type object, which contain XML representation of

passed to the function object.

```
Integer* ref = create(&type_Integer, 28); // ref → 28
String* str = toXML(ref);
// str → "<object kind=\"spec\" type=\"Integer\" text=\"28\"/>"
printf("*ref == '%s'\n", toCharArray_String(str));
```

14.2 Predefined specification types

Predefined data types are convenient in specifications (for example, for implementation state modeling) as they provide ready for use, universal, guaranteed faultless functionality.

14.2.1 Char

Type `Char` is a specification analogue of C build-in data type `char`.

Additional information

Functions, defined for the type `Char`:

- creating specification reference;
- getting a value of the type `char`.

Header file: atl/char.h

create_Char

The function creates a specification reference of the type `Char`.

```
Char* create_Char( char c )
```

Parametrs

c

Initializing value.

Return value

A specification reference of the type `Char` to the created object.

Additional information

This function is defined along with the standard function `create`.

```
Char* ch1 = create(&type_Char, 'a');

Char* ch2 = create_Char('a');
```

The two shown methods of creating a specification reference are functionally equivalent, but the use of `create` is not type-safe and cause a compiler warning (see function `create` definition).

value_Char

The function gets a `char value`, contained in a specification type.

```
char value_Char( Char* c )
```

Parametrs

c

A specification reference to the value of the type `Char`.

Return value

A symbol, which is a value of specification reference *c*.

Additional information

This value can be also referred to using dereferencing of specification reference:

```
Char* ch = create_Char('a');

char  val = *ch;
```

Function value_Char() ensures a run-time type control and a pointer Object* can be passed to it without preliminary casting:

```
List* l;

char val;

...

val = value_Char(get_List(l,0)); // Object* get_List(List*,int)
```

However if there is no confidence in the right type of the passed reference an explicit checking with a standard function type()can be needed:

```
Object* o = get_List(l,0));

if (type(o) == &type_Char) {
  val = value_Char(o);
}
```

14.2.2 Integer and UInteger

Types `Integer` and `UInteger` are specification analogues of C built-in data types `int` and `unsigned int`.

Additional information

The functions, defined for the types `Integer` and `UInteger`:

- creating specification reference;
- getting a value of the type `int` or `unsigned int`.

Header file: atl/Integer.h

create_Integer, create_UInteger

The functions creates specification references of the types `Integer` and `UInteger`.

```
Integer* create_Integer( int i )

UInteger* create_UInteger( unsigned int i )
```

Parametrs

i

Initializing value.

Return value

A specification reference of the type `Integer` (`UInteger`) to the created object.

Additional information

This function is defined along with the standard function `create`.

```
Integer* i1 = create(&type_Integer, -28);

Integer* i2 = create_Integer(-28);


UInteger* i1 = create(&type_UInteger, 28);

UInteger* i2 = create_UInteger(28);
```

The two shown methods of creating a specification reference are functionally equivalent, but the use of create is not type-safe and cause a compiler warning (see function create definition).

value_Integer, value_UInteger

The function gets an `int value`, contained in a specification type.

```
int value_Integer ( Integer* i )

unsigned int value_UInteger ( UInteger* i )
```

Parametrs

i

A specification reference to the value of the type `Integer` or `UInteger`.

Additional information

This value can be also referred to using dereferencing of specification reference:

```
Integer* i = create_Integer(-28);

int  val = *i;
```

Function value`_Integer()` ensures a run-time type control and a pointer Object* can be passed to it without preliminary casting:

```
List* l;

int val;

...

val = value_Integer(get_List(l,0)); // Object* get_List(List*,int)
```

However if there is no confidence in the right type of the passed reference an explicit checking with a standard function type()can be needed:

```
Object* o = get_List(l,0));

if (type(o) == &type_Integer) {
 val = value_Integer(o);
}
```

14.2.3 Short and Ushort

Types `Short` and `UShort` are specification analogues of C built-in data types `short` and `unsigned short`.

Additional information

The functions, defined for the types `Short` and `UShort`:

creating specification reference;

getting a value of the type `short` or `unsigned short`.

Header file: atl/short.h

create_Short, create_UShort

The functions creates specification references of the types `Short` and `UShort`.

```
Short* create_Short ( short i )

UShort* create_UShort ( unsigned short i )
```

Parametrs

i

Initializing value.

Return value

A specification reference of the type `Short` (`UShort`) to the created object.

Additional information

This function is defined along with the standard function `create`.

```
Short* i1 = create(&type_Short, -28);

Short* i2 = create_Short(-28);

UShort* i1 = create(&type_UShort, 28);

UShort* i2 = create_UShort(28);
```

The two shown methods of creating a specification reference are functionally equivalent, but the use of create is not type-safe and cause a compiler warning (see function create definition).

value_Short, value_UShort

The function gets a `short value`, contained in a specification type.

```
short value_Short ( Short* i )

unsigned short value_UShort ( UShort* i )
```

Parametrs

i

A specification reference to the value of the type `Short` or `UShort`.

Additional information

This value can be also referred to using dereferencing of specification reference:

```
Short* i = create_Short(-28);

short  val = *i;
```

Function value`_Short()` ensures a run-time type control and a pointer Object* can be passed to it without preliminary casting:

```
List* l;

short val;

...

val = value_Short(get_List(l,0)); // Object* get_List(List*,int)
```

However if there is no confidence in the right type of the passed reference an explicit checking with a standard function type()can be needed:

```
Object* o = get_List(l,0));

if (type(o) == &type_Short) {
  val = value_Short(o);
}
```

14.2.4 Long and Ulong

Types `Long` and `ULong` are specification analogues of C built-in data types `long int` and `unsigned long int`.

Additional information

Functions, defined for the type `Long`:

- creating specification reference;
- getting a value of the type `long int` or `unsigned long int`.

Header file: atl/long.h

create_Long, create_ULong

The functions creates specification references of the types `Long` and `ULong`.

```
Long* create_Long( long i )

ULong* create_ULong ( unsigned long i )
```

Parametrs

i

Initializing value.

Return value

A specification reference of the type `Long` (`ULong`) to the created object.

Additional information

This function is defined along with the standard function `create`.

```
Long* i1 = create(&type_Long, -28);

Long* i2 = create_Long(-28);

ULong* i1 = create(&type_ULong, 28);

ULong* i2 = create_ULong(28);
```

The two shown methods of creating a specification reference are functionally equivalent, but the use of create is not type-safe and cause a compiler warning (see function create definition).

value_Long, value_ULong

The function gets a `long value`, contained in a specification type.

```
long value_Long ( Long* i )

unsigned long value_ULong ( ULong* i )
```

Parametrs

i

A specification reference to the value of the type `Long` or `ULong`.

Additional information

This value can be also referred to using dereferencing of specification reference:

```
Long* i = create_Long(-28);

long  val = *i;
```

Function value`_Long()` ensures a run-time type control and a pointer Object* can be passed to it without preliminary casting:

```
List* l;

long val;

...

val = value_Long(get_List(l,0)); // Object* get_List(List*,int)
```

However if there is no confidence in the right type of the passed reference an explicit checking with a standard function type()can be needed:

```
Object* o = get_List(l,0));

if (type(o) == &type_Long) {
  val = value_Long(o);
}
```

14.2.5 Float

Type `Float` is a specification analogue of C built-in data type `float`.

Additional information

Functions, defined for the type Float:

- creating specification reference;
- getting a value of the type `float`.

Header file: atl/float.h

create_Float

The function creates a specification reference of the type `Float`.

```
Float* create_Float( float f )
```

Parametrs

f

Initializing value.

Return value

A specification reference of the type `Float` to the created object.

Additional information

This function is defined along with the standard function `create`.

```
Float* f1 = create(&type_Float, 3.14);

Float* f2 = create_Float(3.14);
```

The two shown methods of creating a specification reference are functionally equivalent, but the use of create is not type-safe and cause a compiler warning (see function create definition).

value_Float

The function gets a `float value`, contained in a specification type.

```
float value_Float( Float* f )
```

Parametrs

f

A specification reference to the value of the type `Float`.

Additional information

This value can be also referred to using dereferencing of specification reference:

```
Float* f = create_Float(3.14);

float  val = *f;
```

Function value_Float() ensures a run-time type control and a pointer Object* can be passed to it without preliminary casting:

```
List* l;

float val;

...

val = value_Float(get_List(l,0)); // Object* get_List(List*,int)
```

However if there is no confidence in the right type of the passed reference an explicit checking with a standard function type()can be needed:

```
Object* o = get_List(l,0));

if (type(o) == &type_Float) {
  val = value_Float(o);
}
```

14.2.6 Double

Type `Double` is a specification analogue of C built-in data type `double`.

Additional information

Functions, defined for the type `Double`:

- creating specification reference;
- getting a value of the type `double`.

Header file: atl/double.h

create_Double

The function creates a specification reference of the type `Double`.

```
Double* create_Double( double d )
```

Parametrs

d

Initializing value.

Return value

A specification reference of the type `Double` to the created object.

Additional information

This function is defined along with the standard function `create`.

```
Double* d1 = create(&type_Double, 3.14);

Double* d2 = create_Double(3.14);
```

The two shown methods of creating a specification reference are functionally equivalent, but the use of create is not type-safe and cause a compiler warning (see function create definition).

value_Double

The function gets a `double value`, contained in a specification type.

```
double value_Double( Double* d )
```

Parametrs

d

A specification reference to the value of the type `Double`.

Additional information

This value can be also referred to using dereferencing of specification reference:

```
Double* d = create_Double(3.14);

double *p = (double*)d;
```

Function value_Double() ensures a run-time type control and a pointer Object* can be passed to it without preliminary casting:

```
List* l;

double val;

...

val = value_Double(get_List(l,0)); // Object* get_List(List*,int)
```

However if there is no confidence in the right type of the passed reference an explicit checking with a standard function type()can be needed:

```
Object* o = get_List(l,0));

if (type(o) == &type_Double) {
  val = value_Double(o);
}
```

14.2.7 VoidAst

Type `VoidAst` is a specification analogue of C built-in data type `void`.

Additional information

Functions, defined for the type `VoidAst`:

- creating specification reference;
- getting a value of the type void.

Header file: atl/voidast.h

create_VoidAst

The function creates a specification reference of the type `VoidAst`.

```
VoidAst* create_VoidAst( void* v )
```

Parametrs

v

Initializing value.

Return value

A specification reference of the type `VoidAst` to the created object.

Additional information

This function is defined along with the standard function `create`.

```
VoidAst* v1 = create(&type_VoidAst, NULL);

VoidAst* v2 = create_VoidAst(NULL);
```

The two shown methods of creating a specification reference are functionally equivalent, but the use of create is not type-safe and cause a compiler warning (see function create definition).

value_VoidAst

The function gets a `void*` value, contained in a specification type.

```
void* value_VoidAst( VoidAst* v )
```

Parametrs

v

A specification reference to the value of the type `VoidAst`.

Additional information

This value can be also referred to using dereferencing of specification reference:

```
VoidAst* v = create_VoidAst(NULL);

voidast  val = *v;
```

Function value_VoidAst ensures a run-time type control and a pointer Object* can be passed to it without preliminary casting:

```
List* l;

void* val;

...

val = value_VoidAst(get_List(l,0)); // Object* get_List(List*,int)
```

However if there is no confidence in the right type of the passed reference an explicit checking with a standard function type()can be needed:

```
Object* o = get_List(l,0));

if (type(o) == &type_VoidAst) {
  val = value_VoidAst(o);
}
```

14.2.8 Unit

Type `Unit` is a specification data type with the only one value : two `nonzero` specification references of `Unit` data type are always equal.

Additional information

Functions, defined for the type `Unit`:

- creating specification reference

Header file: atl/unit.h

create_Unit

The function creates a specification reference of the type `Unit`.

```
Unit* create_Unit(void)
```

Return value

A specification reference of the type `Unit` to the created object.

Additional information

This function is defined along with the standard function `create`.

```
Unit* u1 = create(&type_Unit);

Unit* u2 = create_Unit();
```

The two shown methods of creating a specification reference are functionally equivalent, but the use of create is not type-safe and cause a compiler warning (see function create definition).

14.2.9 BigInteger

Type `BigInteger` is used for representation of integers of any size

Header file: atl/bigint.h

add_BigInteger
The function returns a sum of two `BigInteger` numbers.

create_BigInteger
The function creates a `BigInteger` specification reference, which represents input integer.

divide_BigInteger
The function returns a result of division of two `BigInteger numbers`.

intValue_BigInteger
The function returns `int value`, contained in a specification type `c`).

longValue_BigInteger
The function returns `long value`, contained in a specification type (if this value can be put in `long`).

multiply_BigInteger
The function returns a result of multiplication of two `BigInteger numbers`.

negate_BigInteger
The function returns a negation of `BigInteger number`.

power_BigInteger
The function returns a result of integer involution of `BigInteger` number.

remainder_BigInteger
The function returns a remainder in division of two `BigInteger numbers`.

subtract_BigInteger
The function returns a result of subtraction of two `BigInteger numbers`.

valueOf_BigInteger
The function creates a `BigInteger` specification reference, which represents an integer in a form string.

add_BigInteger

The function returns a sum of two `BigInteger` numbers.

```
BigInteger* add_BigInteger( BigInteger* n1, BigInteger* n2 )
```

Parametrs

n1

First summand.

n2

Second summand.

Return value

A specification reference of the type `BigInteger` to a new object — a sum.

create_BigInteger

The function creates a specification reference of the type `BigInteger`, which represents input integer.

```
BigInteger* create_BigInteger( int i )
```

Parametrs

i

Initializing value.

Return value

A specification reference of the type `BigInteger` to the created object.

Additional information

This function is defined along with the standard function `create`.

```
BigInteger* i1 = create(&type_BigInteger, -28);

BigInteger* i2 = create_BigInteger(-28);
```

The two shown methods of creating a specification reference are functionally equivalent, but the use of create is not type-safe and cause a compiler warning (see function create definition).

divide_BigInteger

The function returns a result of division of two `BigInteger numbers`.

```
BigInteger* divide_BigInteger( BigInteger* n1, BigInteger* n2 )
```

Parametrs

n1

Dividend.

n2

Divisor.

Return value

A specification reference of the type `BigInteger` to a new object — a quotient.

intValue_BigInteger

The function returns an `int value`, contained in a specification type (if this value can be put in `int`).

```
int intValue_BigInteger ( BigInteger* i )
```

Parametrs

i

A specification reference to the value of the type `BigInteger`.

Additional information

If `BigInteger value can't be put in int`, program stop running using `assertion`.

longValue_BigInteger

`The function returns long value`, contained in a specification type (if this value can be put in `long`).

```
long value_BigInteger ( BigInteger* i )
```

Parametrs

i

A specification reference to the value of the type `BigInteger`.

Additional information

If `BigInteger value can't be put in int`, program stop running using `assertion`.

multiply_BigInteger

The function returns a result of multiplication of two `BigInteger numbers`.

```
BigInteger* multiply_BigInteger( BigInteger* n1, BigInteger* n2 )
```

Parametrs

n1

First factor.

n2

Second factor.

Return value

A specification reference of the type `BigInteger to a new object` — a product.

negate_BigInteger

The function returns a negation of `BigInteger number`.

```
BigInteger* negate_BigInteger( BigInteger* n )
```

Parametrs

n

Argument of negation.

Return value

A specification reference of the type `BigInteger` to a new object — a negation.

power_BigInteger

The function returns a result of integer involution of `BigInteger` number.

```
BigInteger* divide_BigInteger( BigInteger* n1, int n2 )
```

Parametrs

n1

Base of power.

n2

Exponent.

Return value

A specification reference of the type `BigInteger` to a new object — a result of involution.

remainder_BigInteger

The function returns a remainder in division of two `BigInteger numbers`.

```
BigInteger* remainder_BigInteger( BigInteger* n1, BigInteger* n2 )
```

Parametrs

n1

Dividend.

n2

Divisor.

Return value

A specification reference of the type `BigInteger` to a new object — a remainder.

subtract_BigInteger

`The function returns a result of subtraction of two BigInteger numbers.`

```
BigInteger* subtract_BigInteger( BigInteger* n1, BigInteger* n2 )
```

Parametrs

n1

Minuend.

n2

Deduction.

Return value

A specification reference of the type `BigInteger` to a new object — a difference.

valueOf_BigInteger

The function creates a specification reference of the type `BigInteger`, which represents an integer in a string form.

```
BigInteger* valueOf_BigInteger( String* str )
```

Parametrs

`str`

String type value of the number.

Return value

A specification reference of the type `BigInteger` to the created object.

14.2.10 Complex

Type `Complex` is intended for representation of complex numbers.

Additional information

The usual comparing rules are applied to specification references of `Complex` data type ($(re_1, im_1) = (re_2, im_2) \leftrightarrow re_1 = re_2, im_1 = im_2$). String representation looks like ($re + im*i$).

The base type of type `Complex` is the following structure:

```
struct {

  double re;

  double im;

};
```

There are no special functions to access real and imaginary parts of the complex number; dereferencing should be used:

```
Complex* c = create_Complex(1.4, -0.6);

double re = c->re;

double im = c->im;
```

The functions, defined for the type `Complex`:

creating specification reference;

Header file: atl/complex.h

create_Complex

The function creates a specification reference of the type `Complex`.

```
Complex* create_Complex ( double re, double im )
```

Parametrs

re

Real part of complex number.

im

Imaginary part of complex number.

Return value

A specification reference of the type `Complex` to the created object.

Additional information

This function is defined along with the standard function `create`.

```
Complex* c1 = create(&type_Complex, 1.4, -0.6);

Complex* c2 = create_Complex(1.4, -0.6);
```

The two shown methods of creating a specification reference are functionally equivalent, but the use of create is not type-safe and cause a compiler warning (see function create definition).

14.2.11 String

Type String is intended for representation of string characters.

Additional information

The way strings are represented in C language, as array of `char`, cannot be kept within the concept of allowable data type. Thus, `strings` can be conveniently represented by this specification data type everywhere the allowable data type is required.

Specification references to String data type are compared by normal rules (the same way as `strcmp` function does). Character positions are numbered from 0.

Specification strings can be handled like usual C strings, taking into account that the value of the string should not be changed. To access C string value, `toCharArray_String` function is used. This function returns a pointer to character array within the specification data type.

At the same time specification strings provides a lot of convenient functions themselves. All that functions can accept only non-zero String references.

Header file: atl/string.h

`create_String`
The function creates a specification reference of the type `String`.

`charAt_String`
The function returns a character in the given *`index`* position.

`concat_String`
The function returns a `String` specification reference to a concatenation of two strings.

`startsWith_String`
The function checks if the string *`self`* begins with *`suffix`*.

`startsWithOffset_String`
The function checks if the sub-string *`fix`* `contained in the string` *`self`* in the given position.

`endsWith_String`
The function checks if the string *`self`* ends with the string *`suffix`*.

`indexOfChar_String`
The function searches the given character in the string and returns a number of the first found position.

`indexOfCharFrom_String`
The function searches the given symbol in the string starting from *`fromIndex`* position and return a number of the first found position.

`indexOfString_String`
Searches the given sub-string in a string and returns a number of the first found position.

`indexOfStringFrom_String`
Searches the given sub-string in a string beginning from *`fromIndex`* position and returns a number of the first found position.

`lastIndexOfChar_String`
Searches the given symbol in the string right to left and returns a number of the first found position.

`lastIndexOfCharFrom_String`

Searches the given symbol in the string right to left starting from the *fromIndex* position and returns a number of the first found position.

lastIndexOfString_String
Searches the given sub-string in the string right to left and returns a number of the first found position.

lastIndexOfStringFrom_String
Searches the given sub-string in a string right to left starting from the *fromIndex* position and returns a number of the first found position.

length_String
The function returns the length of a string.

regionMatches_String, regionMatchesCase_String
Checks whether the sub-string of string *self* matches the sub-string of the string *other*. The function `regionMatchesCase_String` takes letters case into account, the function `regionMatches_String` has an additional parameter *ignoreCase*, which let to ignore letters case or to take it into account.

replace_String
Replaces in the given string all the *oldChar* symbols to the *newChar* symbols.

substringFrom_String
The function returns sub-string with all the symbols from the *beginIndex* position to the end.

`substring_String`
The function returns sub-string with all the symbols from the *beginIndex* position to the *endIndex* inclusive.

toLowerCase_String
The function converts letters to the lower case.

toUpperCase_String
The function converts letters to the upper case.

toCharArray_String
The function returns a C-string corresponding to the given specification string.

trim_String
The function returns a string, constructed from a self string by removing space characters from the beginning and the ending of the string.

format_String
The function returns a specification string, corresponding to output of `printf` function, invoked with the same parameters

vformat_String
The function returns a specification string, corresponding to output of `vprintf` function, invoked with the same parameters.

valueOfBool_String
The function returns a string representation of `bool` value.

valueOfChar_String
The function returns a string representation of `char value`.

valueOfShort_String
The function returns a string representation of `short value`.

valueOfUShort_String
The function returns a string representation of `unsigned short value` .

valueOfInt_String

The function returns a string representation of `int value`.

valueOfUInt_String
The function returns a string representation of `unsigned int value`.

valueOfLong_String
The function returns a string representation of `long value`.

valueOfULong_String
The function returns a string representation of `unsigned long value`.

valueOfFloat_String
The function returns a string representation of `float value`.

valueOfDouble_String
The function returns a string representation of `double value`.

valueOfPtr_String
The function returns a string representation of `void*` value.

valueOfObject_String
The function returns a string representation of a specification type.

valueOfBytes_String
The function returns a string hexadecimal representation of `p` byte array of `l` length.

create_String

The function creates a specification reference of the type `String`.

```
String* create_String ( const char *cstr )
```

Parametrs

cstr

Initializing value.

Return value

A specification reference of the type `String` to the created object.

Additional information

This function is defined along with the standard function `create`.

```
String* s1 = create(&type_String, "a string");

String* s2 = create_String("a string");
```

The two shown methods of creating a specification reference are functionally equivalent, but the use of create is not type-safe and cause a compiler warning (see function create definition).

charAt_String

The function returns a character in the given *index* position.

```
char charAt_String( String* self, int index )
```

Parametrs

self

A string a character of which is needed to be defined.

index

A number of the position inside the string.

Return value

A character found in the *self* string in the *index* position.

Additional information

A number of the position must be within the string. A numeration starts with 0; A position of the last meaningful symbol determined with the function length_String as length_String(self) - 1.

```
String* s = create_String("abracadabra");

printf("%c\n", charAt_String(s,5));
```

```
c
```

concat_String

The function returns a String specification reference to a concatenation of two strings.

```
String *concat_String( String* str1, String* str2 )
```

Parametrs

str1

A first part of concatenation.

str2

A second part of concatenation.

Return value

A specification string get by a concatenation of two strings *str1* and *str2*.

Additional information

```
String* s1 = create_String("abra");

String* s2 = create_String("cadabra");

String* s  = concat_String(s1,s2);

printf("%s\n", toCharArray_String(s));
```

```
abracadabra
```

startsWith_String

The function checks if one string begins with the other.

```
bool startsWith_String( String *self, String *prefix )
```

Parametrs

self

A string the beginning of which is needed to be checked.

prefix

A sought-for beginning of a string.

Return value

`True`, if a *self* string begins with a *prefix* sub-string. `false` in any other case

Additional information

If a prefix string is empty, true is returned. If the length of a *prefix* string is bigger than the length of *self* then `false` is returned.

```
String* s  = create_String("abracadabra");

String* s1 = create_String("abra");

String* s2 = create_String("cadabra");

printf("1) %d\n2) %d\n ",

  startsWith_String(s,s1),

  startsWith_String(s,s2)

);
```

```
1) 1
2) 0
```

startsWithOffset_String

The function checks if the sub-string *fix* contained in the string *self* in the given position.

```
bool startsWithOffset_String
     ( String *self, String *fix, int toffset )
```

Parametrs

self

A string where sub-string should be searched.

fix

A sought-for sub-string.

toffset

A position in the given string.

Return value

`true`, if in the string *self* in the position *toffset* begins the sub_string *prefix*. `false` in any other case.

Additional information

If the position is negative or is greater than the *self* string's length, `false` is returned.If string *prefix* is empty, `true is returned`. If the length of string *prefix* is greater than the length of *self*, `false is returned`.

```
String* s  = create_String("abracadabra");

String* s1 = create_String("abra");

printf("1) %d\n2) %d\n",

  startsWithOffset_String(s,s1,7),

  startsWithOffset_String(s,s2,8)

);
```

```
1) 1
2) 0
```

endsWith_String

The function checks if the string *self* ends with the string *suffix*.

```
bool endsWith_String( String *self, String *suffix )
```

Parametrs

self

A string the end of which is needed to be checked.

suffix

A sought-for ending of a string.

Return value

`true`, if the string *self* ends with the string *suffix*. `false` in any other case.

Additional information

If the string *suffix* is empty, the function returns `true`. If the length of the string *suffix* is greater than the length of *self*, the function returns `false`.

```
String* s  = create_String("abracadabra");

String* s1 = create_String("abr");
```

```
String* s2 = create_String("cadabra");

printf( "1) %d\n2) %d\n "
      , endsWith_String(s,s1)
      , endsWith_String(s,s2)
      );
```

```
1) 0
2) 1
```

indexOfChar_String

The function searches the given character in the string and returns a number of the first found position.

```
int indexOfChar_String( String* self, int ch )
```

Parametrs

self

A string where a symbol should be searched.

ch

A sought-for symbol.

Return value

A number of the first symbol *ch in a string self*. If the symbol isn't found -1 is returned. For symbols with code `0` the function always returns `-1`.

Additional information

```
String* s = create_String("abracadabra");

printf( "1) %d\n2) %d\n"

      , indexOfChar_String(s,'b')

      , indexOfChar_String(s,'z')

      );
```

```
1) 1
2) -1
```

indexOfCharFrom_String

The function searches the given symbol in the string starting from *fromIndex* position and return a number of the first found position.

```
int indexOfCharFrom_String( String* self, int ch, int fromIndex )
```

Parametrs

self

A string where a symbol should be searched.

ch

A sought-for symbol.

fromIndex

A beginning positin of the search.

Return value

A number of the first symbol *ch* in a string *self*, beginning from *fromIndex* position. If fromIndex position is greater than length of *self*, i.e. *fromIndex* > length_String(*self*), -1 is returned. Is a symbol couldn't be found -1 is returned. For symbols with the code 0 the function always returns -1.

Additional information

If fromIndex < 0, the position is considered to be 0.

```
String* s = create_String("abracadabra");

printf( "1) %d\n2) %d\n"
      , indexOfCharFrom_String(s,'b',5)
      , indexOfCharFrom_String(s,'b',9)
      );
```

```
1) 8
2) -1
```

indexOfString_String

Searches the given sub-string in a string and returns a number of the first found position.

```
int indexOfString_String( String* self, String* str )
```

Parametrs

self

A string where sub-string should be searched.

str

A sought-for sub-string.

Return value

If sub-string *str* is found in *self*, then a position in the string *self*, from which *str* begins, is returned. If sub-string isn't found, -1 is returned.

Additional information

If sub-string is empty, it considers to be a part of any string (including empty one) from 0 position.

```
String* s  = create_String("abracadabra");

String* s1 = create_String("abra");

String* s2 = create_String("cdbr");

printf( "1) %d\n2) %d\n"
      , indexOfString_String(s,s1)
      , indexOfString_String(s,s2)
      );
```

```
1) 0
2) -1
```

indexOfStringFrom_String

Searches the given sub-string in a string beginning from *fromIndex* position and returns a number of the first found position.

```
int indexOfStringFrom_String
    ( String* self, String* str, int fromIndex )
```

Parametrs

self

A given string.

str

A sought-for sub-string.

fromIndex

A beginning positin of the search.

Return value

If sub-string *str* is found in the string *self*, beginning from *fromIndex* position, then a position in the string *self*, from which *str* begins, is returned. If fromIndex position is greater than length of *self*, i.e. *fromIndex* > length_String(*self*), -1 is returned. If a symbol couldn't be found -1 is returned.

Additional information

If *fromIndex* < 0, then *fromIndex* is considered to be 0. If sub-string is empty, it considers to be a part of any string (including empty one) from fromIndex position.

```
String* s  = create_String("abracadabra");

String* s1 = create_String("abra");

printf( "1) %d\n2) %d\n"
      , indexOfString_String(s,s1,5)
      , indexOfString_String(s,s1,8)
      );
```

```
1) 7
```

```
2) -1
```

lastIndexOfChar_String

Searches the given symbol in the string right to left and returns a number of the first found position.

```
int lastIndexOfChar_String( String* self, int ch )
```

Parametrs

self

A given string.

ch

A sought-for symbol.

Return value

The position of the last symbol *ch* in the string *self*. If the symbol isn't found – 1 is returned. For a symbol with the code 0, -1 is always returned.

Additional information

```
String* s = create_String("abracadabra");

printf( "1) %d\n2) %d\n"
      , lastIndexOfChar_String(s,'b')
      , lastIndexOfChar_String(s,'z')
      );
```

```
1) 8
2) -1
```

lastIndexOfCharFrom_String

Searches the given symbol in the string right to left starting from the `given` position and returns a number of the first found position.

```
int lastIndexOfCharFrom_String( String* self, int ch, int fromIndex )
```

Parametrs

self

A given string.

ch

A sought-for symbol.

fromIndex

A beginning positin of the search.

Return value

A position of the symbol *ch* in the string *self*. If *fromIndex* < 0, – 1 is returned. If the symbol

isn't found, the function returns -1. For a symbol with the code 0, -1 is always returned.

Additional information

If `fromIndex` position is greater than length of *`self`*, i.e. *`fromIndex`* `> length_String(`*`self`*`)`, then the last symbol position of the string is the position of the search beginning.

```
String* s = create_String("abracadabra");

printf( "1) %d\n2) %d\n"
      , lastIndexOfCharFrom_String(s,'b',5)
      , lastIndexOfCharFrom_String(s,'b',0)
      );
```

```
1) 1
2) -1
```

lastIndexOfString_String

Searches the given sub-string in the string right to left and returns a number of the first found position.

```
int lastIndexOfString_String( String* self, String* str )
```

Parametrs

`self`

A given string.

`str`

A sought-for sub-string.

Return value

If sub-string isn't found -1 is returned.

Additional information

If sub-string is empty, it considers to be a part of any string (including empty one) from `length_String(`*`self`*`)` position.

```
String* s  = create_String("abracadabra");

String* s1 = create_String("abra");

String* s2 = create_String("cdbr");

printf( "1) %d\n2) %d\n"
      , indexOfString_String(s,s1)
      , indexOfString_String(s,s2)
      );
```

```
1) 7
2) -1
```

lastIndexOfStringFrom_String

Searches the given sub-string in a string right to left starting from the *fromIndex* position and returns a number of the first found position.

```
int lastIndexOfStringFrom_String( String* self,
                                  String* str,
                                  int fromIndex
                                )
```

Parametrs

self

A given string.

str

A sought-for sub-string.

fromIndex

A beginning positin of the search.

Return value

The position of the sub-string *str* in the string *self*. If *fromIndex* < 0, -1 is returned.

Additional information

If fromIndex position is greater than length of *self*, i.e. *fromIndex* > length_String(*self*), then length_String(self)is the position of the search beginning. If sub-string is empty, it considers to be a part of any string (including empty one) from length_String(*self*) position.

```
String* s  = create_String("abracadabra");

String* s1 = create_String("abra");

printf( "1) %d\n2) %d\n"
      , lastIndexOfString_String(s,s1,3)
      , lastIndexOfString_String(s,s1,2)
      );
```

```
1) 0
2) -1
```

length_String

The function returns the length of a string.

```
int length_String( String* self )
```

Parametrs

self

A string the length of which is to be found.

Return value

The exact number of symbols in the string *self*.

Additional information

This function is defined along with the standard function create.

```
String* s = create_String("abracadabra");

printf( "%d\n", length_String(s) );
```

```
11
```

regionMatches_String, regionMatchesCase_String

Checks whether the sub-string of string *self* matches the sub-string of the string *other*. The function regionMatchesCase_String takes letters case into account, the function regionMatches_String has an additional parameter *ignoreCase*, which let to ignore letters case or to take it into account.

```
bool regionMatches_String
     ( String* self, bool ignoreCase, int toffset, String* other, int
ooffset, int len)

bool regionMatchesCase_String
    ( String* self, int toffset, String* other, int ooffset, int len)
```

Parametrs

self

A first string.

toffset

A position of the fragment in the first string.

other

A second string.

ooffset

A position of the fragment in the second string.

len

A length of compared objects.

ignoreCase

If the parameter is true then case is ignored, if false then it takes into account.

Return value

true, if the given segments coincide and false in any other case.

Additional information

The length of the fragments must be positive (*len* ≥ 0). If the fragments defined by the position and the length go out of the position of the string is false returned.

```
SString* s1 = create_String("aBrAcAdabra");

String* s2 = create_String("cadabra");

printf( "a1) %d\na2) %d\n"
      , regionMatchesCase_String(s1,0,s2,3,4)
      , regionMatchesCase_String(s1,7,s2,3,4)
      );

printf( "b1) %d\nb2) %d\n"
      , regionMatches_String(s1,false,0,s2,3,4)
      , regionMatches_String(s1,true,0,s2,3,4)
      );
```

```
a1) 0
a2) 1
b1) 0
b2) 1
```

replace_String

Replaces in the given string all the *oldChar* symbols to the *newChar* symbols.

```
String* replace_String( String* self, char oldChar, char newChar )
```

Parametrs

self

A given string.

oldChar

A symbol which is to be changed.

newChar

A symbol with which entries of *oldChar* are replaced.

Return value

A specification string is gotten from *self* changing symbols *oldChar* to *newChar*.

Additional information

The symbols can't have the code 0.

```
String* s = create_String("abracadabra");

String* res = replace_String(s,'a','_');

printf( "%s\n", toCharArray(res) );
```

```
_br_c_d_br_
```

substringFrom_String

The function returns sub-string with all the symbols from the *beginIndex* position to the end.

```
String* substringFrom_String( String* self, int beginIndex )
```

Parametrs

self

A given string.

beginIndex

A position of the sought-for string beginning

Return value

A specification string which is a sub-string of *self*, extracted using mentioned rules.

Additional information

The position *beginIndex* can't go beyond the string:
0 ≤ *beginIndex* ≤ length_String(*self*).

```
String* s  = create_String("abracadabra");

String* res = substringFrom_String(s,4);

printf( "%s\n", toCharArray_String(res) );
```

```
cadabra
```

substring_String

The function returns sub-string with all the symbols from the *beginIndex* position to the *endIndex inclusive*.

```
String* substring_String
        ( String* self, int beginIndex, int endIndex )
```

Parametrs

self

A given string.

beginIndex

A position of the sought-for string beginning

endIndex

An incremented position of the end of the string.

Return value

A specification string which is a sub-string of *self*, extracted using mentioned rules.

Additional information

The position *beginIndex* can't be negative and can't be greater than *endIndex*, and the position *endIndex* can't be greater than the length of the string:
0 ≤ *beginIndex* ≤ *endIndex* ≤ length_String(*self*).

If the beginning and the ending positions are the same then an empty string is returned.

```
String* s  = create_String("abracadabra");

String* res = substring_String(s,4,7);

printf( "%s\n", toCharArray_String(res) );
```

```
cad
```

toLowerCase_String
The function converts letters to the lower case.

```
String* toLowerCase_String( String* self )
```

Parametrs

self

A string which is to be converted to the lower case.

Return value

A specification string gotten from *self* by converting to the lower case.

Additional information

```
String* s   = create_String("aBrAcAdAbRa");

String* res = toLowerCase_String(s);

printf( "%s\n", toCharArray_String(res) );
```

```
abracadabra
```

toUpperCase_String
The function converts letters to the upper case.

```
String* toUpperCase_String( String* self )
```

Parametrs

self

A string which is to be converted to the upper case.

Return value

A specification string gotten from *self* by converting to the upper case.

Additional information

```
String* s   = create_String("aBrAcAdAbRa");

String* res = toLowerCase_String(s);

printf( "%s\n", toCharArray_String(res) );
```

```
ABRACADABRA
```

toCharArray_String

The function returns a C-string corresponding to the given specification string.

```
const char* toCharArray_String( String* self )
```

Parametrs

self

A given specification reference of the type `String`.

Return value

A symbol array, corresponding to the specification string *self*.

Additional information

```
String* s = create_String("abracadabra");

printf( "%s\n", toCharArray_String(s) );
```

```
abracadabra
```

trim_String

The function returns a string, constructed from a self string by removing space characters from the beginning and the ending of the string.

```
String* trim_String( String* self )
```

Parametrs

self

A string from which first and last whitespaces are to be excluded.

Return value

A specification reference of the type `String` to the string gotten from *self* excluding first and last whitespaces.

Additional information

Whitespaces are whitespaces, tabulations and line feeds.

```
String* s   = create_String(" \tabracadabra  \n");
```

```
String* res = trim_String(s);

printf( "'%s'\n", toCharArray_String(res) );
```

```
'abracadabra'
```

format_String

The function returns a specification string, corresponding to output of printf function, invoked with the same parameters

```
String* format_String(const char *format, ...)
```

Parametrs

format

A string , setting formatting using rules of the `printf function`.

Return value

A specification reference of the type `String`, corresponding to the output of the function `printf` with a parameter *format*.

Additional information

```
char s[12];

String* str;

sprintf(s,"abra%s","cadabra");

str = create_String(s);

String* str = format_String("abra%s","cadabra");
```

The two shown methods are equivalent.

vformat_String

The function returns a specification string, corresponding to output of `vprintf` function, invoked with the same parameters.

```
String* vformat_String(const char *format, va_list args )
```

Parametrs

format

A string , setting formatting using rules of the `function printf`.

args

Arguments which are passed to one of the functions of `printf` family.

Return value

A specification reference of the type `String`, corresponding to the output of the function `printf`

with a parameter *format*.

Additional information

```
char s[12];

String* str;

vsprintf(s,"abra%s","cadabra");

str = create_String(s);

String* str = vformat_String("abra%s","cadabra");
```

The two shown methods are equivalent.

valueOfBool_String

The function returns a string representation of `bool`.

```
String* valueOfBool_String( bool b )
```

Parametrs

b

A value of the type `bool`, for which a string representation is constructing.

Return value

A specification reference of the type `String` to the string representation of value *b*.

Additional information

```
String* s1 = valueOfBool_String(true);

String* s2 = valueOfBool_String(false);

printf( "1) %s\n2) %s\n"
      , toCharArray_String(s1)
      , toCharArray_String(s2)
      );
```

```
1) true
2) false
```

valueOfChar_String

The function returns a string representation of `char`.

```
String* valueOfChar_String( char c )
```

Parametrs

c

A value of the type `char`, for which a string representation is constructing.

Return value

A specification reference of the type `String` to the string representation of value *c*.

Additional information

```
String* s = valueOfChar_String('a');

printf( "%s\n", toCharArray_String(s) );
```

```
a
```

valueOfShort_String

The function returns a string representation of `short`.

```
String* valueOfShort_String( short i )
```

Parametrs

i

A value of the type `short`, for which a string representation is constructing.

Return value

A specification reference of the type `String` to the string representation of value *i*.

Additional information

```
String* s = valueOfShort_String (-28);

printf( "%s\n", toCharArray_String(s) );
```

```
-28
```

valueOfUShort_String

The function returns a string representation of `unsigned short`.

```
String* valueOfUShort_String( unsigned short i )
```

Parametrs

i

A value of the type `unsigned short`, for which a string representation is constructing.

Return value

A specification reference of the type `String` to the string representation of value *i*.

Additional information

```
String* s = valueOfUShort_String(47);

printf( "%s\n", toCharArray_String(s) );
```

```
47
```

valueOfInt_String

The function returns a string representation of `int`.

```
String* valueOfInt_String( int i )
```

Parametrs

i

A value of the type `int`, for which a string representation is constructing.

Return value

A specification reference of the type `String` to the string representation of value *i*.

Additional information

```
String* s = valueOfInt_String(-28);

printf( "%s\n", toCharArray_String(s) );
```

```
-28
```

valueOfUInt_String

The function returns a string representation of `unsigned int`.

```
String* valueOfUInt_String( unsigned int i )
```

Parametrs

i

A value of the type `unsigned int`, for which a string representation is constructing.

Return value

A specification reference of the type `String` to the string representation of value *i*.

Additional information

```
String* s = valueOfUInt_String(47);

printf( "%s\n", toCharArray_String(s) );
```

```
47
```

valueOfLong_String

The function returns a string representation of `long`.

```
String* valueOfLong_String( long i )
```

Parametrs

i

A value of the type `long`, for which a string representation is constructing.

Return value

A specification reference of the type `String` to the string representation of value *i*.

Additional information

```
String* s = valueOfLong_String(-28);

printf( "%s\n", toCharArray_String(s) );
```

```
-28
```

valueOfULong_String

The function returns a string representation of `unsigned long`.

```
String* valueOfULong_String( unsigned long i )
```

Parametrs

i

A value of the type `unsigned long`, for which a string representation is constructing.

Return value

A specification reference of the type `String` to the string representation of value *i*.

Additional information

```
String* s = valueOfULong_String(47);

printf( "%s\n", toCharArray_String(s) );
```

```
47
```

valueOfFloat_String

The function returns a string representation of `float`.

```
String* valueOfFloat_String( float f )
```

Parametrs

f

A value of the type `float`, for which a string representation is constructing.

Return value

A specification reference of the type `String` to the string representation of value *f*.

Additional information

```
String* s = valueOfFloat_String(3.14);

printf( "%s\n", toCharArray_String(s) );
```

```
3.140000
```

valueOfDouble_String

The function returns a string representation of `double`.

```
String* valueOfDouble_String( double d )
```

Parametrs

d

A value of the type `double`, for which a string representation is constructing.

Return value

A specification reference of the type `String` to the string representation of value *d*.

Additional information

```
String* s = valueOfDouble_String(3.14);

printf( "%s\n", toCharArray_String(s) );
```

```
3.140000
```

valueOfPtr_String

The function returns a string representation of `void*`.

```
String* valueOfPtr_String( void *p )
```

Parametrs

p

A value of the type `void*`, for which a string representation is constructing.

Return value

A specification reference of the type `String` to the string representation of not typified pointer *p*.

Additional information

```
int i;

String* s = valueOfPtr_String((void*)&i);

printf( "%s\n", toCharArray_String(s) );
```

```
0012FF6C
```

valueOfObject_String

The function returns a string representation of a specification type.

```
String* valueOfObject_String( Object* ref )
```

Parametrs

ref

A value of specification type, for which a string representation is constructing.

Return value

A specification reference of the type `String` to the string representation of a specification type value *p*.

Additional information

The result is the same as the returned value of standard function `toString`:

```
Object* ref;

String* s = valueOfObject_String(ref);

String* s = toString(ref);
```

valueOfBytes_String

The function returns a string hexadecimal representation of `p` byte array of a given length.

```
String* valueOfBytes_String( const char* p, int l )
```

Parametrs

p

A byte array, for which a string representation is constructing.

l

A length of byte array *p*.

Return value

A specification reference of the type `String` to the string representation of byte array *p*.

Additional information

```
char a[6] = { 0x00, 0x33, 0x66, 0x99, 0xCC, 0xFF };

String* s = valueOfBytes_String(a,6);

printf( "%s\n", toCharArray_String(s) );
```

```
[00 33 66 99 CC FF]
```

14.2.12 List

Type `List` is a container data type, implementing an ordered list of items.

Any specification references can be elements of the list. Type of elements can be constrained when a list is created. Such a list is called *typified*, and all functions, related to the typified list, will check that `Object*` parameter actual type is equal to data type of list's elements.

Two lists are equal if they have the same length and their elements are equal in pairs. Typification of lists is not considered at that. Particularly, empty lists are always equal.

Elements of the list are numbered from 0.

Header file: atl/list.h

`add_List`
The function inserts an element into the given position.

`addAll_List`
The function adds all the elements of one list to another one inserting them in the same order from the given position.

`append_List`
The function appends the given element at the end of the list.

`appendAll_List`
The function adds all the elements of one list to the end of another one.

`clear_List`
The function removes all the elements from the list.

`contains_List`
The function checks whether the list contains an element, equals to the given one.

`create_List`
The function creates a list and returns a specification reference of `List` data type.

`elemType_List`
The function returns a pointer to a constant descriptor of specification data type, that constrains data type of the list's elements.

`get_List`
The function returns a specification reference to the element at the given position.

`indexOf_List`
The function returns a number of the position of the first element with the given value.

`isEmpty_List`
The function checks if the list is empty.

`lastIndexOf_List`
The function returns a number of the position of the last element in the list which has the given value.

`remove_List`
The function removes the element at the given position from the list.

`set_List`
The function replaces an element at the given position by the given object.

`size_List`
The function returns the length of the list.

`subList_List`

The function returns elements of the given list, containing between the given positions, as a new list.

toSet_List
The function returns set which consists of the unique elements of the given list.

add_List

The function inserts an element into the given position.

```
void add_List( List* self, int index, Object* ref )
```

Parametrs

self

A list, where an insertion is proceeded.

index

A position in the list where value *ref* is inserted.

ref

A value of the specification type which is inserted in the list.

Additional information

If the list is typified, data type of ref element must coincide with data type of list's elements. Number of the position must be in the range from 0 to the length of the list including, i.e. `0 ≤ index ≤ size_List(self)`. If the number of the position is equal to the length of the list, the element is appended to the end of this list

```
List* l = create_List(&type_Integer);

add_List(l,0,create_Integer(28));

add_List(l,0,create_Integer(47));

add_List(l,1,create_Integer(63));
```

List *l* is changing during the run-time this way:

1. `< 28 >`
2. `< 47, 28 >`
3. `< 47, 63, 28 >`

addAll_List

The function adds all the elements of one list to another one inserting them in the same order from the given position.

```
void addAll_List(List* self, int index, List* other)
```

Parametrs

self

A list, where new elements are inserted.

index

A position from which begins the insertion of new elements.

other

A list the elements of which are inserted in the list *self*.

Additional information

If self list is typified, types of all elements of other list must coincide with type of elements of self list (other list itself must not be typified). Number of position must be in the range from 0 to the length of self list including, i.e. 0 ≤ index ≤ size_List(self). If number of the position is equal to the length of self list, elements are appended to the end of this list.

```
List* l1 = create_List(&type_Integer);

List* l2 = create_List(NULL);

append_List(l1,create_Integer(28));

append_List(l1,create_Integer(47));

append_List(l2,create_Integer(63));

append_List(l2,create_Integer(85));

addAll_List(l1,1,l2);
```

List *l1* before the call of `addAll_List` consists of the following elements: `< 28, 47 >`. List *l2*: `< 63, 85 >`.After the call of `addAll_List` the list *l1* also consists of the elements of *l2*: `< 28, 63, 85, 47 >`.

append_List

The function appends the given element at the end of the list.

```
void append_List( List* self, Object* ref )
```

Parametrs

self

A list, where an insertion is proceeded.

ref

A value of the specification type which is inserted in the list.

Return value

If the list is not typified (during the creation no restrictions was placed on the elements' types), the function returns `NULL`.

Additional information

```
List* l = create_List(&type_Integer);

append_List(l,create_Integer(28));
```

```
append_List(l,create_Integer(47));

append_List(l,create_Integer(63));
```

The list *l* is changing during the run-time this way:

1. < 28 >
2. < 28, 47 >
3. < 28, 47, 63 >

appendAll_List

The function adds all the elements of one list to the end of another one.

```
void appendAll_List(List* self, List* other)
```

Parametrs

self

A list, where new elements are inserted.

other

A list the elements of which are inserted into the list *self*.

Additional information

If list `self` is typified, types of all elements of the list `other` must coincide with the type of the elements of the list `self` (list `other` itself is not required to be typified).

```
List* l1 = create_List(&type_Integer);

List* l2 = create_List(NULL);

append_List(l1,create_Integer(28));

append_List(l1,create_Integer(47));

append_List(l2,create_Integer(63));

append_List(l2,create_Integer(85));

appendAll_List(l1,l2);
```

List *l1* before the call of `addAll_List` consists of the following elements: `< 28, 47 >`. List *l2*: `< 63, 85 >`. After the call of appendAll_List list *l1* also consists of the elements of *l2* `< 28, 47, 63, 85 >`.

clear_List

The function removes all the elements from the list.

```
void clear_List( List* self )
```

Parametrs

self

A list which should be cleared.

Additional information

```
List* l = create_List(&type_Integer);

append_List(l,create_Integer(28));

append_List(l,create_Integer(47));

clear_List(l);
```

List l is changing during the run-time this way:

1. < 28 >
2. < 28, 47 >
3. <>

The same result can be gotten by recreating the list:

```
List* l = create_List(&type_Integer);

...

l = create_List(elemType_List(l));
```

But this method is less effective than the call of function `clear_List`.

contains_List

The function checks whether the list contains an element, equals to the given one.

```
bool contains_List( List* self, Object* ref )
```

Parametrs

self

An under test list.

ref

A specification type value which is searched in the list.

Return value

`true`, if *ref* is in the *self*. `false` in any other case.

Additional information

If the list is typified then the type of element *ref* must be the same as the types of the elements of the list.

```
bool contains;
```

```
List* l = create_List(&type_Integer);

append_List(l,create_Integer(28));

append_List(l,create_Integer(47));

contains = contains_List(l,create_Integer(28));
```

List l is changing during the run-time this way:

1. `< 28 >`
2. `< 47, 28 >`

the value of variable contains after the code execution will be `true`.

create_List

The function creates a list and returns a specification reference of `List` data type.

```
List* create_List( const Type *elem_type )
```

Parametrs

elem_type

A pointer to constant descriptor of the list's elements type.

Return value

A specification reference to the created list of the type `List`.

Additional information

If parameter elem_type is NULL, then types of the elements are not restricted .

```
List* l1 = create_List(NULL);

List* l2 = create_List(&type_Integer);
```

By the first call of `create_List` a list is created, in which any elements can be added. A list created by the second call can store only `Integer` elements.

This function `create_List` is defines along with the standard function `create`.

```
List* l1 = create(&type_List, NULL);

List* l2 = create_List(NULL);
```

The two shown methods of creating a specification reference are functionally equivalent, but the use of create is not type-safe and cause a compiler warning (see function create definition).

elemType_List

The function returns a pointer to a constant descriptor of specification data type, that constrains data type of the list's elements.

```
Type *elemType_List(List* self)
```

Parametrs

self

A list limitation of which should be returned by the function.

Return value

A pointer to the constant descriptor of a specification type by which values of the list are limited.

If the list is not typified (during the creation no restrictions was placed on the elements' types), the function returns NULL.

Additional information

```
List* l = create_List(&type_Integer);

Type *t = elemType_List(l);
```

A variable *t* gets value `&type_Integer`.

get_List

The function returns a specification reference to the element at the given position.

```
Object* get_List( List* self, int index )
```

Parametrs

self

A list with a needed element.

index

A position of the element which must be returned by the function.

Return value

A position where in the list is value *ref*. If there is no such value in the list, the return value is `-1`.

Additional information

A number of the position must be in the rage from 0 to the length of the list decremented one, i.e. $0 \leq index < size\_List(self)$.

```
List* l = create_List(&type_Integer);

Object* o;

append_List(l,create_Integer(28));

append_List(l,create_Integer(47));

append_List(l,create_Integer(63));

o = get_List(l,1);
```

List l is changing during the run-time this way:

1. `< 28 >`
2. `< 28, 47 >`
3. `< 28, 47, 63 >`

The value of object *o* after the call of `get_List` will be `47`.

indexOf_List

The function returns a number of the position of the first element with the given value.

```
int indexOf_List( List* self, Object* ref )
```

Parametrs

self

A list with a needed element.

ref

A specification reference with the value of which elements of the list are compared.

Return value

A position where in the list is value *ref*. If there is no such value in the list, the return value is `-1`.

Additional information

If the list is typified when the type of the element *ref* must be the same as the type of the elements of the list. If there is no such value in the list, the return value is `-1`.

```
List* l = create_List(&type_Integer);

append_List(l,create_Integer(28));

append_List(l,create_Integer(47));

append_List(l,create_Integer(28));

int pos = indexOf_List(l,create_Integer(28));
```

List l is changing during the run-time this way:

1. `< 28 >`
2. `< 28, 47 >`
3. `< 28, 47, 28 >`

A value of the variable *pos* after the call of `get_List` will be `0`.

isEmpty_List

The function checks if the list is empty.

```
bool isEmpty_List( List* self )
```

Parametrs

self

A list with a needed element.

Return value

`true`, if the list has no elements. `false` in any other case.

Additional information

If the list is typified when the type of the element *ref* must be the same as the type of the elements of the list. If there is no such element in the list, the return value is `-1`.

```
bool empty;

List* l = create_List(&type_Integer);

empty = isEmpty_List(l);

append_List(l,create_Integer(28));

empty = isEmpty_List(l);
```

The value of variable *empty* after the first call of `isEmpty_List` will be `true`, and after the second one (when 28 will be added) — `false`.

lastIndexOf_List

The function returns a number of the position of the last element in the list which has the given value.

```
int lastIndexOf_List( List* self, Object* ref )
```

Parametrs

self

A list with a needed element.

ref

A specification reference with the value of which elements of the list are compared.

Return value

A position where in the list is value *ref*. If there is no such value in the list, the return value is `-1`.

Additional information

If the list is typified when the type of the element *ref* must be the same as the type of the elements of the list. If there is no such value in the list, the return value is `-1`.

```
List* l = create_List(&type_Integer);

append_List(l,create_Integer(28));
```

```
append_List(l,create_Integer(47));

append_List(l,create_Integer(28));

int pos = lastIndexOf_List(l,create_Integer(28));
```

List l is changing during the run-time this way:

1. < 28 >
2. < 28, 47 >
3. < 28, 47, 28 >

The value of the variable *pos* after the call of `get_List` will be `2`.

remove_List

The function removes the element at the given position from the list.

```
void remove_List( List* self, int index )
```

Parametrs

self

A list, from which elements are deleted.

index

A position in the list from which an element is deleted.

Additional information

The number of the position must be in the interval from 0 to the length of the list including, i.e. $0 \leq index \leq size\_List(self)$.

```
List* l = create_List(&type_Integer);

append_List(l,create_Integer(28));

append_List(l,create_Integer(47));

append_List(l,create_Integer(63));

remove_List(l,1);
```

List l is changing during the run-time this way:

1. < 28 >
2. < 28, 47 >
3. < 28, 47, 63 >
4. < 28, 63 >

set_List

The function replaces an element at the given position by the given object.

```
void set_List( List* self, int index, Object* ref )
```

Parametrs

self

A list where the replacement is done.

index

A position in the list in which value *ref* is placed.

ref

A value of a specification type inserted in the list.

Additional information

If the list is typified when the type of the element ref must be the same as the type of the elements of the list. The number of the position must be in the interval from 0 to the length of the list including, i.e. `0 ≤ index ≤ size_List(self)`.If the number of the position is equal to the length of the list when the element is inserted at the end of the list.

```
List* l = create_List(&type_Integer);

append_List(l,create_Integer(28));

append_List(l,create_Integer(47));

set_List(l,1,create_Integer(63));
```

List l is changing during the run-time this way:

1. `< 28 >`
2. `< 28, 47 >`
3. `< 28, 63 >`

size_List

The function returns the length of the list.

```
int size_List( List* self )
```

Parametrs

self

A list the length of which the function must return.

Return value

A particular number of the elements of list *self*.

Additional information

```
int size;

List* l = create_List(&type_Integer);
```

```
size = size_List(l);

append_List(l,create_Integer(28));

size = size_List(l);
```

After the first call of `size_List` variable *`size`* becomes 0. Ater the second call of `size_List` the variable is `1`: one value has been added to the list.

subList_List

The function returns elements of the given list, containing between the given positions, as a new list.

```
List* subList_List( List* self, int fromIndex, int toIndex )
```

Parametrs

`self`

An initial list.

`fromIndex`

A position of the first element of the new list.

`toIndex`

A number of the last element's position of the new list incremented by one.

Return value

A specification reference of the type `List`, a sub-list of *`self`*.

Additional information

A position of the first element of the new list — *`fromIndex`*; a position of the last element — *`toIndex`*`-1`. Position *`fromIndex`* can't be negative and can't be greater than *`toIndex`*, and position *`toIndex`* can't be greater than the length of the list: `0 ≤` *`fromIndex`* `≤` *`toIndex`* `≤ size_List(`*`self`*`)`. If the first and the last positions are equal an empty list is returned.

```
List* l = create_List(&type_Integer);

List* l2;

append_List(l,create_Integer(28));

append_List(l,create_Integer(47));

append_List(l,create_Integer(63));

l2 = subList_List(l,1,3);
```

List *`l`* changes and in the result has the following elements: `< 28, 47, 63 >`. List *`l2`* after the call of function `subList_List` has two elements: `< 47, 63 >`.

toSet_List

The function returns a set which consists of the unique elements of the given list.

```
Set* toSet_List(List* self)
```

Parametrs

self

A list, a set of elements of which is to be gotten.

Return value

A specification reference of the type `Set`, set of elements of list *self*.

Additional information

A returned set has the same typification as the list: if elements of the list was constrained by a data type, elements of the set will be constrained by the same type.

```
List* l = create_List(&type_Integer);

Set* s;

Type *t;

append_List(l,create_Integer(28));

append_List(l,create_Integer(47));

append_List(l,create_Integer(28));

s = toSet_List(l);

t = elemType_Set(s);
```

List l is changing during the run-time this way:

1. `< 28 >`
2. `< 28, 47 >`
3. `< 28, 47, 28 >`

Set *s*, returned by function `toSet_List`, has elemnts: `< 28, 47 >`. A call of `elemType_Set(s)` returns `&type_Integer`.

14.2.13 Set

Type `Set` is a container type, implementing a set of elements.

Any specification references can be elements of a set. Type of elements can be constrained when creating a set. Such a set is called typified, and all functions, related to the typified set, will check that `Object*` parameter actual type is equal to the data type of set's elements.

Two sets are equal if they have the same elements. Typification of sets is not considered at that. Particularly, empty sets are always equal.

Header file: atl/set.h

`add_Set`
The function inserts the given element into the set.

`addAll_Set`
The function unions two sets: adds the elements of one set to another one.

`clear_Set`
The function removes all the elements from the set.

`contains_Set`
The function checks whether the set contains an element, equals to the given one.

`containsAll_Set`
The function checks whether one set is a subset of another one, in other words, checks whether first set contains all the elements of the second one.

`create_Set`
The function creates a set and returns a specification reference of Set data type.

`elemType_Set`
The function returns a pointer to a constant descriptor of specification data type, that constrains the set's elements.

`get_Set`
The function returns a specification reference to the element with `the given` number; elements are numerated in a random order .

`isEmpty_Set`
The function checks whether the set is empty.

`remove_Set`
The function removes an element from the set.

`removeAll_Set`
Subtracts one set from another one: removes from the first set elements, that belong to the second one.

`retainAll_Set`
Gets an intersection of the sets: retains in the first set only such elements, which also belong to the second one.

`size_Set`
The function returns the number of the elements in the set.

`toList_Set`
The function returns a `List` containing all the elements of the given set .

add_Set

The function inserts the given element into the set.

```
bool add_Set( Set* self, Object* ref )
```

Parametrs

self

A set where elements are inserted.

ref

A value of a specification type which is inserted in the set.

Return value

`true`, if the insertion was successful.In any other case — `false`.

Additional information

If the set is typified, then the type of element *ref* should be the same as the type of the set's elements.

```
Set* s = create_Set(&type_Integer);

add_Set(s, create_Integer(28));

add_Set(s, create_Integer(47));

add_Set(s, create_Integer(28));
```

Set s is changing during the run-time this way:

1. `{ 28 }`
2. `{ 28, 47 }`
3. `{ 28, 47 }`

addAll_Set

The function unions two sets: adds the elements of one set to another one.

```
bool addAll_Set( Set* self, Set* set )
```

Parametrs

self

A set, in which elements are added.

set

A set, elements of which are added to *self*.

Return value

If during the unification even one element was added to *self* , the function returns `true`.In any other case — `false`.

Additional information

If set *self* is typified, then the types of all the elements of set *set* must be the same as the type of set's *self* elements (*set* itself is not required to be typified).

```
bool changed;

Set* s1 = create_Set(&type_Integer);

Set* s2 = create_Set(NULL);

add_Set(s1,create_Integer(28));

add_Set(s1,create_Integer(47));

add_Set(s2,create_Integer(28));

add_Set(s2,create_Integer(47));

add_Set(s2,create_Integer(63));

changed = addAll_Set(s1,s2);
```

Set *s1* changes this way:

1. `{ 28 }`
2. `{ 28, 47 }`
3. `{ 28, 47, 63 }`

Set *s2* changes this way:

1. `{ 28 }`
2. `{ 28, 47 }`

after the call of `addAll_Set` to set *s1* `is added` element `63`, and variable *changed* go to `true`.

clear_Set

The function removes all the elements from the set.

```
void clear_Set( Set* self )
```

Parametrs

self

A cleared set.

Additional information

```
Set* s = create_Set(&type_Integer);

add_Set(s,create_Integer(28));

add_Set(s,create_Integer(47));

clear_Set(s);
```

Set s is changing during the run-time this way:

1. `{ 28 }`
2. `{ 28, 47 }`
3. `{}`

The same result can be gotten by having recreating the set:

```
Set* s = create_Set(&type_Integer);

...

s = create_Set(elemType_Set(s));
```

But this result is less effective than the call of `clear_Set`.

contains_Set

The function checks whether the set contains an element, equals to the given one.

```
bool contains_Set( Set* self, Object* ref )
```

Parametrs

self

An under test set.

ref

A value of a specification type a search of which occurs in the set.

Return value

`true`, if element *ref* belongs to set *self*. `false` otherwise.

Additional information

If the set is typified, then the type of element *ref* should be the same as the type of the set's elements.

```
bool contains;

Set* s = create_Set(&type_Integer);

add_Set(s,create_Integer(28));

add_Set(s,create_Integer(47));

contains = contains_Set(s,create_Integer(28));
```

Set s is changing during the run-time this way:

1. `{ 28 }`
2. `{ 47, 28 }`

An element *contains* value after the call of `contains_Set` will be `true`.

containsAll_Set

The function checks whether one set is a subset of another one, in other words, checks whether the first set contains all the elements of the second one.

```
bool containsAll_Set( Set* self, Set* set )
```

Parametrs

self

A set, which is to be checked whether it contains elements of set *set*.

set

A priori subset of *self*.

Return value

`true`, if all of the elements of set *set* belong to the set *self*. `false` in any other case.

Additional information

```
bool contains;

Set* s1 = create_Set(&type_Integer);

Set* s2 = create_Set(NULL);

add_Set(s1,create_Integer(28));

add_Set(s1,create_Integer(47));

add_Set(s2,create_Integer(28));

add_Set(s2,create_String("a"));

add_Set(s2,create_Integer(47));

contains = containsAll_Set(s1,s2));

contains = containsAll_Set(s2,s1));
```

Set *s1* changes this way:

1. `{ 28 }`
2. `{ 28, 47 }`

Set *s2* changes this way:

1. `{ 28 }`
2. `{ 28, "a" }`
3. `{ 28, "a", 47 }`

After the first call of `containsAll_Set` variable *contains* go to `false`, and after the second call — `true`.

create_Set

The function creates a set and returns a specification reference of Set data type.

```
Set* create_Set( const Type *elem_type )
```

Parametrs

elem_type

A pointer to constant descriptor of the set's elements' type.

Return value

A pointer to constant descriptor of specification data type by which values of this set are limited. If the set is not typified (during the creation there was no limitations of the elements' type), the function returns `NULL`.

Additional information

If parameter *elem_type* is NULL, than types of set's elements are not limited.

```
Set* s1 = create_Set(NULL);

Set* s2 = create_Set(&type_Integer);
```

After the first call of `create_Set` was created a set, in which any elements can be added. A set created by the second call can store elements of `Integer` type only.

The function `create_Set` is defined along with the standard function `create`.

```
Set* s1 = create(&type_Set, NULL);

Set* s2 = create_Set(NULL);
```

The two shown methods of creating a specification reference are functionally equivalent, but the use of create is not type-safe and cause a compiler warning (see function create definition).

elemType_Set

The function returns a pointer to a constant descriptor of specification data type, that constrains the set's elements.

```
Type *elemType_Set(Set* self)
```

Parametrs

self

A set , limitations of which the function must return.

Return value

A pointer to constant descriptor of specification data type by which values of this set are limited. If the set is not typified (during the creation there was no limitations of the elements' type), the function returns `NULL`.

Additional information

```
Set* s = create_Set(&type_Integer);
```

```
Type *t = elemType_Set(s);
```

Variable *t* gets value `&type_Integer`.

get_Set

The function returns a specification reference to the element with `the given` number; elements are numerated in a random order .

```
Object* get_Set( Set* self, int index )
```

Parametrs

self

A set, an element of which the function should return.

index

An element number, which should be returned by the function.

Return value

A specification reference of the type `Object`.

Additional information

This function is intended for enumeration of set's elements. Since a set is not ordered, elements are numerated in a random order. A number of position must be in the range from 0 to the size of the set – 1, i.e. `0 ≤ index < size_Set(self)`.

```
Set* s1 = create_Set(&type_Integer);

Set* s2 = create_Set(&type_Integer);

int i;

bool equ;

add_Set(s1,create_Integer(28));

add_Set(s1,create_Integer(47));

add_Set(s1,create_Integer(63));

for(i=0; i < size_Set(s1); i++)

  add_Set(s2, get_Set(s1,i));

equ = equals(s1,s2);
```

Set *s1* changes during the run-time this way:

1. `{ 28 }`
2. `{ 28, 47 }`
3. `{ 28, 47, 63 }`

An order of adding the elements to set *s2* is not defined, however after an iteration all the elements

of *s1* will be added to *s2* and the value of variable *equ* after the call of `equals` will be `true`.

isEmpty_Set

The function checks whether the set is empty.

```
bool isEmpty_Set( Set* self )
```

Parametrs

self

An under-test set.

Return value

`true`, if there is no elements in the set and `false` otherwise.

Additional information

```
bool empty;

Set* s = create_Set(&type_Integer);

empty = isEmpty_Set(s);

add_Set(s,create_Integer(28));

empty = isEmpty_Set(s);
```

A value of variable *empty* after the first call of `isEmpty_Set` will be `true`, and after the second one (when number 28 will be added) — `false`.

remove_Set

The function removes an element from the set.

```
void remove_Set( Set* self, Object* ref )
```

Parametrs

self

A set from which elements should be deleted .

ref

An element which is to be deleted from the set.

Additional information

If the set is typified, then the type of element *ref* shoud be the same as the type of the set's elements

```
Set* s = create_Set(&type_Integer);

add_Set(s,create_Integer(28));

add_Set(s,create_Integer(47));
```

```
remove_Set(s,create_Integer(28));
```

Set s is changing during the run-time this way:

1. `{ 28 }`
2. `{ 28, 47 }`
3. `{ 47 }`

removeAll_Set

Subtracts one set from another one: removes from the first set elements, that belong to the second one.

```
bool removeAll_Set( Set* self, Set* set )
```

Parametrs

self

A set from which elements should be deleted.

set

A set elements of which are deleted from *self*.

Return value

If during the subtraction from set *self* at list one element was deleted, the function returns `true`.In any other case — `false`.

Additional information

```
bool changed;

Set* s1 = create_Set(&type_Integer);

Set* s2 = create_Set(NULL);

add_Set(s1,create_Integer(28));

add_Set(s1,create_Integer(47));

add_Set(s1,create_Integer(63));

add_Set(s2,create_Integer(28));

add_Set(s2,create_Integer(47));

changed = removeAll_Set(s1,s2);
```

Set *s1* changes this way:

1. `{ 28 }`
2. `{ 28, 47 }`
3. `{ 28, 47, 63 }`

Set *s2* changes this way:

1. `{ 28 }`
2. `{ 28, 47 }`

After the call of `removeAll_Set` n set *`s1`* element 63 remains, and variable *`changed`* go to `true`.

retainAll_Set

Gets an intersection of the sets: retains in the first set only such elements, which also belong to the second one.

```
bool retainAll_Set( Set* self, Set* set )
```

Parametrs

`self`

A first argument of the sets' intersection. This set will store the result of the intersection.

`set`

A second argument of the sets' intersection.

Return value

`true`, if the number of elements in the intersection is less than the number of elements in *`self`*.In any other case — `false`.

Additional information

```
bool changed;

Set* s1 = create_Set(&type_Integer);

Set* s2 = create_Set(NULL);

add_Set(s1,create_Integer(28));

add_Set(s1,create_Integer(47));

add_Set(s1,create_Integer(63));

add_Set(s2,create_Integer(28));

add_Set(s2,create_Integer(47));

changed = retainAll_Set(s1,s2);
```

Set *`s1`* changes this way:

1. `{ 28 }`
2. `{ 28, 47 }`
3. `{ 28, 47, 63 }`

Set *`s2`* changes this way:

1. `{ 28 }`
2. `{ 28, 47 }`

After the call of `retainAll_Set` in set *`s1`* `elements` *`28`* `and` *`47`* remain,and variable *`changed`* go to `true`.

size_Set

The function returns the number of the elements in the set.

```
int size_Set( Set* self )
```

Parametrs

`self`

A set, power of which the function should return.

Return value

An exact number of elements in set *`self`*.

Additional information

```
int size;

Set* s = create_Set(&type_Integer);

size = size_Set(s);

add_Set(s,create_Integer(28));

size = size_Set(s);
```

After the first call of `size_Set` variable *`size`* go to zero. After the second call of `size_Set` the variable is `1`: one element was added to the set.

toList_Set

The function returns a `List` containing all the elements of the given set.

```
List* toList_Set( Set* self )
```

Parametrs

`self`

A given set.

Return value

A specification reference of the type `List`.

Additional information

Order of the elements in the list is not defined. The returned list has the same typification as the set: if the elements of the set was constrained by a data type, the elements of the list will be constrained by the same type.

```
Set* s = create_Set(&type_Integer);
```

```
List* l;

Type *t;

add_Set(s,create_Integer(28));

add_Set(s,create_Integer(47));

l = toList_Set(s);

t = elemType_Set(l);
```

Set s is changing during the run-time this way:

1. `{ 28 }`
2. `{ 28, 47 }`

List *l*, returned by function `toList_Set`, has elements: `{ 28, 47 }`; They can be stored in any order. The call of `elemType_Set(l)` returns `&type_Integer`.

14.2.14 Map

Type Map is a container type, implementing a mapping of elements from range of definition to range of values. An element from the range of definition is called a *key*, and a corresponding element from the range of values is called a *value*. One key has exactly one corresponding value.

Any specification references can be elements of a map. Type of elements can be constrained when creating a map, separately for elements of range of definition and for elements of range of values. Such a map is called *typified*, and all functions, related to the typified map, will check that an actual parameter type is the same as the type of map's elements of the appropriate range.

Two maps are equal if they have the same range of keys, and values corresponding to the same keys from these maps are equal. Particularly, empty maps are always equal.

Header file: atl/map.h

clear_Map
The function removes all the elements from the map.

containsKey_Map
The function checks whether the map contains a key equals to the given one.

containsValue_Map
The function checks whether the map contains a value equals to the given one.

create_Map
The function creates a map and returns a specification reference of `Map` data type.

get_Map
The function returns a specification reference to the value, corresponding to the given key.

getKey_Map
The function returns a specification reference to some key,to which corresponds the given value.

isEmpty_Map
The function checks whether the map is empty.

key_Map
The function returns a key of the map at the given position.

keyType_Map
The function returns a pointer to a constant descriptor of a specification data type that constrains keys of this map.

put_Map
The function adds a “key–value” pair to the map.

putAll_Map
Adds to the first map all the pairs “key-value” from the second one.

remove_Map
The function removes from the map a pair “key-value” for the given key.

size_Map
The function returns the size of the map.

valueType_Map
The function returns a pointer to a constant descriptor of a specification data type, that constrains values of the map.

clear_Map

The function removes all the elements from the map.

```
void clear_Map( Set* self )
```

Parametrs

self

A cleared map.

```
Map* m = create_Map(&type_Integer, &type_String);

put_Map(m,create_Integer(28),create_String("a"));

put_Map(m,create_Integer(47),create_String("b"));

clear_Map(m);
```

Map *m* changes during the run-time this way:

1. `{ 28 → "a" }`
2. `{ 28 → "a", 47 → "b" }`
3. `{}`

The same result can be gotten having recreating the set:

```
Map* m;

...

m = create_Map(keyType_Map(m), valueType_Map(m));
```

But this method is less effective than the call of function `clear_Map`.

containsKey_Map

The function checks whether the map contains a key equals to the given one.

```
bool containsKey_Map( Map* self, Object* key )
```

Parametrs

self

An under-test map.

key

A key, which is searched in the map.

Return value

`true`, if key *key* is in *self*. `false` otherwise.

Additional information

If a definitional domain is typified then the type of element key should be the same as the type of

map's keys .

```
bool contains;

Map* m = create_Map(&type_Integer, &type_String);

put_Map(m,create_Integer(28),create_String("a"));

put_Map(m,create_Integer(47),create_String("b"));

contains = containsKey_Map(m,create_Integer(28);
```

Map m changes during the run-time this way:

1. { 28 → "a" }
2. { 28 → "a", 47 → "b" }

A value of variable *contains* after the call of containsKey_Map will be true.

containsValue_Map

The function checks whether the map contains a key equals to the given one.

```
bool containsValue_Map( Map* self, Object* value )
```

Parametrs

self

An under-test map.

value

A value, which is searched in the map.

Return value

true,if value *value* containes in *self*. false otherwise.

Additional information

If domain is typified than the type of element *value* must be the same as the type of map's values.

```
bool contains;

Map* m = create_Map(&type_Integer, &type_String);

put_Map(m,create_Integer(28),create_String("a"));

put_Map(m,create_Integer(47),create_String("b"));

contains = containsValue_Map(m,create_String("b"));
```

Map m changes during the run-time this way:

1. { 28 → "a" }
2. { 28 → "a", 47 → "b" }

A value of variable *contains* after the call of containsValue_Map will be true.

create_Map

The function creates a map and returns a specification reference of `Map` data type.

```
Map* create_Map( const Type *key_type, const Type *val_type )
```

Parametrs

`key_type`

A pointer to constant descriptor of the map's key's type.

`val_type`

A pointer to constant descriptor of map's value type.

Return value

A specification reference of the type `Map`.

Additional information

If parameter *`key_type`* is NULL, than types of elements of range of definition are not limited. Otherwise the parameter must be a pointer to a constant descriptor of the map's keys' data type. In a similar manner, if parameter *`val_type`* is NULL, than types of domain's elements are not limited. Otherwise the parameter must be the pointer to a constant descriptor of the map's values' type.

The first line of the example creates a map without typification; the second line creates a map with keys of type `Integer`; the third line creates a map from `Integer` to `String`.

```
Map* m1 = create_Map(NULL, NULL);

Map* m2 = create_Map(&type_Integer, NULL);

Map* m3 = create_Map(&type_Integer, &type_String);
```

The function `create_Map` defined along with standard function `create`.

```
Map* m1 = create(&type_Set, NULL, NULL);

Map* m2 = create_Map(NULL, NULL);
```

The two shown methods of creating a specification reference are functionally equivalent, but the use of create is not type-safe and cause a compiler warning (see function create definition).

get_Map

The function returns a specification reference to the value, corresponding to the given key.

```
Object* get_Map( Map* self, Object* key )
```

Parametrs

`self`

A map, an element of which should be returned by the function.

`key`

A key, corresponded to the needed value.

Return value

A value corresponded to the given key is returned. If a map has no such key `NULL` is returned.

Additional information

If a definitional domain is typified then the type of element key should be the same as the type of map's keys.

```
Map* m = create_Map(&type_Integer, &type_String);

Object* val;

put_Map(m,create_Integer(28),create_String("a"));

put_Map(m,create_Integer(47),create_String("b"));

val = get_Map(m,create_Integer(28));
```

Map m changes during the run-time this way:

1. `{ 28 → "a" }`
2. `{ 28 → "a", 47 → "b" }`

A value of variable *`val`* after the call of `get_Map` will be `"a"`.

getKey_Map

The function returns a specification reference to some key,to which corresponds the given value.

```
Object* getKey_Map( Map* self, Object* value )
```

Parametrs

`self`

A map, an element of which should be returned by the function.

`value`

A key, corresponded to the needed value.

Return value

One of the keys corresponded to the given value is returned. If a map has no such key `NULL` is returned.

Additional information

If domain is typified than the type of element *`value`* must be the same as the type of map's values.

```
bool equ;

Map* m = create_Map(&type_Integer, &type_String);

Object* key;

Object* val;
```

```
put_Map(m,create_Integer(28),create_String("a"));

put_Map(m,create_Integer(47),create_String("a"));

val = create_String("a");

key = getKey_Map(m,val);

equ = equals( get_Map(m,key), val);
```

Map m changes during the run-time this way:

1. `{ 28 → "a" }`
2. `{ 28 → "a", 47 → "a" }`

A value of specification reference *`key`* after the call of `getKey_Map` will be `28` or `47`. A value of variable *`equ`* after the call of `equals` `true`.

isEmpty_Map

The function checks whether the map is empty.

```
bool isEmpty_Map( Map* self )
```

Parametrs

`self`

An under-test map.

Return value

`True is returned`, if a map has no pair “key-value”; otherwise `false` is returned.

Additional information

```
bool empty;

Map* m = create_Map(&type_Integer, &type_String);

empty = isEmpty_Map(m);

put_Map(m,create_Integer(28),create_String("a"));

empty = isEmpty_Map(m);
```

A value of variable *`empty`* after the first call of `isEmpty_Map` will be `true`, and after the second one (when pair `28 → "a"` will be added to the map) — `false`.

key_Map

The function returns a key of the map at the given position.

```
Object* key_Map( Map* self, int index )
```

Parametrs

self

A map, a key of which the function should return.

index

A key's number, which should be returned by the function.

Return value

A specification reference to the map's key at the *index* position.

Additional information

This function is intended for enumeration all the keys of the map. Since a key set is not ordered the elements are numbered in a random order. A number of the position must be in the range from 0 to the size of the map – 1, i.e. `0 ≤ index < size_Map(self)`.

```
Map* m1 = create_Map(&type_Integer, &type_String);

Map* m2 = create_Map(&type_Integer, &type_String);

int i;

bool equ;

put_Map(m1,create_Integer(28),create_String("a"));

put_Map(m1,create_Integer(47),create_String("b"));

for (i = 0; i < size_Map(m1); i++) {

  Object* key = key_Map(m1,i);

  Object* val = get_Map(m1,key);

  put_Map(m2,key,val);

}

equ = equals(m1,m2);
```

Map *m1* changes during the run-time like this:

1. `{ 28 → "a" }`
2. `{ 28 → "a", 47 → "b" }`

Map *m2* `changes by the analogy`, but since the keys are not ordered in the map the order of adding “key-value” pairs can't be described definitely. A value of variable *equ* after the call of `equals` will be `true`.

keyType_Map

The function returns a pointer to a constant descriptor of a specification data type that constrains keys of this map.

```
Type *keyType_Map( Map* self )
```

Parametrs

self

A specification reference to the given map.

Return value

The function returns a pointer to the constant descriptor of a specification type, by which the map's keys are limited.

Additional information

If the domain of the map is not typified then NULL is returned.

```
Map* m = create_Map(&type_Integer, &type_String);

Type *t = keyType_Map(m);
```

A specification reference value *t* after the call of function keyType_Map will be &type_Integer.

put_Map

The function adds a “key–value” pair to the map.

```
Object* put_Map( Map* self, Object* key, Object* value )
```

Parametrs

self

A map, to which a “key-value” pair is added.

key

A key of an added pair.

value

A value of an added pair.

Return value

If a map has no key *key*, the function returns NULL. If a map has key *key*, then the function returns an old value corresponded to it.

Additional information

If a definitional domain is typified then the type of element key should be the same as the type of map's keys. If domain is typified than the type of element *value* must be the same as the type of map's values.

If a map has no key *key*, then a key *key* and a corresponded value *value* are added to the map. If a map has key *key*, then the function returns an old value corresponded to it and replaces an old value with a new one in a map.

```
String *val;

Map* m = create_Map(&type_Integer, &type_String);
```

```
val = put_Map(m,create_Integer(28),create_String("a"));

val = put_Map(m,create_Integer(47),create_String("b"));

val = put_Map(m,create_Integer(28),create_String("c"));
```

Map m changes during the run-time this way:

1. `{ 28 → "a" }`
2. `{ 28 → "a", 47 → "b" }`
3. `{ 28 → "c", 47 → "b" }`

Value *val* after each of the first two calls of `put_Map` is `NULL`; after the third call it goes to `"a"`.

putAll_Map

Adds to the first map all the pairs “key-value” from the second one.

```
void putAll_Map( Map* self, Map* t )
```

Parametrs

self

A map, to which elements are added.

t

A map, elements of which are added to *self*.

Additional information

If a range of definition of map *self* is typified then types of all the keys of map *t* must be the same as types of the keys of map *self*. In a same manner, if a domain of map *self* is typified, then types of all the values of map *t* must be the same as types of the values of map *self*. Range of definition and range of values of `t` map are not required to be typified.

If map *self* has already got a key from map *t*, then a value corresponded to it replaces with a new one.

```
Map* m1 = create_Map(&type_Integer, &type_String);

Map* m2 = create_Map(NULL, NULL);

put_Map(m1,create_Integer(28),create_String("a"));

put_Map(m1,create_Integer(63),create_String("b"));

put_Map(m2,create_Integer(28),create_String("c"));

put_Map(m2,create_Integer(47),create_String("d"));

putAll_Map(m1,m2);
```

Map *m1* changes like this:

1. `{ 28 → "a" }`
2. `{ 28 → "a", 63 → "b" }`

Map *m2* changes like this:

1. `{ 28 → "c" }`
2. `{ 28 → "c", 47 → "d" }`

After the call of `putAll_Map` pair `47 → "d"` is added to map *m1* and a value to key `28` changes to `"c"`:

3. `{ 28 → "c", 47 → "d", 63 → "b" }`

remove_Map

The function removes from the map a pair “key-value” for the given key.

```
Object* remove_Map(Map* self, Object* key);
```

Parametrs

self

A map, from which an element is removed.

key

A key of a removed pair “key-value”.

Return value

Returns a value, which corresponded to key *key* in map *self*, or `NULL`, if there was no such value.

Additional information

```
Map* m = create_Map(&type_Integer, &type_String);

put_Map(m,create_Integer(28),create_String("a"));

remove_Map(m,create_Integer(28));

remove_Map(m,create_Integer(28));
```

The first call of function `remove_Map` returns a string representation corresponded to key `28`: `"a"`. The second call returns `NULL`, because the pair with key `28` was already removed.

size_Map

The function returns the size of the map.

```
int size_Map( Map* self )
```

Parametrs

self

A map, the size of which must be returned by the function.

Return value

The function returns the exact number of pairs in a map.

Additional information

A size is considered to be a number of pairs “key-value”, which a map has.

```
int size;

Map* m = create_Map(&type_Integer, &type_String);

size = size_Map(m);

put_Map(m,create_Integer(28),create_String("a"));

size = size_Map(m);
```

After the first call of `size_Map` variable *`size`* goes to NULL. Ater the second call of `size_Map` variable goes to `1`, as one element was added to the map.

valueType_Map

The function returns a pointer to a constant descriptor of a specification data type, that constrains values of the map.

```
Type *valueType_Map( Map* self )
```

Parametrs

`self`

A specification reference to the given map.

Return value

The function returns a reference to a constant descriptor of a specification type, by which values of the given map are limited.

Additional information

If a range of definition of a map is not typified then `NULL` is returned.

```
Map* m = create_Map(&type_Integer, &type_String);

Type *t = valueType_Map(m);
```

Value of specification reference *`t`* after the call of function `valueType_Map` will be `&type_String`.

14.2.15 MultiSet

Type `MultiSet` is a container type, implementing a multiset of elements(i.e. a set, elements of which can repeat).

Any specification references can be elements of a multiset. ype of elements can be constrained when creating a multiset. Such a multiset is called typified, and all functions, related to the typified multiset, will check that `Object*` parameter actual type is equal to the data type of multiset's elements.

Two multisets are equal if they have the same elements (with regard to repetition factor). Typification of multisets is not considered at that. Particularly, empty multisets are always equal.

Header file: atl/multiset.h

add_MultiSet
The function inserts the given element into the multiset.

addAll_MultiSet
Adds the elements of one multiset to another one.

clear_MultiSet
The function removes all the elements from the multiset.

contains_MultiSet
The function returns a repetition factor of an element, equals to the given one, in a multiset.

containsAll_MultiSet
checks whether one set is a subset of another one, in other words, checks whether first set contains all the elements of the second one (with regard to repetition factor).

create_MultiSet
The function creates a multiset and returns a specification reference of MultiSet data type.

elemType_MultiSet
The function returns a reference to a constant descriptor of a specification type, hat constrains the multiset's elements.

get_MultiSet
The function returns a specification reference to the element with `the given` number; elements are numerated in a random order .

isEmpty_MultiSet
The function hecks whether the multiset is empty.

remove_MultiSet
The function removes an element from the multiset once (i.e. with regard to repetition factor of this element's entry in the multiset descend by 1).

removeAll_MultiSet
Subtracts one multiset from another one: removes from the first multiset elements, that belong to the second one (with regard to repetition factor).

removeCount_MultiSet
The function decreases the repetition factor of the given element's entry by the given quantity.

removeFull_MultiSet
The function totally removes an element from multiset (i.e. the repetition factor of this elements entry in multiset goes to 0).

retainAll_MultiSet

Gets an intersection of the multisets: retains in the first multiset only such elements, which also belong to the second one (with regard to repetition factor).

size_MultiSet
The function returns the number of the elements in the multiset.

toList_MultiSet
The function returns a List containing all the elements of the given multiset.

toSet_MultiSet
The function returns a Set, containing all the elements of the given multiset.

add_MultiSet

The function inserts the given element into the multiset.

```
bool add_MultiSet( MultiSet* self, Object* ref )
```

Parametrs

self

A multiset where elements are inserted.

ref

A value of a specification type which is inserted in the multiset.

Return value

`true`, if the insertion was successful.In any other case — `false`.

Additional information

If the multiset is typified, then the type of element *ref* should be the same as the type of the multiset's elements.

```
MultiSet* s = create_MultiSet(&type_Integer);

add_MultiSet(s, create_Integer(28));

add_MultiSet(s, create_Integer(47));

add_MultiSet(s, create_Integer(28));
```

The multiset s is changing during the run-time this way:

1. `{ 28 }`
2. `{ 28, 47 }`
3. `{ 28, 47, 28 }`

addAll_MultiSet

Adds the elements of one multiset to another one.

```
bool addAll_MultiSet( MultiSet* self, MultiSet* set )
```

Parametrs

self

A multiset, in which elements are added.

set

A multiset, elements of which are added to *self*.

Return value

If during the unification even one element was added to *self* , the function returns true.In any other case — false.

Additional information

If multiset *self* is typified, then the types of all the elements of multiset *set* must be the same as the type of multiset's *self* elements (*set* itself is not required to be typified).

```
bool changed;

MultiSet* s1 = create_MultiSet(&type_Integer);

MultiSet* s2 = create_MultiSet(NULL);

add_MultiSet(s1,create_Integer(28));

add_MultiSet(s1,create_Integer(47));

add_MultiSet(s2,create_Integer(28));

add_MultiSet(s2,create_Integer(47));

add_MultiSet(s2,create_Integer(63));

changed = addAll_MultiSet(s1,s2);
```

multiset *s1* changes like this:

1. { 28 }
2. { 28, 47 }

multiset *s2* changes like this:

1. { 28 }
2. { 28, 47 }
3. { 28, 47, 63 }

after the call of addAll_MultiSet multiset *s1* changes like this:

3. { 28, 47, 28, 47, 63 }

and variable *changed* go to true.

clear_MultiSet

The function removes all the elements from the multiset.

```
void clear_MultiSet( MultiSet* self )
```

Parametrs

self

A cleared multiset.

Additional information

```
MultiSet* s = create_MultiSet(&type_Integer);

add_MultiSet(s,create_Integer(28));

add_MultiSet(s,create_Integer(47));

clear_MultiSet(s);
```

multiset s is changing during the run-time this way:

1. { 28 }
2. { 28, 47 }
3. {}

The same result can be gotten by recreating the multiset:

```
MultiSet* s = create_MultiSet(&type_Integer);

...

s = create_MultiSet(elemType_MultiSet(s));
```

But this result is less effective than the call of function `clear_MultiSet`.

contains_MultiSet

The function returns a repetition factor of an element, equals to the given one, in a multiset.

```
int contains_MultiSet( MultiSet* self, Object* ref )
```

Parametrs

self

An under-test multiset.

ref

A value of a specification type a search of which occurs in the multiset.

Return value

A repetition factor of element's *ref* entry in multiset *self* .

Additional information

If the multiset is typified, then the type of element *ref* should be the same as the type of the multiset's elements.

```
int contains;

MultiSet* s = create_MultiSet(&type_Integer);

add_MultiSet(s,create_Integer(28));

add_MultiSet(s,create_Integer(47));

add_MultiSet(s,create_Integer(28));

contains = contains_MultiSet(s,create_Integer(28));
```

Multiset s is changing during the run-time this way:

1. `{ 28 }`
2. `{ 28, 47 }`
3. `{ 28, 47, 28 }`

A value of variable *contains* after the call of `contains_MultiSet` will be `2`.

containsAll_MultiSet

Checks whether one multiset is a submultiset of another one, in other words, checks whether the first multiset contains all the elements of the second one.

```
bool containsAll_MultiSet( MultiSet* self, MultiSet* set )
```

Parametrs

self

A multiset, which is to be checked whether it contains elements of multiset *set*.

set

A priori submultiset of *self*.

Return value

`ttrue, if all of the elements of multiset` *set* `belong to the multiset` *self*`. false` in any other case.

Additional information

```
bool contains;

MultiSet* s1 = create_MultiSet(&type_Integer);

MultiSet* s2 = create_MultiSet(NULL);

add_MultiSet(s1,create_Integer(28));

add_MultiSet(s1,create_Integer(47));

add_MultiSet(s2,create_Integer(28));

add_MultiSet(s2,create_String("a"));

add_MultiSet(s2,create_Integer(47));
```

```
contains = containsAll_MultiSet(s1,s2));

contains = containsAll_MultiSet(s2,s1));
```

multiset *s1* changes like this:

1. `{ 28 }`
2. `{ 28, 47 }`

multiset *s2* changes like this:

1. `{ 28 }`
2. `{ 28, a }`
3. `{ 28, a, 47 }`

After the first call of `containsAll_MultiSet` variable *contains* go to `false`, and after the second call — `true`.

create_MultiSet

The function creates a multiset and returns a specification reference of MultiSet data type.

```
MultiSet* create_MultiSet( const Type *elem_type )
```

Parametrs

elem_type

A pointer to constant descriptor of the multiset's elements' type.

Return value

A specification reference of the type `MultiSet` to the just created multiset.

Additional information

If parameter *elem_type* is NULL, than types of multiset's elements are not limited.

```
MultiSet* s1 = create_MultiSet(NULL);

MultiSet* s2 = create_MultiSet(&type_Integer);
```

After the first call of `create_Multiset` was created a multiset, in which any elements can be added. A multiset created by the second call can store elements of `Integer` type only.

The function `create_MultiSet` defined along with standart function `create`.

```
MultiSet* s1 = create(&type_MultiSet, NULL);

MultiSet* s2 = create_MultiSet(NULL);
```

The two shown methods of creating a specification reference are functionally equivalent, but the use of create is not type-safe and cause a compiler warning (see function create definition).

elemType_MultiSet

The function returns a reference to a constant descriptor of a specification type, hat constrains the multiset's elements.

```
Type *elemType_MultiSet(MultiSet* self)
```

Parametrs

self

A multiset , limitations of which the function must return.

Return value

A pointer to constant descriptor of specification data type by which values of this multiset are limited.
If the multiset is not typified (during the creation there was no limitations of the elements' type), the function returns `NULL`.

Additional information

```
MultiSet* s = create_MultiSet(&type_Integer);

Type *t = elemType_MultiSet(s);
```

Variable *t* gets value `&type_Integer`.

get_MultiSet

The function returns a specification reference to the element with the given number; elements are numerated in a random order .

```
Object* get_MultiSet( MultiSet* self, int index )
```

Parametrs

self

A multiset, an element of which the function should return.

index

An element number, which should be returned by the function.

Return value

A specification reference of the type `Object`.

Additional information

This function is intended for enumeration of multiset's elements. Since a multiset is not ordered, elements are numerated in a random order. A number of position must be in the range from 0 to the size of the multiset – 1, i.e. $0 \le index < size\_MultiSet(self)$.

```
MultiSet* s1 = create_MultiSet(&type_Integer);

MultiSet* s2 = create_MultiSet(&type_Integer);

int i;
```

```
bool equ;

add_MultiSet(s1,create_Integer(28));

add_MultiSet(s1,create_Integer(47));

add_MultiSet(s1,create_Integer(63));

for(i=0; i < size_MultiSet(s1); i++)

  add_MultiSet(s2, get_MultiSet(s1,i));

equ = equals(s1,s2);
```

Multiset *s1* changes during the run-time this way:

1. `{ 28 }`
2. `{ 28, 47 }`
3. `{ 28, 47, 63 }`

A value of variable *equ* after the call of `equals` will be `true`, since after an iteration all the elements of *s1* will be added to *s2*, i.e the multisets will be equal.

isEmpty_MultiSet

The function hecks whether the multiset is empty.

```
bool isEmpty_MultiSet( MultiSet* self )
```

Parametrs

self

An under-test multiset.

Return value

`true`, if there is no elements in the multiset and `false` otherwise.

Additional information

```
bool empty;

MultiSet* s = create_MultiSet(&type_Integer);

empty = isEmpty_MultiSet(s);

add_MultiSet(s,create_Integer(28));

empty = isEmpty_MultiSet(s);
```

A value of variable *empty* After the first call of `isEmpty_MultiSet` will be `true`, and after the second one (when number 28 will be added) — `false`.

remove_MultiSet

The function removes an element from the multiset once (i.e. with regard to repetition factor of this element's entry in the multiset descend by 1).

```
bool remove_MultiSet( MultiSet* self, Object* ref )
```

Parametrs

self

A multiset from which elements should be deleted.

ref

An element which is to be deleted from the multiset.

Return value

If during the subtraction from multiset *self* at list one element was deleted, the function returns `true`.In any other case — `false`.

Additional information

If the multiset is typified, then the type of element *ref* shoud be the same as the type of the multiset's elements.

```
bool changed;

MultiSet* s = create_MultiSet(&type_Integer);

add_MultiSet(s,create_Integer(28));

add_MultiSet(s,create_Integer(47));

add_MultiSet(s,create_Integer(28));

changed = remove_MultiSet(s,create_Integer(28));

changed = remove_MultiSet(s,create_Integer(63));
```

Multiset s is changing during the run-time this way:

1. `{ 28 }`
2. `{ 28, 47 }`
3. `{ 28, 47, 28 }`
4. `{ 47, 28 }`

A value of variable *changed*. After the first call of `remove_MultiSet` will be `true`, and after the second one — `false`.

removeAll_MultiSet

Subtracts one multiset from another one: removes from the first multiset elements, that belong to the second one.

```
bool removeAll_MultiSet( MultiSet* self, MultiSet* set )
```

Parametrs

self

A multiset from which elements should be deleted.

set

A multiset elements of which are deleted from *self*.

Return value

If during the subtraction from multiset *self* at list one element was deleted, the function returns `true`. In any other case — `false`.

Additional information

```
bool changed;

MultiSet* s1 = create_MultiSet(&type_Integer);

MultiSet* s2 = create_MultiSet(NULL);

add_MultiSet(s1,create_Integer(28));

add_MultiSet(s1,create_Integer(47));

add_MultiSet(s1,create_Integer(28));

add_MultiSet(s1,create_Integer(63));

add_MultiSet(s2,create_Integer(28));

add_MultiSet(s2,create_Integer(47));

changed = removeAll_MultiSet(s1,s2);
```

Multiset *s1* changes like this:

1. `{ 28 }`
2. `{ 28, 47 }`
3. `{ 28, 47, 28 }`
4. `{ 28, 47, 28, 63 }`

Multiset *s2* changes like this:

1. `{ 28 }`
2. `{ 28, 47 }`

after the call of `removeAll_MultiSet` in multiset *s1* elements `28` (once) and `63 remain`, and variable *changed* go to `true`.

removeCount_MultiSet

The function decreases the repetition factor of the given element's entry by the given quantity.

```
int removeCount_MultiSet( MultiSet* self, Object* ref, int count )
```

Parametrs

self

A multiset from which elements should be deleted.

ref

An element which is to be deleted from the multiset.

count

The number on which the repetition factor of the given element's entry should be lessened.

Return value

A number of really deleted entries of element *ref* in multiset *self*.

Additional information

If the multiset is typified, then the type of element *ref* shoud be the same as the type of the multiset's elements.

```
int changed;

MultiSet* s = create_MultiSet(&type_Integer);

add_MultiSet(s,create_Integer(28));

add_MultiSet(s,create_Integer(47));

add_MultiSet(s,create_Integer(28));

add_MultiSet(s,create_Integer(28));

changed = removeCount_MultiSet(s,create_Integer(28),2);

changed = removeCount_MultiSet(s,create_Integer(28),2);
```

Multiset s is changing during the run-time this way:

1. { 28 }
2. { 28 47 }
3. { 28 47 28 }
4. { 28 47 28 28 }
5. { 47 28 }
6. { 47}

After the first call of `removeCount_MultiSet` variable *changed* goes to `2`, and sfter the second one — `1`.

removeFull_MultiSet

The function totally removes an element from multiset (i.e. the repetition factor of this elements entry in multiset goes to 0).

```
int removeFull_MultiSet( MultiSet* self, Object* ref )
```

Parametrs

self

A multiset from which elements should be deleted.

ref

An element which is to be deleted from the multiset.

Return value

A number of really deleted entries of element *ref* in multiset *self*.

Additional information

If the multiset is typified, then the type of element *ref* shoud be the same as the type of the multiset's elements.

```
int changed;

MultiSet* s = create_MultiSet(&type_Integer);

add_MultiSet(s,create_Integer(28));

add_MultiSet(s,create_Integer(47));

add_MultiSet(s,create_Integer(28));

changed = removeFull_MultiSet(s,create_Integer(28));
```

Multiset s is changing during the run-time this way:

1. { 28 }
2. { 28, 47 }
3. { 28, 47, 28 }
4. { 47 }

after the call of `removeFull_MultiSet` variable *changed* goes to `2`.

retainAll_MultiSet

Gets an intersection of the multisets: retains in the first multiset only such elements, which also belong to the second one (with regard to repetition factor).

```
bool retainAll_MultiSet( MultiSet* self, MultiSet* set )
```

Parametrs

self

A first argument of the multisets' intersection. This multiset will store the result of the intersection.

set

A second argument of the multisets' intersection.

Return value

`true`, if the number of elements in the intersection is less than the number of elements in *`self`*.In any other case — `false`.

Additional information

```
bool changed;

MultiSet* s1 = create_MultiSet(&type_Integer);

MultiSet* s2 = create_MultiSet(NULL);

add_MultiSet(s1,create_Integer(28));

add_MultiSet(s1,create_Integer(47));

add_MultiSet(s1,create_Integer(28));

add_MultiSet(s1,create_Integer(63));

add_MultiSet(s2,create_Integer(28));

add_MultiSet(s2,create_Integer(47));

changed = retainAll_MultiSet(s1,s2);
```

Multiset *`s1`* changes like this:

1. `{ 28 }`
2. `{ 28, 47 }`
3. `{ 28, 47, 28 }`
4. `{ 28, 47, 28, 63 }`

Multiset *`s2`* changes like this:

1. `{ 28 }`
2. `{ 28, 47 }`

after the call of `retainAll_MultiSetin` set *`s1`* elements *`28`* and *`47`* remain,and variable *`changed`* go to true.

size_MultiSet

The function returns the number of the elements in the multiset.

```
int size_MultiSet( MultiSet* self )
```

Parametrs

`self`

A multiset, power of which the function should return.

Return value

An exact number of elements in multiset *self*.

Additional information

```
int size;

MultiSet* s = create_MultiSet(&type_Integer);

size = size_MultiSet(s);

add_MultiSet(s,create_Integer(28));

add_MultiSet(s,create_Integer(28));

size = size_MultiSet(s);
```

After the first call of `size_MultiSet` variable *size* go to zero. Ater the second call of `size_MultiSet` the variable is `2`: two (equal) elements were added to the multiset.

toList_MultiSet

The function returns a `List` containing all the elements of the given multiset.

```
List* toList_MultiSet( MultiSet* self )
```

Parametrs

self

A given multiset.

Return value

A specification reference of the type `List`.

Additional information

Order of the elements in the list is not defined. The returned list has the same typification as the multiset: if the elements of the multiset was constrained by a data type, the elements of the list will be constrained by the same type.

```
MultiSet* s = create_MultiSet(&type_Integer);

List* l;

Type *t;

add_MultiSet(s,create_Integer(28));

add_MultiSet(s,create_Integer(47));

add_MultiSet(s,create_Integer(28));

l = toList_MultiSet(s);

t = elemType_MultiSet(l);
```

Multiset s is changing during the run-time this way:

1. `{ 28 }`
2. `{ 28, 47 }`
3. `{ 28, 47, 28 }`

`List l`, returned by function `toList_MultiSet`, has elements: `< 28, 47, 28 >`; has elements `elemType_MultiSet(l)` returns `&type_Integer`.

toSet_MultiSet

The function returns a `Set`, containing all the elements of the given multiset.

```
Set* toSet_MultiSet( MultiSet* self )
```

Parametrs

self

A given multiset.

Return value

A specification reference of the type `Set`.

Additional information

The returned list has the same typification as the multiset: if the elements of the multiset was constrained by a data type, the elements of the list will be constrained by the same type.

```
MultiSet* ms = create_MultiSet(&type_Integer);

Set* s;

Type *t;

add_MultiSet(ms,create_Integer(28));

add_MultiSet(ms,create_Integer(47));

add_MultiSet(ms,create_Integer(28));

s = toSet_MultiSet(ms);

t = elemType_MultiSet(s);
```

Multiset *ms* is changing during the run-time this way:

1. `{ 28`
2. `{ 28, 47 }`
3. `{ 28, 47, 28 }`

Multiset *s*, returned by function `toSet_MultiSet`, has elements: `{ 28, 47 }`. The call of `elemType_MultiSet(s)` returns `&type_Integer`.