CTesK 2.2:
SeC Language Reference

Contents

Alphabetical Index

Introduction

SeC[1] is a *S*pecification *E*xtension of *C* programming language. It is developed specially for supporting testing C software with CTesK based on UniTesK testing method. Below main features of the method are presented.

Testing servers for checking and demonstration of a proper software behavior meeting requirements to the software. To be representative testing should cover a sufficient set of various situations and conditions under that a software is checked.

UniTesK testing method is a method for automated development tests checking whether a system under test meets functional requirements, that define what externally observable effects the system must produce when interacting with the environment by means of an interface of the system.

Functional requirements are described in the form of specification, which defines an interface of the system under test and correct results producing by means of each element of the interface. Specification is defined in computer readable form that is called formal specification. For C software specification SeC is used. CTesK provides generation from SeC specification oracles on C. Oracles are test components checking the system under test behavior for meeting specified requirements.

For functional testing not only the implementation code coverage but a coverage of the functionality of the system under test is important. SeC allows defining coverage criteria for functional requirements described in specification. Based on them CTesK provides automatic estimation of functional requirement coverage during testing.

For construction of relevant and representative set of various situations and conditions, under that the behavior of the system under test is checked, UniTesK uses FSM models of the system under test. SeC allows describing FSM models in the form, which is simple and compact and allows reuse. CTesK provides the test engine implemented in C, which builds relevant and representative test sequence using test component generated form SeC description of FSM model.

[1] Pronounced as [sek]

General information about SeC

SeC[2] is a *S*pecification *E*xtension of *C* programming language. It is used for automated test development based on computer readable specifications of software requirements. Additional SeC constructions are compact and suitable for describing software requirements and test components. It allows automating test development and reducing costs for training developers skilled in C language.

SeC introduces specification types (see *Specification types*), invariants of types and global variables, test scenarios (see *Test scenarios*) and three kinds of functions: specification (see *Specification functions, Deferred reactions*), mediator (see *Mediator functions*) and scenario (see *Scenario functions*) functions. They are defined in specification files with extension **.sec** and declared in specification header files with extension **.seh**. Specification header files are included into specification files by means of C preprocessor directive `#include`. Specification files can contain ordinary C functions for auxiliary needs. For using in specification files special external types or functions ordinary C headers files are included by means of C preprocessor directive `#include`.

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 `?:`. An 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`.

[2] Ponounced as [sek]

Specifications

Specifications are formal requirements for a system under test in form of data invariants and behavior description.

Specifications can utilize either internal data of a system under test or its own specification data model for description of requirements. Mediators are used to bind specification model data, specification functions, and deferred reactions to data, functions, and reactions of a system under test.

Specification types

Purpose

Specification data type combines a data type of C with basic functions for dealing with it: creating, initializing, copying, comparing, stringifying, destroying.

Description

```
specification typedef base_type new_type =
{
  .init       = pointer_to_initializing_function
, .copy       = pointer_to_copying_function
, .compare    = pointer_to_comparing_function
, .to_string  = pointer_to_stringifying_function
, .enumerator = pointer_to_enumerating_function
, .destroy    = pointer_to_destroying_function
};
```

Specification data types are defined by usual C-like `typedef` construction marked by `specification` SeC keyword. This construction can contain several declarations with or without

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 constraints

- Specification data types cannot be local.
- Names of specification data types belong to the same namespace as typedef-names.
- Definition of specification data type with a given name (declarator with initializer) can occur only once in all translation units (`translation_unit`).
- 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 (initialize given <data> area) */
typedef void (*Init)( Object* ref, va_list* arg_list );

/* Object copier type (copy <data> from 'src' to 'dst') */
typedef void (*Copy)( Object* src, Object* dst );

/* Object comparer type (compare <data> of 'left' and 'right') */
typedef int (*Compare)( Object* left, Object* right );

/* Object stringifier type (make string representation of <data>) */
typedef String (*ToString)( Object* obj );
```

```
/* Subobjects enumerator type
   (enumerate objects belonging to the given one) */
typedef void (*Enumerate)
  (Object* obj, void (*callback)(void* ref,void* par),
   void* par
  );
/* Object destructor type (free resources allocated by object) */
typedef void (*Destroy)( Object* obj );
```

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

- 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).
- The following data types are prohibited to be used as base types for specification data types:
  1. specification, functional, and incomplete,
  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:
  3. unions and arrays of variable length,
  4. structures and arrays, containing all above-listed data typed,
  5. pointers to data types, listed in 2, 3, and 4.
- If specification data type declaration contains keyword `invariant`, an invariant must be declared for this data type (see Invariants of data types).
- If at least one specification data type declaration contains `invariant` keyword, all declarations and definition must contain this keyword.
- Specification data types can be used only by reference, like incomplete data types of C. The following are prohibited: declarations of variables of specification data types; unions, structures, and arrays, containing fields or elements of specification data types; and so on.

Example

Declaration:

```
specification typedef struct {int* x, int* y} XY;
```

Definition:

```
specification typedef struct {int* x, int* y} XY =
{
  .init = initXY
, .copy = copyXY
, .compare = compareXY
, .to_string = to_stringXY
, .destroy = destroyXY
};
```

Invariants of types

Purpose

Invariants of data types are intended for description of data types constraints, used in specifications. Data type invariant can be considered as common part of all pre- and postconditions of functions depending on values of this data type. On the other hand, data type invariant can be considered as subtype constraints ensuring data integrity.

Description

Definition of data type with invariant:

```
invariant typedef base_type subtype;
```

Definition of invariant itself:

```
invariant (subtype param)
{
  ...
  return boolean_value;
  ...
}
```

Invariant call:

```
invariant(value_to_check)
```

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

Definition of data type with invariant:

```
declaration ::=   (( declaration_specifiers )?
                    "invariant"
                    ( declaration_specifiers )?
                    "typedef"
                    ( declaration_specifiers )?
                  )
```

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

        ;
```

Definition of data type invariant:

```
"invariant" "(" parameter_declaration ")" compound_statement ;
```

Invariant call:

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

Semantic constraints

- Data type invariant cannot be defined for a local data type.
- 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`).
- 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.
- Functional data types and `void` are prohibited as base types for data types with invariant.
- 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.
- Data type invariant call expression must contains in parenthesis an expression of the data type being checking.

Example

Declaration:

```
invariant typedef int Nat;
```

or

```
invariant specification typedef int Nat;
```

Definition:

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

or

```
invariant ( Nat* n )
{
```

```
  return *n > 0;
}
```

Call:

```
Nat n = 1;
...
invariant (n);
```

Invariants of variables

Purpose

Invariants of variables are intended for describing constraints of values of global variables, used in specifications. Variable invariant can be considered as common part of all pre- and postconditions of functions depending on values of this variables. On the other hand, variable invariant can be considered as variable constraints ensuring data integrity.

Description

Declaration of variable with invariant:

```
invariant data_type variable;
```

Definition of invariant:

```
invariant (variable)
{
  ...
  return boolean_value;
  ...
}
```

Invariant call:

```
invariant(variable_name)
```

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. 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 and definition of variable with invariant:

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

Definition of variable invariant:

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

Variable invariant call:

```
"invariant" "(" ( assignment_expr ( "," assignment_expr )* )? ")"
```

Semantic constraints

- Variable invariants can be defined only for global variables.
- 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_specifiers`).
- If definition or at least one declaration of variable contains `invariant` SeC specifier (`se_declaration_specifiers`), 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.
- Variable invariant definition and call expression must contain in parenthesis the name of the appropriate variable as the only parameter.
- Invariant body must syntactically and semantically be equal to a function body without side effects, without parameters, and returning a boolean value.

Example

Declaration of variable with invariant:

```
invariant int EvenNum = 2;
```

Definition:

```
invariant ( EvenNum )
{
  return EvenNum % 2 == 0;
}
```

Call:

```
invariant (EvenNum);
```

Specification functions

Purpose

Specification functions are intended for describing behavior of the system under test, experiencing external actions throw a part of its interface. Specification functions describes behavior in form of constraints of access to data, preconditions, coverage criteria, and postconditions.

Description

Declaration:

```
specification signature access_constraints;
```

Definition:

```
specification signature access_constraints
{
  auxiliary_code
  pre {...}
  {
    auxiliary_code
    coverage name_1 {...}
    ...
    coverage name_n {...}
    {
      auxiliary_code
      post {...}
      auxiliary_code
    }
    auxiliary_code
  }
  auxiliary_code
}
```

Call:

```
specification_function_name(arguments)
```

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 (see Access constraints).
- Precondition, describing when behavior of the system under test is defined (see Preconditions).
- 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 (see Coverage criteria).
- Postcondition, describing constraints of results of the system under test functioning, described by the specification function (see Postconditions).
- 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 constraints

- Names of specification functions belong to the same namespace as names of usual C functions.
- Specification function must be defined exactly once within all translation units (`translation_unit`).
- For every global variable and parameter (or their parts), used in a specification function, declaration of the function must contain appropriate access constraints (`se_access_description`, see Access constraints).
- Compound statement of a specification function must contain exactly one postcondition (`se_post_block_statement`, see Postconditions) after coverage criteria and precondition blocks, if they present.
- Compound statement of a specification function can contain no more than one precondition (`se_pre_block_statement`, see Preconditions) before coverage criteria (it they present) and postcondition blocks; precondition must be followed by compound statement, or first coverage criterion, or postcondition.
- Compound statement of a specification function can contain several coverage criteria (`se_coverage_block_statement`, see Coverage criteria) following one after another, after precondition (if it present) and before postcondition; there can be no auxiliary code between coverage criteria, last coverage criterion must be followed by compound statement or postcondition.
- Auxiliary code can occur in the compound statement of a specification function outside precondition, coverage criteria, and postcondition, in the following points:
    - before precondition or after coverage criteria, postcondition, and compound statements containing coverage criteria and postcondition,
    - in compound statement after precondition—before first coverage criterion or postcondition (if coverage criteria is omitted), or after coverage criteria, postcondition, and compound statements containing postcondition,
    - in compound statement after last coverage criterion—before or after postcondition.
- Specification function must have no side effects:
    - values of global variables and data, passed by reference, must not change,
    - dynamic storage, allocated in a specification function, must be freed at the same nesting level (in the same compound statement):
        - storage, allocated in precondition, coverage criteria, or postcondition, must be freed in the same block,
        - 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,
        - 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,

- storage, allocated in compound statement before postcondition, must be freed in the end of this compound statement after postcondition.

- Declaration of specification function must contain signature, marked by `specification` keyword, and access constraints (see Access constraints); access constraints must be equal in all declarations of this specification function.

Example

Declaration:

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

Definition:

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

Call:

```
sqrt_spec(5.2)
```

Deferred reactions

Purpose

Deferred reaction are intended for description of behavior of the system under test, responding with some delay for external actions. Deferred reactions describe behavior in form of constraints of access to data, preconditions, and postconditions.

Description

```
reaction signature access_constraints
{
  auxiliary_code
  pre {...}
  {
    auxiliary_code
    post {...}
    auxiliary_code
  }
```

```
    auxiliary_code
}
```

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 (see Access constraints).
- Precondition, describing when appearance of such reactions is possible (see Preconditions).
- Postcondition, describing constraints of global variables that must be satisfied after occurrence of this deferred reaction (see Postconditions).
- Auxiliary SeC code outside precondition and postcondition.

Syntax

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

Semantic constraints

- Names of deferred reactions belong to the same name space as names of usual C functions.
- Deferred reaction must be defined exactly once within all translation units (`translation_unit`).
- Deferred reactions must have no parameters and must return pointer to a specification data type.
- For every global variable (or its parts), used in a deferred reaction, declaration of the reaction must contain appropriate access constraints (`se_access_description`, see Access constraints).
- Compound statement of a deferred reaction must contain exactly one postcondition (`se_post_block_statement`, see Postconditions) after precondition block, if it present.
- Compound statement of a deferred reaction can contain no more than one precondition (`se_pre_block_statement`, see Preconditions) before postcondition block; precondition must be followed by compound statement or postcondition.
- Auxiliary code can occur in the compound statement of a deferred reaction outside precondition and postcondition, in the following points:
    - in compound statement of reaction—before precondition or after postcondition and compound statement containing postcondition,
    - in compound statement after precondition—before or after postcondition.
- Deferred reaction must have no side effects:
    - values of global variables must not change,
    - dynamic storage, allocated in a deferred reaction, must be freed at the same nesting level (in the same compound statement):

- storage, allocated in precondition or postcondition, must be freed in the same block,
- storage, allocated in the beginning of deferred reaction, must be freed in its ending after precondition, postcondition, and compound statement containing postcondition,
- storage, allocated in compound statement after precondition, must be freed in the end of this compound statement after postcondition.

- Declaration of deferred reaction must contain signature, marked by `reaction` keyword, and access constraints (see Access constraints); access constraints must be equal in all declarations of this deferred reaction.

Example

```
int last_message_id = 1;

reaction Integer* incoming_message()
  writes last_message_id
{
  post {return last_message_id == *incoming_message; }
}
```

Access constraints

Purpose

Access constraints are intended for description of the way in which specification functions and deferred reactions use global variables, parameters, and expressions (l-values) with them. Access constraints are used for checking system under test behavior correctness. SeC language supports three types of access: read, write, and update.

Description

```
reads   expression_1, ..., alias_name = expression_n, ...
writes  expression_1, ..., alias_name = expression_n, ...
updates expression_1, ..., alias_name = expression_n, ...
```

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 (see Aliases) can be defined in access constraints.

Syntax

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


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


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

Grammar of `se_access_alias` element can be found in Aliases section.

Semantic constraints

- In access constraints for a function or reaction the same expression cannot be used several times with different access modifiers.
- If an expression has write access (`writes` modifier), it cannot be used in function or reaction before `post` keyword.
- 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 then can be accessed via `@` SeC operator (see Preexpressions).
- If an expression has read-only access (`reads` modifier), its value is accessible everywhere inside function or reaction, and cannot change.

Example

```
invariant List *stck;
...
specification bool push_spec(int i)
  reads i
  updates stck
```

Aliases

Purpose

Aliases are intended to simplify reference to expressions with access constraints in specification functions and deferred reactions.

Description

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 constraints

- Alias identifier cannot coincide with parameters identifiers and must be unique within access constraints of specification function or deferred reaction.

Example

```
invariant List *stck;
...
specification bool pop_spec(int *item)
  updates stck
  updates i = *item
```

Preconditions

Purpose

Precondition of a specification function describes when behavior of the system under test is defined during interactions with it through a part of the interface, described by the function. During test run precondition is checked every time when appropriate function of the system under test is invoked. Violation of precondition indicates incorrectness of the test.

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.

Description

```
pre
{
  ...
  return boolean_value;
}
```

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 boolean value. These instructions must be enclosed in curly braces and marked by `pre` keyword.

Syntax

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

Semantic constraints

- Specification function or deferred reaction can contain no more than one precondition, defined before coverage criteria (if they present) and postcondition.
- Precondition can be omitted, that is equal to precondition, always returning true.
- Precondition must have no side effects, i.e. it cannot change values of global variables and data, passed by reference.
- 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.

- Precondition cannot use expressions with write access (i.e. defined in access constraints with `writes` specifier).

Example

```
specification double log ( double x )
    reads x
{
    pre { return x > 0; }
    post { ... }
}
```

Coverage criteria

Purpose

Coverage criteria are intended for criteria of requirements coverage description. Each coverage criterion partitions behavior of the system under test during interactions with it through a part of the interface, described by the specification function, into functionality branches. During test run coverage criteria are used for evaluating reached testing completeness.

Description

```
coverage name
{
  ...
  return {identifier, string_literal};
  ...
}
```

Coverage criterion 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 and returning special construction that looks like initialization of a structure with two fields. The construction consists of identifier and string literal, separated by comma and enclosed in curly braces. Instructions of coverage criterion must be enclosed in curly braces, marked by `coverage` keyword, and named.

To avoid repeated calculation of expressions that specify functional branches partition, construction `coverage(coverage_name)` can be used in following coverage criteria and postcondition. This construction evaluates to identifier of reached functional branch of the specified coverage criterion. It can be used in `if-else` and `switch` operators of C language.

Syntax

```
se_coverage_block_statement ::= ( "default" )?
                                "coverage"
                                <ID>
                                compound_statement
                              ;
```

Semantic constraints

- Specification function can contain several coverage criteria following one after another.
- Names of different coverage criteria must be unique within specification function.

- Coverage criteria must be defined after precondition (if it presents) and before postcondition.
- Coverage criteria can be omitted, that is equal to a single coverage criterion with a single functional branch.
- Instructions in coverage criterion must be syntactically and semantically equal to a function body with the same signature as the specification function and returning special construction that looks like initialization of a structure with two fields. The construction must consist of identifier and string literal, separated by comma and enclosed in curly braces. The same identifier or literal cannot be used for indicating different branches within the same coverage criterion.
- Coverage criterion must have no side effects, i.e. it cannot change values of global variables and data, passed by reference.
- Coverage criterion cannot use expressions with write access (i.e. defined in access constraints with `writes` specifier).
- Any set of parameters and global variables, allowable by precondition, must correspond to one of functional branches, defined in coverage criterion.

Example

```
specification bool pop_spec(int *item)
  updates stck
  writes i = *item
{
  coverage c {
    if (size_List(stck) == 0)
      return { empty, "empty stack" };
    else if (size_List(stck) == STACK_SIZE)
      return { full, "full stack" };
    else
      return { nonempty, "nonempty stack" };
  }
  post { ... }
}
```

Postconditions

Purpose

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.

Description

```
post
{
  ...
  return boolean_value;
  ...
}
```

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 constraints

- Specification function or deferred reaction must have exactly one postcondition.
- Postcondition must be defined after precondition (if it presents) and after the last coverage criterion (if coverage criteria present).
- 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.
- Postcondition must have no side effects, i.e. it cannot change values of global variables and data, passed by reference.
- Postcondition and code after postcondition can use the following additional constructions:
  - Preexpressions with `@` operator (see Preexpressions),
  - 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,
  - 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.

Example

```
specification bool pop_spec(int *item)
  updates stck
  updates i = *item
{
  coverage c { ... };
  post {
    if (size_List(@stck) == 0)
      return 0 == compare(@stck, stck) && !pop_spec;
                  /* compare returns 0 for equal parameters */
    else return (   0
                 == compare(stck,
                            subList_List(@stck,1,size_List(@stck))
                           )
                 && i == value_Integer(get_List(@stck, 0))
                 && pop_spec
                );
```

```
    }
}
```

Preexpressions

Purpose

Preexpressions 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.

Description

`@`*`expression`*

Entry of `post` keyword in a specification function is treated as a test action. Entry of `post` keyword in deferred reaction is treated as an occurrence of the reaction. All instructions before `post` keyword in the thread of execution are fulfilled before test action or occurrence of the reaction. Instructions after `post` keyword are fulfilled after test action or occurrence of the reaction.

Preexpressions are exceptions from this rule. Preexpressions are marked by prefix operator `@`. Identifiers of objects with write access cannot be used in preexpressions.

The following rules should be considered for preexpressions:

- `@` operator has the same priority as other unary operators,
- Preexpression must be computable directly before `post` keyword in the thread of execution. In particular, preexpressions can only contain identifiers of variables, declared before `post` keyword.

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 constraints

- `@` operator can be used only after `post` keyword in the thread of execution.

- Preexpression that is used in a postcondition, must be computable before this postcondition in the thread of execution.

Example

```
specification void deposit_spec (Account *acct, int sum)
  reads   sum
  updates balance = acct->balance
  updates *acct
{
  pre { return sum > 0; }
  post {
    return balance == @balance + sum;
  }
}
```

Mediators

Mediators are intended for binding specification to implementation of the system under test, or to specification of another level of abstraction. For convenience later in this reference we assume that a system under test means directly implementation of a system under test, or its specification of another level of abstraction.

Mediators do the following tasks: conversion of model representation of a test action into implementation representation and conversion of implementation representation of reaction into model representation, and synchronization of specification data model state with the system under test state. Mediators of deferred reactions do only synchronization of states.

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

Mediator functions

Purpose

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.

Description

```
mediator name for signature access_constraints
{
  auxiliary_code
  call { ... }
  state { ... }
```

```
  auxiliary_code
}
```

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:

- Call block (marked by `call` keyword), implementing behavior, described in the corresponding specification function, by means of performing test action.
- 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.
- Auxiliary code before first or after the last named block.

Syntax

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

Semantic constraints

- Mediator function names belong to the same namespace as names of usual C functions.
- Mediator function must be defined exactly once within all translation units (`translation_unit`).
- Specification function or deferred reaction must be declared before declaration or definition of corresponding mediator function.
- 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.
- Mediator of a specification function must contain call block; state block may be omitted.
- Mediator of a deferred reaction must contain state block; state block is not permitted.
- If mediator contains both call and state blocks, state block must follows immediately after call block.
- Auxiliary code is permitted before first and after the last named block.

Example

```
stack *impl_stack;

mediator push_media for specification bool push_spec(int i)
  updates stck
{
  call {
    return push(impl_stack,i);
  }
  state {
```

```
    int k;
    clear_List(stck);
    for (k = impl_stack->size; k > 0;
         append_List(stck, create_Integer(impl_stack->elems[--k]))
        );
  }
}
```

Call blocks

Purpose

Call blocks of mediator functions implement behavior of the system under test, described in corresponding specification functions, by means of performing test actions.

Description

```
call
{
  ...
  return value;
  ...
}
```

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.

Syntax

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

Semantic constraints

- Call block is the mandatory block of mediators of specification functions.
- Call block is not permitted in mediators of deferred reactions.
- 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.

Example

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

State blocks

Purpose

State blocks of mediator functions implement synchronization of specification data model state with the system under test state after performed test action or occurred deferred reaction.

Description

```
state
{
  ...
  return;
  ...
}
```

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 ;
```

Semantic constraints

- Instructions in state block must be syntactically and semantically equal to a function body with the same signature as the corresponding specification function, without return value.
- 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.
- 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.

Example

```
mediator pop_media for
specification bool pop_spec(int *item)
  updates stck
  writes i = *item
{
  call {
    return pop(impl_stack,item);
  }
  state {
    int k;
    clear_List(stck);
    for (k = impl_stack->size; k > 0;
         append_List(stck, create_Integer(impl_stack->elems[--k]))
        );
  }
}
```

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.

Test scenario

Purpose

Test scenario is intended for test building on basis of the given test engine and necessary data.

Description

Declaration:

```
scenario type identifier;
```

Definition:

```
scenario type identifier = initializer;
```

Call:

```
identifier( argc, argv );
```

In SeC language test scenario is defined in the same way as global variable which specifiers contain `scenario` SeC 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 constraints

- Names of test scenarios belong to the same namespace as names of usual global variable of C language.
- Test scenario must be defines exactly once within all translation units (`translation_unit`).
- Test scenario cannot be defined locally.
- Type of test scenario must be specified by an identifier defined by `typedef` construction.
- Identifier of test scenario type is a name of a test engine.
- 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
                        )
```

- Initializer data type must be compatible with test scenario type.
- If scenario type is a structure, it is possible to use dereferencing syntax of C99 standard in spite of actually used C standard.
- 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.
- Test scenario call evaluates to boolean value.

Example

Declaration:

```
scenario dfsm stack_scenario;
```

Definition:

```
scenario dfsm stack_scenario =
{
  .init = init_stack_scenario,
  .getState = get_stack_scenario_state,
  .actions = { push_scen, pop_scen, NULL },
  .finish = finish_stack_scenario
};


int main(int argc,char** argv)
{
  if (!stack_scenario(argc,argv))
    return 1;
  return 0;
}
```

Scenario functions

Purpose

Scenario functions are intended for description of test actions sequences to be executed in each test situation reached during the testing. Scenario functions can also perform correctness check of invoked system under test functions.

Description

```
scenario bool scenario_function ()
{
  ...
  return boolean_value;
  ...
}
```

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

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

Semantic constraints

- Names of scenario functions belong to the same namespace as names of usual functions of C language.
- Scenario function must be defined exactly once within all translation units (`translation_unit`).

- Scenario functions must have no parameters and must return a boolean value.
- Scenario functions cannot be invoked directly.
- Scenario functions can contain the following constructions:
  - Iteration operators (see Iteration statements),
  - State variables (see State variables).

Example

```
scenario bool push_scen() {
  iterate (int i = 0; i < 10; i++;) {
    push_spec(i);
  }
  return true;
}
```

Iteration statements

Purpose

Iteration statements are intended for enumeration of test actions in scenario functions.

Description

```
iterate (  expression_1
         ; expression_2
         ; expression_3
         ; expression_4
        )
```

Iteration statement syntactically looks like `for` statement of C language. 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.

The following iteration statement:

```
iterate(var_decl; control_expr; iteration_expr; filter_expr)
    iteration_body
```

is similar in a certain sense to the following C code:

```
var_decl;
for(; control_expr; iteration_expr) {
  if (!filter_expr) continue;
  iteration_body;
}
```

Iteration variable are not local variables, they are a kind of state variables (see 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 constraints

- Iteration statements can be used only in scenario functions.
- Controlling expression and filter expression must have `bool` data type, if they present.
- Iteration variable cannot be of incomplete or local data type and must be initialized.

Example

```
scenario bool push_scen() {
  iterate (int i = 0; i < 10; i++;) {
    push_spec(i);
  }
  return true;
}
```

State variables

Purpose

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.

Description

```
stable type variable = value;
```

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 constraints

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

Example

```
scenario bool fibbonachi_scen()
{
    stable int f1 = 0;
    stable int f2 = 1;
    iterate(int i = 1; i <= 10; i++;)
    {
        f2 = f2 + f1;
        f1 = f2 - f1;
        fibbonachi_spec(f1);
    }
    return true;
}
```

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.

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.

System functions

The following are system functions of SeC 2.2 language:

- setBadVerdict
- assertion

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.

Description

`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.

Additional information

Header file: ts/ts.h

Library: ts

See also

System functions

Example

```
/*
 * Example of setBadVerdict usage.
 */


/*
 * Implementation contains "errMsg" state variable,
 * which can possess limited number of values.
 */
const char* errMsg;

/*
 * "errMsg" variable are modelled by "errno" variable
 * of enumeration data type.
 */
enum ErrorKind { NoError, ErrorKind1, ErrorKind2 } errno;

/*
 * "updateSystemState" function synchronizes value of "errno"
 * model variable with value of "errMsg" implementation variable.
 */
void updateSystemState()
{
  if (errMsg == NULL) { errno = NoError; return; }
  if (strcmp(errMsg,"ErrorKind1Msg") == 0) { errno = ErrorKind1; return; }
  if (strcmp(errMsg,"ErrorKind2Msg") == 0) { errno = ErrorKind2; return; }
  setBadVerdict("errMsg has bad value" );
```

```
}

mediator function_media for specification void function_spec( void )
    updates errno
{
  call {
    function();
  }
  state {
    updateSystemState();
  }
}
```

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.

Description

`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.

Additional information

Header file: utils/assertion.h

Library: utils

See also

System functions, Tracing services

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

To determine time mark for the current moment of time, the following functions are used:

getCurrentTimeMark

setDefaultCurrentTimeMarkFunction

TSTimeModel

`TSTimeModel` enumeration data type defines all possible modes of time handling by CTesK test system.

```
typedef
enum TSTimeModel
{
  NotUseTSTime,
  LinearTSTime,
  DistributedTSTime
} TSTimeModel;
```

Description

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

- works without accounting time, if "deferred reactions" property of test engine is disabled,
- uses linear time model, if "deferred reactions" property is enabled.

Time handling mode can be changed in scenario initialization function.

Additional information

Header file: ts/timemark.h

Library: ts

See also

Time model, setTSTimeModel, getTSTimeModel

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.

Description

When dfsm or ndfsm 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.

Additional information

Header file: ts/timemark.h

Library: ts

See also

Time model, TSTimeModel, setTSTimeModel, dfsm test engine

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.

Additional information

Header file: ts/timemark.h

Library: ts

See also

Time model, TSTimeModel, getTSTimeModel

LinearTimeMark

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

```
typedef unsigned long LinearTimeMark;
```

Description

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.

Additional information

Header file: ts/timemark.h

Library: ts

See also

Time model, TimeMark

TimeFrameOfReferenceID

`TimeFrameOfReferenceID` type defines identifiers of time frames.

```
typedef int TimeFrameOfReferenceID;
```

Description

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.

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.

Additional information

Header file: ts/timemark.h

Library: ts

See also

Time model, systemTimeFrameOfReferenceID, getTimeFrameOfReferenceID, setSystemTimeFrameOfReferenceName

TimeMark

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

```
typedef struct TimeMark TimeMark;

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

Description

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.

Additional information

Header file: ts/timemark.h

Library: ts

See also

Time model, TimeFrameOfReferenceID, LinearTimeMark, minTimeMark, maxTimeMark, createTimeMark, createDistributedTimeMark

TimeInterval

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

```
typedef struct TimeInterval TimeInterval;

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

Description

`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.

Additional information

Header file: ts/timemark.h

Library: ts

See also

Time model, TimeMark, createTimeInterval, minTimeMark, maxTimeMark

systemTimeFrameOfReferenceID

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

```
extern const TimeFrameOfReferenceID systemTimeFrameOfReferenceID;
```

Description

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.

Additional information

Header file: ts/timemark.h

Library: ts

See also

Time model, TimeFrameOfReferenceID, setSystemTimeFrameOfReferenceName

minTimeMark

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

```
extern const TimeMark minTimeMark;
```

Description

`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.

Additional information

Header file: ts/timemark.h

Library: ts

See also

Time model, TimeMark, maxTimeMark

maxTimeMark

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

```
extern const TimeMark maxTimeMark;
```

Description

`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.

Additional information

Header file: ts/timemark.h

Library: ts

See also

Time model, TimeMark, maxTimeMark

getTimeFrameOfReferenceID

`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.

Description

`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.

Additional information

Header file: ts/timemark.h

Library: ts

See also

Time model, TSTimeModel, TimeFrameOfReferenceID, setSystemTimeFrameOfReferenceName, systemTimeFrameOfReferenceID

setSystemTimeFrameOfReferenceName

`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.

Description

`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.

Additional information

Header file: ts/timemark.h

Library: ts

See also

Time model, TSTimeModel, TimeFrameOfReferenceID, getTimeFrameOfReferenceID, systemTimeFrameOfReferenceID

createTimeMark

`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.

Description

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

Additional information

Header file: ts/timemark.h

Library: ts

See also

Time model, LinearTimeMark, TimeMark, systemTimeFrameOfReferenceID, createDistributedTimeMark

createDistributedTimeMark

`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

Library: ts

See also

Time model, TimeFrameOfReferenceID, LinearTimeMark, TimeMark, createTimeMark

createTimeInterval

`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.

Description

`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.

Additional information

Header file: ts/timemark.h

Library: ts

See also

Time model, TimeInterval, TimeMark, createTimeMark, createDistributedTimeMark, minTimeMark, maxTimeMark

GetCurrentTimeMarkFuncType

`GetCurrentTimeMarkFuncType` data type is used to set up the function, evaluating time mark of the current moment of time.

```
typedef TimeMark (*GetCurrentTimeMarkFuncType)(void);
```

Description

`GetCurrentTimeMarkFuncType` data type is used to set up the function, evaluating time mark of the current moment of time.

Additional information

Header file: ts/timemark.h

Library: ts

See also

Time model, TimeMark, setDefaultCurrentTimeMarkFunction

getCurrentTimeMark

`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.

Description

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.

Additional information

Header file: ts/timemark.h

Library: ts

See also

Time model, TimeMark, createTimeMark, createDistributedTimeMark, setDefaultCurrentTimeMarkFunction, systemTimeFrameOfReferenceID

setDefaultCurrentTimeMarkFunction

`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.

Description

`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 getCurrentTimeMark function 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.

Additional information

Header file: ts/timemark.h

Library: ts

See also

Time model, GetCurrentTimeMarkFuncType, getCurrentTimeMark

Standard test engines

CTesK 2.2 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.

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 (see Scenario functions).

The `dfsm` data type of test scenario is used in initialization of a test scenario, based on the `dfsm` test engine. This data type is a structure with the following fields:

init

finish

getState

actions

saveModelState

restoreModelState

isStationaryState

observeState

The only mandatory field is `actions`, defining a set of test actions.

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

setFinishMode

setDeferredReactionsMode

setWTime

setFindFirstSeriesOnly

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 d`fsm` test engine processes standard parameters and passes the rest of parameters to the scenario initialization function. In CTesK 2.2 the `dfsm` test engine supportes the following standard parameters:

-t <file-name>

-tc

-tt

-nt

-uerr

-uend

--trace-accidental

--find-first-series-only

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 (see Scenario functions).

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 with the following fields:

init

finish

getState

actions

saveModelState

restoreModelState

isStationaryState

observeState

The only mandatory field is `actions`, defining a set of test actions.

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

- setFinishMode
- setDeferredReactionsMode
- setWTime
- setFindFirstSeriesOnly

A list of parameters is passed to the test scenario on its invocation. The `ndfsm` test engine has the same standard parameters affected its behavior. Standard parameters must precede user parameters. The `ndfsm` test engine processes standard parameters and passes the rest of parameters to the scenario initialization function. In CTesK 2.2 the `ndfsm` test engine supportes the following standard parameters:

- -t <file-name>
- -tc
- -tt
- -nt
- -uerr
- -uend
- --trace-accidental
- --find-first-series-only

Types and parameters of test engines

This section describes the types and parameters using by dfsm and ndfsm standard test engines.

PtrInit

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

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

Description

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.

Additional information

Header file: ts/engine.h

Library: ts

See also

dfsm test engine, ndfsm test engine, init field

init

`init` field contains a pointer to test scenario initialization function.

```
PtrInit init;
```

Description

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.

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 is allowed to 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 behaves like an empty initialization function returning `true`.

Additional information

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

Library: ts

See also

dfsm test engine, ndfsm test engine, PtrInit, finish field

PtrFinish

`PtrFinish` field specifies test scenario finalization function type.

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

Description

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

Scenario finalization function is allowed to contain specification functions calls, freeing resources 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 finalization function call and all reactions received.

Additional information

Header file: ts/engine.h

Library: ts

See also

dfsm test engine, ndfsm test engine, finish field

finish

`finish` field contains a pointer to test scenario finalization function.

```
PtrFinish finish;
```

Description

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

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.

`finish` field can be initialized by a `NULL` pointer or can be not initialized at all. It behaves like an empty finalization function.

Additional information

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

Library: ts

See also

dfsm test engine, ndfsm test engine, PtrFinish, finish field

PtrGetState

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

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

Description

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

Additional information

Header file: ts/engine.h

Library: ts

See also

dfsm test engine, ndfsm test engine, getState field

getState

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

```
PtrGetState getState;
```

Description

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 presense 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 n`dfsm` test engines consider this as scenario with a single state.

Additional information

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

Library: ts

See also

dfsm test engine, ndfsm test engine, PtrGetState

actions

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

```
ScenarioFunctionID actions[];
```

Description

`actions` field contains an array of scenario functions, ended with a `NULL` pointer. See Scenario functions for more details.

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

Additional information

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

Library: ts

See also

dfsm test engine, ndfsm test engine,

PtrSaveModelState

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

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

Description

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

Additional information

Header file: ts/engine.h

Library: ts

See also

dfsm test engine, ndfsm test engine, saveModelState field

saveModelState

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

```
PtrSaveModelState saveModelState;
```

Description

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.

Additional information

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

Library: ts

See also

dfsm test engine, ndfsm test engine, PtrSaveModelState, restoreModelState field

PtrRestoreModelState

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

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

Description

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.

Additional information

Header file: ts/engine.h

Library: ts

See also

dfsm test engine, ndfsm test engine, restoreModelState field

restoreModelState

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

```
PtrRestoreModelState restoreModelState;
```

Description

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.

Additional information

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

Library: ts

See also

dfsm test engine, ndfsm test engine, PtrRestoreModelState, saveModelState field

PtrIsStationaryState

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

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

Description

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

Additional information

Header file: ts/engine.h

Library: ts

See also

dfsm test engine, ndfsm test engine, isStationaryStaty field

isStationaryState

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

```
PtrIsStationaryState isStationaryState;
```

Description

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.

Additional information

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

Library: ts

See also

dfsm test engine, ndfsm test engine, PtrIsStationaryState

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 );
```

Description

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

Additional information

Header file: ts/engine.h

Library: ts

See also

dfsm test engine, ndfsm test engine, observeState field

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;
```

Description

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.

Additional information

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

Library: ts

See also

dfsm test engine, ndfsm test engine, PtrObserveState

FinishMode

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

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

Description

`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.

Additional information

Header file: ts/engine.h

Library: ts

See also

dfsm_test_engine, ndfsm_test_engine, setFinishMode, getFinishMode, '-uerr' standard parameter, '-uend' standard parameter

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.

Description

`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.

Additional information

Header file: ts/engine.h

Library: ts

See also

dfsm test engine, ndfsm test engine, setFinishMode, getFinishMode, '-uerr' standard parameter, '-uend' standard parameter

getFinishMode

`getFinishMode` function returns test engine current finalization mode.

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

Return value

Current test engine finalization mode.

Description

`getFinishMode` function returns current test engine finalization mode.

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

Additional information

Header file: ts/engine.h

Library: ts

See also

dfsm_test_engine, ndfsm_test_engine, FinishMode, setFinishMode, '–uerr' standard parameter, '-uend' standard parameter

setDeferredReactionsMode

`setDeferredReactionsMode` function sets deferred reactions support mode of test engine.

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

Parameters

enable

If this parameter 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.

Description

`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.

By default deferred reactions support mode is:

- off, if some of the saveModelState, restoreModelState, or isStationaryState fields of test scenario are not defined or are initialized by a `NULL` pointer,
- on, if all of the saveModelState, restoreModelState, or isStationaryState fields of test scenario are initialized by a non-`NULL` pointer.

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.

Additional information

Header file: ts/engine.h

Library: ts

See also

dfsm test engine, ndfsm test engine, areDeferredReactionsEnabled, saveModelState field, restoreModelState field, isStationaryState field

areDeferredReactionsEnabled

`areDefferedReactionsEnabled` function returns current deferred reactions support mode of test engine.

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

Return value

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

Description

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

setDeferredReactionsMode function can be used to change deferred reactions support mode.

Additional information

Header file: ts/engine.h

Library: ts

See also

dfsm test engine, ndfsm test engine, setDeferredReactionsMode

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.

Description

`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 test scenario initialization function.

getWTime function can be used to get current value of waiting period.

Additional information

Header file: ts/engine.h

Library: ts

See also

dfsm test engine, ndfsm test engine, getWTime

getWTime

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

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

Return value

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

Description

`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.

Additional information

Header file: ts/engine.h

Library: ts

See also

dfsm test engine, ndfsm test engine, setWTime

setFindFirstSeriesOnly

`setFindFirstSeriesOnly` function sets `FindFirstSeriesOnly` property of a test system.

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

Parameters

`new_value`

Value of `FindFirstSeriesOnly` property of the test system.

Return value

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

Description

`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.

Additional information

Header file: ts/engine.h

Library: ts

See also

dfsm test engine, ndfsm test engine, isFindFirstSeriesOnly

isFindFirstSeriesOnly

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

```
bool isFindFirstSeriesOnly( void );
```

Return value

`isFindFirstSeriesOnly` function returns current value of `FindFirstSeriesOnly` property of the test system.

Description

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

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

Additional information

Header file: ts/engine.h

Library: ts

See also

dfsm test engine, ndfsm test engine, setFindFirstSeriesOnly

setFindFirstSeriesOnlyBound

`setFindFirstSeriesOnlyBound` function sets `FindFirstSeriesOnlyBound` property of a test system.

```
int setFindFirstSeriesOnly( 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.

Description

`setFindFirstSeriesOnlyBound` function sets `FindFirstSeriesOnlyBound` property of the test system.

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.

`setFindFirstSeriesOnlyBound(0)` call is equal to `setFindFirstSeriesOnly(false)` call. `setFindFirstSeriesOnlyBound(1)` call is equal to `setFindFirstSeriesOnly(true)` call.

By default the property is `0`.

`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.

Additional information

Header file: ts/engine.h

Library: ts

See also

dfsm test engine, ndfsm test engine, setFindFirstSeriesOnly, getFindFirstSeriesOnlyBound

getFindFirstSeriesOnlyBound

`getFindFirstSeriesOnlyBound` function returns current value of `FindFirstSeriesOnlyBound` property of a test system.

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

Return value

`getFindFirstSeriesOnlyBound` function returns current value of `FindFirstSeriesOnlyBound` property of the test system.

Description

`getFindFirstSeriesOnlyBound` function returns current value of `FindFirstSeriesOnlyBound` property of the test system.

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

Additional information

Header file: ts/engine.h

Library: ts

See also

dfsm test engine, ndfsm test engine, setFindFirstSeriesOnlyBound

‘–t’ standard parameter

‘`-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.

During test scenario operation set of devices for receiving test trace can be changed by functions for tracing control.

If scenario parameters contain several standard parameters for trace saving, trace will be sent to all specified devices.

If scenario parameters do not contain standard parameters for trace saving, it has the same effect if they contain ‘–tt’ standard parameter.

All devices added to the set by standard scenario parameters are automatically removed after the test scenario finish.

See also

dfsm test engine, ndfsm test engine, ‘–tc’ standard parameter, ‘–tt’ standard parameter, ‘–nt’ standard parameter, Tracing services, Tracing control, addTraceToFile, removeTraceToFile

‘–tc’ standard parameter

‘`-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.

During test scenario operation set of devices for receiving test trace can be changed by functions for tracing control.

If scenario parameters contain several standard parameters for trace saving, trace will be sent to all specified devices.

If scenario parameters do not contain standard parameters for trace saving, it has the same effect if they contain ‘–tt’ standard parameter.

All devices added to the set by standard scenario parameters are automatically removed after the test scenario finish.

See also

dfsm test engine, ndfsm test engine, ‘–t <file-name>’ standard parameter, ‘–tt’ standard parameter, ‘–nt’ standard parameter, Tracing services, Tracing control, addTraceToFile, removeTraceToFile

‘–tt’ standard parameter

‘`-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.

During test scenario operation set of devices for receiving test trace can be changed by functions for tracing control.

If scenario parameters contain several standard «`-tt`» parameters, only one of them will be taken into account.

If scenario parameters contain several standard parameters for trace saving, trace will be sent to all specified devices.

If scenario parameters do not contain standard parameters for trace saving, it has the same effect if they contain ‘`-tt`’ standard parameter.

All devices added to the set by standard scenario parameters are automatically removed after the test scenario finish.

See also

dfsm test engine, ndfsm test engine, ‘–t <file-name>’ standard parameter, ‘–tc’ standard parameter, ‘–nt’ standard parameter, Tracing services, Tracing control, addTraceToFile, removeTraceToFile

‘–nt’ standard parameter

‘`-nt`’ standard parameter disables tracing.

‘`-nt`’ standard parameter can not be used with ‘`-t`’, ‘`-tc`’ or ‘`-tt`’ standard parameters.

During test scenario operation set of devices for receiving test trace can be changed by functions for tracing control.

If scenario parameters do not contain standard parameters for trace saving, it has the same effect if they contain ‘–tt’ standard parameter.

See also

dfsm test engine, ndfsm test engine, ‘–t <file-name>’ standard parameter, ‘–tc’ standard parameter, ‘–tt’ standard parameter, Tracing services, Tracing control, addTraceToFile, removeTraceToFile

‘–uerr’ standard parameter

‘`-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.

See also

dfsm test engine, ndfsm test engine, ‘–uend’ standard parameter, setFinishMode

‘–uend’ standard parameter

‘`-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.

See also

dfsm test engine, ndfsm test engine, ‘–uerr’ standard parameter, setFinishMode

‘--trace-accidental’ standard parameter

‘`--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.

See also

dfsm test engine, ndfsm test engine, setTraceAccidental

‘--find-first-series-only’ standard parameter

‘`--find-first-series-only`’ standard parameter (‘`-ffso`’ for short) sets `FindFirstSeriesOnly` property of the test system. 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`.

See also

dfsm test engine, ndfsm test engine, setFindFirstSeriesOnly

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. More information about report generator can be found in *"CTesK 2.2: User's Guide"*.

For a user the tracer provides tracing control interface and message tracing interface.

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 2.2 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 2.2 the only available trace contents control function is:

- setTraceAccidental

Function to set trace character encoding is:

- setTraceEnconding

addTraceToConsole

`addTraceToConsole` function adds the console to the set of devices for trace saving.

```
void addTraceToConsole( void );
```

Description

`addTraceToConsole` 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.

Additional information

Header file: ts/c_tracer.h

Library: tracer

See also

Trace services, Trace control, removeTraceToConsole, addTraceToFile, removeTraceToFile

removeTraceToConsole

`removeTraceToConsole` function removes the console from the set of devices for trace saving.

```
void removeTraceToConsole( void );
```

Description

`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.

Additional information

Header file: ts/c_tracer.h

Library: tracer

See also

Trace services, Trace control, addTraceToConsole, addTraceToFile, removeTraceToFile

addTraceToFile

`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.

Description

`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.

Additional information

Header file: ts/c_tracer.h

Library: tracer

See also

Trace services, Trace control, removeTraceToFile, addTraceToConsole, removeTraceToConsole

removeTraceToFile

`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` otherwise.

Description

`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.

Additional information

Header file: ts/c_tracer.h

Library: tracer

See also

Trace services, Trace control, addTraceToFile, addTraceToConsole, removeTraceToConsole

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.

Description

`setTraceAccidental` function turns tracing of information about *uncertain transitions* on or off.

By default the tracer does not save information about uncertain transitions.

`setTraceAccidental` function can be invoked only when no one test scenario is running.

Additional information

Header file: ts/c_tracer.h

Library: tracer

See also

Trace services, Trace control

setTraceEncoding

`setTraceEncoding` function sets trace character encoding.

```
void setTraceEncoding( const char *encoding );
```

Parameters

`encoding`

The identifier of the trace character encoding.

Description

`setTraceEncoding` function sets trace character encoding. By default, character encoding of the trace is `UTF-8`.

A proper trace character encoding should be set in order to display localized functional branches names, subsystems names or user messages correctly in the reports on the tests executed.

Additional information

Header file: ts/c_tracer.h

Library: tracer

See also

Trace services, Trace control

Message tracing

CTesK 2.2 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. More information about test reports can be found in *"CTesK 2.2: User's Guide"*.

The following functions are used for tracing user messages:

traceUserInfo

traceFormattedUserInfo

traceUserInfo

`traceUserInfo` function saves the user message in the test trace.

```
void traceUserInfo( const char* info );
```

Parameters

info

Pointer to a character string, followed by a zero character, which contains a user message.

Description

`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. More information about test reports can be found in *"CTesK 2.2: User's Guide"*.

Additional information

Header file: ts/c_tracer.h

Library: tracer

See also

Trace services, Trace control

traceFormattedUserInfo

`traceUserInfo` function saves the formatted user message in the test trace.

```
void traceFormattedUserInfo( 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.

Description

`traceFormattedUserInfo` function formats and 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. More information about test reports can be found in *"CTesK 2.2: User's Guide"*.

Additional information

Header file: ts/c_tracer.h

Library: tracer

See also

Trace services, Trace control

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.

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;
```

Description

`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.

Additional information

Header file: ts/register.h

Library: ts

See also

Deferred reactions registration services, Interaction channels, WrongChannel, UniqueChannel, getChannelID, releaseChannelID

WrongChannel

`WrongChannel` constant indicates an incorrect interactions channel identifier.

```
extern const ChannelID WrongChannel;
```

Description

`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.

Additional information

Header file: ts/register.h

Library: ts

See also

Deferred reactions registration services, Interaction channels, ChannelID, UniqueChannel, getChannelID, releaseChannelID

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;
```

Description

`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.

Additional information

Header file: ts/register.h

Library: ts

See also

Deferred reactions registration services, Interaction channels, ChannelID, WrongChannel, getChannelID, releaseChannelID

getChannelID

`getChannelID` function returns a newly allocated interactions channel identifier.

```
ChannelID getChannelID( void );
```

Return value

`getChannelID` function returns a newly allocated interactions channel identifier if it can, or WrongChannel otherwise.

Description

`getChannelID` function returns a newly allocated interactions channel identifier. If it cannot allocate an identifier, it returns WrongChannel constant.

No longer necessary channel identifier can be freed by releaseChannelID function.

Additional information

Header file: ts/register.h

Library: ts

See also

Deferred reactions registration services, Interaction channels, ChannelID, WrongChannel, UniqueChannel, releaseChannelID

releaseChannelID

`releaseChannelID` function frees the specified interactions channel identifier.

```
void releaseChannelID( ChannelID chid );
```

Parameters

chid

Channel identifier to be freed.

The parameter must be a channel identifier, returned previously by getChannelID function.

Description

`releaseChannelID` function frees the specified interactions channel identifier, returned previously by getChannelID function.

Additional information

Header file: ts/register.h

Library: ts

See also

Deferred reactions registration services, Interaction channels, ChannelID, WrongChannel, UniqueChannel, getChannelID

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 setStimulusChannel( 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.

Description

`setStimulusChannel` function set the value of `StimulusChannel` property. `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.

Additional information

Header file: ts/register.h

Library: ts

See also

Deferred reactions registration services, Interactions registrar, Interaction channels, ChannelID, UniqueChannel, getStimulusChannel

getStimulusChannel

`getStimulusChannel` function returns the value of `StimulusChannel` property.

```
ChannelID getStimulusChannel( void );
```

Return value

The function returns the value of `StimulusChannel` property.

Description

`getStimulusChannel` function returns the value of `StimulusChannel` property. `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.

Additional information

Header file: ts/register.h

Library: ts

See also

Deferred reactions registration services, Interactions registrar, Interaction channels, ChannelID, UniqueChannel, setStimulusChannel

registerReaction

`registerReaction` function is intended for registration of reactions, received from the target system. Reaction is an interaction initiated by the target system.

```
void registerReaction(
          ChannelID         chid,
          const char*       name,
          SpecificationID   reactionID,
          Object*           data
                      );
```

Parameters

chid

Identifier of an interactions channel, in which the reaction has occurred.

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.

Description

`registerReaction` function is intended for registration of reactions, received from the target system. Reaction is an interaction initiated by the target system.

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.

Additional information

Header file: ts/register.h

Library: ts

See also

Deferred reactions registration services, Interactions registrar, Interaction channels, ChannelID, registerReactionWithTimeMark, registerReactionWithTimeInterval, registerWrongReaction

registerReactionWithTimeMark

`registerReactionWithTimeMark` function is intended for registration of reactions, received from the target system, specifying time mark of occurrence moment. Reaction is an interaction initiated by the target system.

```
void registerReactionWithTimeMark(
        ChannelID         chid,
        const char*       name,
        SpecificationID   reactionID,
        Object*           data,
        TimeMark          mark
                                );
```

Parameters

chid

Identifier of an interactions channel, in which the reaction has occurred.

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.

Description

`registerReactionWithTimeMark` function is intended for registration of reactions, received from the target system, specifying time mark of occurrence moment. Reaction is an interaction initiated by the target system.

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.

Additional information

Header file: ts/register.h

Library: ts

See also

Deferred reactions registration services, Interactions registrar, Interaction channels, ChannelID, TimeMark, registerReaction, registerReactionWithTimeInterval, registerWrongReaction

registerReactionWithTimeInterval

`registerReactionWithTimeInterval` function is intended for registration of reactions, received from the target system, specifying time interval of its occurrence. Reaction is an interaction initiated by the target system.

```
void registerReactionWithTimeInterval(
        ChannelID         chid,
        const char*       name,
        SpecificationID   reactionID,
        Object*           data,
        TimeInterval      interval
                                  );
```

Parameters

chid

Identifier of an interactions channel, in which the reaction has occurred.

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.

Description

`registerReactionWithTimeInterval` function is intended for registration of reactions, received from the target system, specifying time interval of its occurrence. Reaction is an interaction initiated by the target system.

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.

Additional information

Header file: ts/register.h

Library: ts

See also

Deferred reactions registration services, Interactions registrar, Interaction channels, ChannelID, TimeInterval, registerReaction, registerReactionWithTimeMark, registerWrongReaction

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.

Description

`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.

Additional information

Header file: ts/register.h

Library: ts

See also

Deferred reactions registration services, Interactions registrar, registerReaction, registerReactionWithTimeMark, registerReactionWithTimeInterval, registerStimulusWithTimeInterval

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 registerStimulusWithTimeInterval(
        ChannelID         chid,
        const char*       name,
        SpecificationID   stimulusID,
        TimeInterval      interval,
        ...
                                  );
```

Parameters

chid

Identifier of an interactions channel, in which the stimulus has given.

The parameter must be a correct channel identifier.

name

Name of the stimulus. Used only for tracing.

The parameter can be a `NULL` pointer. If so, name of the interaction is considered to be equal to the specification function name *stimulusID*.

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:

- List of specification function parameters values before its invocation.
- List of specification function parameters values after its invocation.
- Value returned by the specification function (if data type of result value is not `void`).

Description

`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.

Additional information

Header file: ts/register.h

Library: ts

See also

Deferred reactions registration services, Interactions registrar, Interaction channels, ChannelID, TimeInterval, registerWrongReaction

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

ReactionCatcherFuncType

`ReactionCatcherFuncType` data type is used for registration of catcher functions in the test system.

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

Description

`ReactionCatcherFuncType` data type is used as data type of catcher functions in the test system.

Additional information

Header file: ts/timemark.h

Library: ts

See also

Deferred reactions registration services, Catcher functions registering service, registerReactionCatcher, unregisterReactionCatcher, unregisterReactionCatchers

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 registered.

The parameter can be a `NULL` pointer.

Description

`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.

Additional information

Header file: ts/register.h

Library: ts

See also

Deferred reactions registration services, Catcher functions registering service, ReactionCatcherFuncType, unregisterReactionCatcher, unregisterReactionCatchers

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 that was provided during registration.

Return value

The function returns `false` if the specified function with the specified auxiliary data was not registered before, and `true` otherwise.

Description

`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.

Additional information

Header file: ts/register.h

Library: ts

See also

Deferred reactions registration services, Catcher functions registering service, ReactionCatcherFuncType, registerReactionCatcher, unregisterReactionCatchers

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.

Description

`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.

Additional information

Header file: ts/register.h

Library: ts

See also

Deferred reactions registration services, Catcher functions registering service, ReactionCatcherFuncType, registerReactionCatcher, unregisterReactionCatcher

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*, строки char*), for complex numbers, for data type with a single value, and for container data types (list, set, map).

Predefined data types are convenient in *specifications* (for example, for implementation state modeling) as they provide ready for use, universal, guaranteed faultless functionality.

Standard functions

Standard functions can be used for any specification references. Result of function execution is dependent 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 `type_` prefix:

```
const Type type_specification_data_type_name;
```

Specification reference creation function

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

The function takes a pointers to *specification data type descriptor* as its first parameter. Other parameters are data type initialization parameters and vary for different data types. Initialization parameters for all predefined data types are described below, and the parameters for used-defined data types are specified in the data type definition.

The function returns a pointer to the created object.

```
Integer* ref = create(&type_Integer, 28); // ref → 28
```

In the above example a reference to library `Integer` data type (specification analogue of `int` data type) is created and initialized.

Specification reference data type function

`const Type *type(Object* ref)`

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
```

Specification reference copying function

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

The function copies the contents by `src` reference to contents by `dst` reference. 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 the appropriate 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.

`Object* clone(Object* ref)`

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); // ref1 → 28
String*  ref2 = clone(ref1);               // ref2 → 28
```

Values, referenced by `ref1` and `ref2` pointers, become equal after invocation of `clone()` function.

Specification reference comparing functions

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

The function returns zero if the values, referenced by the given references, are equal. Otherwise 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 the both references are `NULL`, the function returns zero.

```
if (!compare(ref1,ref2)) {/* values are equal */
  ...
}
else {/* values are not equal */
  ...
}
```

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

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 */
  ...
}
```

Specification reference stringifying function

`String* toString(Object* ref)`

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));
```

```
*ref == '28'
```

Predefined specification data types

Char

```
#include <atl/char.h>
```

`Char` data type is a specification analogue of `char` C data type.

`Char* create_Char( char d )`

Creates a specification reference to the value of `Char` data type.

This function is defines along with the standard `create()` function:

```
Char* ch1 = create(&type_Char, 'a');
Char* ch2 = create_Char('a');
```

The above ways of creating a specification reference are equivalent.

`char value_Char( Char* d )`

Returns the `char` value, contained in this specification data type.

This value can also be accessed by dereferencing of the specification reference:

```
Char* ch = create_Char('a');
char  val = *ch;
```

`value_Char()` function provides run time data type checking and it can take a pointer to `Object` without previous type cast:

```
List* l;
char val;
...
val = value_Char(get_List(l,0)); // Object* get_List(List*,int)
```

If the reference can probably have a wrong data type, a direct check by `type()` standard function can be useful:

```
Object* o = get_List(l,0));
if (type(o) == &type_Char) {
  val = value_Char(o);
}
```

Integer и UInteger

```
#include <atl/integer.h>
```

`Integer` and `UInteger` data types are specification analogues of `int` and `unsigned int` C data types.

```
Integer*                    create_Integer(          int          i          )
UInteger* create_UInteger( unsigned int i )
```

Creates a specification reference to the value of `Integer` or `UInteger` data types.

This function is defines along with the standard `create()` function:

```
Integer* i1 = create(&type_Integer, -28);
Integer* i2 = create_Integer(-28);
```

```
UInteger* i1 = create(&type_UInteger, 28);
UInteger* i2 = create_UInteger(28);
```

The above ways of creating a specification reference are equivalent.

```
int          value_Integer( Integer* i )
unsigned int value_UInteger( UInteger* i )
```

Returns the `int` or `unsigned int` value, contained in this `Integer` or `UInteger` specification data type.

These values can also be accessed by dereferencing of the specification reference:

```
Integer* i = create_Integer(-28);
int val = *i;
```

`value_Integer()` or `value_UInteger()` functions provide run time data type checking and it can take a pointer to `Object` without previous type cast:

```
List* l;
int val;
...
val = value_Integer(get_List(l,0)); // Object* get_List(List*,int)
```

If the reference can probably have a wrong data type, a direct check by `type()` standard function can be useful:

```
Object* o = get_List(l,0));
if (type(o) == &type_Integer) {
  val = value_Integer(o);
}
```

Short и UShort

```
#include <atl/short.h>
```

`Short` and `UShort` data types are specification analogues of `short int` and `unsigned short int` C data types.

```
Short*  create_Short  ( short i )
UShort* create_UShort ( unsigned short i )
```

Creates a specification reference to the value of `Short` or `UShort` data types.

This function is defines along with the standard `create()` function:

```
Short* i1 = create(&type_Short, -28);
Short* i2 = create_Short(-28);

UShort* i1 = create(&type_UShort, 28);
UShort* i2 = create_UShort(28);
```

The above ways of creating a specification reference are equivalent.

```
short          value_Short ( Short* i )
unsigned short value_UShort ( UShort* i )
```

Returns the `short` or `unsigned short` value, contained in this `Short` or `UShort` specification data type.

These values can also be accessed by dereferencing of the specification reference:

```
Short* i = create_Short(-28);
short val = *i;
```

`value_Short()` or `value_UShort()` functions provide run time data type checking and it can take a pointer to `Object` without previous type cast:

```
List* l;
short val;
...
val = value_Short(get_List(l,0)); // Object* get_List(List*,int)
```

If the reference can probably have a wrong data type, a direct check by `type()` standard function can be useful:

```
Object* o = get_List(l,0));
if (type(o) == &type_Short) {
  val = value_Short(o);
}
```

Long и ULong

```
#include <atl/long.h>
```

`Long` and `ULong` data types are specification analogues of `long int` and `unsigned long int` C data types.

```
Long*                        create_Long(          long          i          )
ULong* create_ULong ( unsigned long i )
```

Creates a specification reference to the value of `Long` or `ULong` data types.

This function is defines along with the standard `create()` function:

```
Long* i1 = create(&type_Long, -28);
Long* i2 = create_Long(-28);

ULong* i1 = create(&type_ULong, 28);
ULong* i2 = create_ULong(28);
```

The above ways of creating a specification reference are equivalent.

```
long          value_Long ( Long* i )
unsigned long value_ULong ( ULong* i )
```

Returns the `long` or `unsigned long` value, contained in this `Long` or `ULong` specification data type.

These values can also be accessed by dereferencing of the specification reference:

```
Long* i = create_Long(-28);
long *p = (long*)i;
```

`value_Long()` or `value_ULong()` functions provide run time data type checking and it can take a pointer to `Object` without previous type cast:

```
List* l;
long val;
...
val = value_Long(get_List(l,0)); // Object* get_List(List*,int)
```

If the reference can probably have a wrong data type, a direct check by `type()` standard function can be useful:

```
Object* o = get_List(l,0));
if (type(o) == &type_Long) {
  val = value_Long(o);
}
```

Float

```
#include <atl/float.h>
```

`Float` data type is a specification analogue of `float` C data type.

`Float* create_Float( float d )`

Creates a specification reference to the value of `Float` data type.

This function is defines along with the standard `create()` function:

```
Float* f1 = create(&type_Float, 3.14);
Float* f2 = create_Float(3.14);
```

The above ways of creating a specification reference are equivalent.

`float value_Float ( Float* d )`

Returns the `float` value, contained in this specification data type.

This value can also be accessed by dereferencing of the specification reference:

```
Float* f = create_Float(3.14);
float val = *f;
```

`value_Float()` function provides run time data type checking and it can take a pointer to `Object` without previous type cast:

```
List* l;
float val;
...
val = value_Float(get_List(l,0)); // Object* get_List(List*,int)
```

If the reference can probably have a wrong data type, a direct check by `type()` standard function can be useful:

```
Object* o = get_List(l,0));
if (type(o) == &type_Float) {
  val = value_Float(o);
}
```

Double

```
#include <atl/double.h>
```

`Double` data type is a specification analogue of `double` C data type.

`Double* create_Double ( double d )`

Creates a specification reference to the value of `Double` data type.

This function is defines along with the standard `create()` function:

```
Double* d1 = create(&type_Double, 3.14);
Double* d2 = create_Double(3.14);
```

The above ways of creating a specification reference are equivalent.

`double value_Double ( Double* d )`

Returns the `double` value, contained in this specification data type.

This value can also be accessed by dereferencing of the specification reference:

```
Double* d = create_Double(3.14);
double *p = (double*)d;
```

`value_Double()` function provides run time data type checking and it can take a pointer to `Object` without previous type cast:

```
List* l;
double val;
...
val = value_Double(get_List(l,0)); // Object* get_List(List*,int)
```

If the reference can probably have a wrong data type, a direct check by `type()` standard function can be useful:

```
Object* o = get_List(l,0));
if (type(o) == &type_Double) {
  val = value_Double(o);
}
```

VoidAst

```
#include <atl/void_ast.h>
```

`VoidAst` data type is a specification analogue of `void*` C data type.

`VoidAst* create_VoidAst ( void *d )`

Creates a specification reference to the value of `VoidAst` data type.

This function is defines along with the standard `create()` function:

```
VoidAst* v1 = create(&type_VoidAst, NULL);
VoidAst* v2 = create_VoidAst(NULL);
```

The above ways of creating a specification reference are equivalent.

`void *value_VoidAst ( VoidAst* d )`

Returns the `void*` value, contained in this specification data type.

This value can also be accessed by dereferencing of the specification reference:

```
VoidAst* v = create_VoidAst(NULL);
void *val = *v;
```

`value_VoidAst()` function provides run time data type checking and it can take a pointer to `Object` without previous type cast:

```
List* l;
void *val;
...
val = value_VoidAst(get_List(l,0)); // Object* get_List(List*,int)
```

If the reference can probably have a wrong data type, a direct check by `type()` standard function can be useful:

```
Object* o = get_List(l,0));
if (type(o) == &type_VoidAst) {
  val = value_VoidAst(o);
}
```

Unit

```
#include <atl/unit.h>
```

`Unit` data type is a specification data type with the only value.

Two non-`NULL` specification references of `Unit` data type are always equal.

`Unit* create_Unit ()`

Creates a specification reference to the value of `Unit` data type.

This function is defines along with the standard `create()` function:

```
Unit* u1 = create(&type_Unit);
Unit* u2 = create_Unit();
```

The above ways of creating a specification reference are equivalent.

Complex

```
#include <atl/complex.h>
```

`Complex` data type is intended for representation of complex numbers.

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 look like `( re + im*i )`.

Base type of `Complex` data type is the following structure:

```
struct {
  double re;
  double im;
};
```

There are no special functions to access real and imaginary parts of the complex number, so dereferencing should be used:

```
Complex* c = create_Complex(1.4, -0.6);
double re = c->re;
double im = c->im;
```

`Complex* create_Complex ( double re, double im )`

Creates a specification reference to the value of `Complex` data type with real part equals to `re` and imaginary part equals to `im`.

This function is defines along with the standard `create()` function:

```
Complex* c1 = create(&type_Complex, 1.4, -0.6);
Complex* c2 = create_Complex(1.4, -0.6);
```

The above ways of creating a specification reference are equivalent.

String

```
#include <atl/string.h>
```

`String` data type is intended for representation of character strings.

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 did. 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 string provides itself a lot of convenient functions. All that functions can accept only non-`NULL String` references.

`String* create_String ( const char *cstr )`

Creates a specification reference to the value of `String` data type and initialized it by `cstr` C string.

This function is defines along with the standard `create()` function:

```
String* s1 = create(&type_String, "a string");
String* s2 = create_String("a string");
```

The above ways of creating a specification reference are equivalent.

`char charAt_String( String* self, int index )`

Returns a character in the given `index` position.

Number of the position must be in the range from `0` to `length_String(self)-1`.

```
String* s = create_String("abracadabra");
printf("%c\n", charAt_String(s,5));
```

```
c
```

`String *concat_String( String* self, String* str )`

Returns concatenation of `self` and `str` string.

```
String* s1 = create_String("abra");
String* s2 = create_String("cadabra");
String* s  = concat_String(s1,s2);
printf("%s\n", toCharArray_String(s));
```

```
abracadabra
```

`bool endsWith_String( String *self, String *suffix )`

Checks whether `self` string ends with `suffix` string.

If `suffix` string is empty, returns `true`. If length of `suffix` string is greater than length of `self` string, 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
```

`int indexOfChar_String( String* self, int ch )`

Returns the position of first `ch` character in the string. If the character is not found, returns –1.

For the character with zero code always returns –1.

```
String* s = create_String("abracadabra");
printf("1) %d\n2) %d\n",
  indexOfChar_String(s,'b'),
  indexOfChar_String(s,'z')
);
```

```
1) 1
2) -1
```

`int indexOfCharFrom_String( String* self, int ch, int fromIndex )`

Returns the position of `ch` character in the string, starting from `fromIndex` position. If the character is not found, returns –1.

If `fromIndex` position is greater than length of `self` string, i.e. `fromIndex > length_String(self)`, returns –1. If `fromIndex < 0`, the position is considered to be 0. For the character with zero code always returns –1.

```
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
```

`int indexOfString_String( String* self, String* str )`

Returns the position of `str` substring within `self` string. If the substring is not found, returns –1.

If the substring is empty, it considered to belong to any string (including empty string) from zero 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
```

int indexOfStringFrom_String(String* self, String* str, int fromIndex)

Returns the position of `str` substring within `self` string, starting from `fromIndex` position. If the substring is not found, returns –1.

If `fromIndex` position is greater that length of `self` string, i.e. `fromIndex > length_String(self)`, returns –1. If `fromIndex < 0`, `fromIndex` position is considered to be 0. If the substring is empty, it considered to belong to any string (including empty string) 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
```

int lastIndexOfChar_String(String* self, int ch)

Returns the position of first `ch` character when searching the string backwards. If the character is not found, returns –1.

For the character with zero code always returns –1.

```
String* s = create_String("abracadabra");
printf("1) %d\n2) %d\n",
  lastIndexOfChar_String(s,'b'),
  lastIndexOfChar_String(s,'z')
);
```

```
1) 8
2) -1
```

int lastIndexOfCharFrom_String(String* self, int ch, int fromIndex)

Returns the position of first `ch` character when searching the string backwards, starting from `fromIndex` position. If the character is not found, returns –1.

If `fromIndex < 0`, returns –1. If `fromIndex` is greater that length of `self` string, i.e. `fromIndex > length_String(self)`, the position is considered to be `length_String(self)`. For the character with zero code always returns –1.

```
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
```

int lastIndexOfString_String(String* self, String* str)

Returns the position of `str` substring when searching the string backwards. If the substring is not found, returns –1.

If the substring is empty, it considered to belong to any string (including empty string) from `length_String(self)` position.

```
String* s  = create_String("abracad"abra");
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
```

`int lastIndexOfStringFrom_String( String* self, String* str, int fromIndex )`

Returns the position of `str` substring when searching the string backwards, starting from `fromIndex` position. If the substring is not found, returns –1.

If `fromIndex < 0`, returns –1. If `fromIndex` is greater than length of `self` string, i.e. `fromIndex > length_String(self)`, the position is considered to be `length_String(self)`. If the substring is empty, it considered to belong to any string (including empty string) from `fromIndex` 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
```

`int length_String( String* self )`

Returns length of the string.

```
String* s = create_String("abracadabra");
printf("%d\n", length_String(s));
```

```
11
```

`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)`

Checks whether the substring of `self` string (`len` length, from `toffset` position) matches the substring of `other` string (`len` length, `ooffset` position). `regionMatchesCase_String` function takes letters case into account, and `regionMatches_String` function has an additional parameter `ignoreCase` (case is not considered if `true`, and considered if `false`).

Length of substrings must be non-negative (`len ≥ 0`). If substrings, defined by the position and length, are out of the string bounds, the function returns `false`.

```
String* 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
```

String* replace_String(String* self, char oldChar, char newChar)

Returns a string constructed from self string by changing all occurrences of `oldChar` character to `newChar` character.

The characters must have a non-zero code.

```
String* s = create_String("abracadabra");
String* res = replace_String(s,'a','_');
printf("%s\n",toCharArray(res));
```

```
_br_c_d_br_
```

bool startsWith_String(String *self, String *prefix)

Checks whether `self` string starts with `prefix` string.

If `prefix` string is empty, returns `true`. If length of `prefix` string is greater than length of `self` string, returns `false`.

```
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
```

bool startsWithOffset_String(String *self, String *prefix, int toffset)

Checks whether `self` string starts with `prefix` string from `offset` position.

If the position is negative or is greater than length of `self` string, returns `false`. If `prefix` string is empty, returns `true`. If length of `prefix` string is greater than length of `self` string, returns `false`.

```
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
```

`String* substringFrom_String( String* self, int beginIndex )`

Returns a substring of `self` string, from `beginIndex` position till the end of the string.

`beginIndex` position must be within the string bounds: `0 ≤ beginIndex ≤ length_String(self)`.

```
String* s  = create_String("abracadabra");
String* res = substringFrom_String(s,4);
printf("%s\n",toCharArray_String(res));
```

```
cadabra
```

`String* substring_String( String* self, int beginIndex, int endIndex )`

Returns a substring of `self` string, from `beginIndex` position to `endIndex-1` position (including the bounds).

`beginIndex` position must be non-negative and must be not greater than `endIndex`. `endIndex` position must be not greater than length of the string: `0 ≤ beginIndex ≤ endIndex ≤ length_String(self)`. If starting and ending positions are equal, returns an empty string.

```
String* s  = create_String("abracadabra");
String* res = substring_String(s,4,7);
printf("%s\n",toCharArray_String(res));
```

```
cad
```

`const char* toCharArray_String( String* self )`

Returns a C string, corresponding to the given specification string.

The function returns a pointer to character array within the specification object, therefore `free()` function cannot be used for the pointer and the pointer cannot be used after destroying the object.

```
String* s = create_String("abracadabra");
printf("%s\n",toCharArray_String(s));
```

```
abracadabra
```

`String* toLowerCase_String( String* self )`

Returns a string, constructed from `self` string by converting it to lower case.

```
String* s   = create_String("aBrAcAdAbRa");
String* res = toLowerCase_String(s);
printf("%s\n",toCharArray_String(res));
```

```
abracadabra
```

`String* toUpperCase_String( String* self )`

Returns a string, constructed from `self` string by converting it to upper case.

```
String* s   = create_String("aBrAcAdAbRa");
String* res = toLowerCase_String(s);
printf("%s\n",toCharArray_String(res));
```

```
ABRACADABRA
```

String* trim_String(String* self)

Returns a string, constructed from self string by trimming space characters from beginning and ending of the string.

Space characters are spaces, tabs, and new line characters.

```
String* s   = create_String(" \tabracadabra  \n");
String* res = trim_String(s);
printf("'%s'\n",toCharArray_String(res));
```

```
'abracadabra'
```

String* format_String(const char *format, ...)

Returns a specification string, corresponding to output of `printf()` function, invoked with the same parameters:

```
char s[12];
String* str;
sprintf(s,"abra%s","cadabra");
str = create_String(s);
```

```
String* str = format_String("abra%s","cadabra");
```

The above ways of creating a string are equivalent.

String* valueOfBool_String(bool b)

Returns string representation of `bool` value.

```
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
```

String* valueOfChar_String(char c)

Returns string representation of `char` value.

```
String* s = valueOfChar_String('a');
printf("%s\n",toCharArray_String(s));
```

```
a
```

String* valueOfShort_String(short i)

Returns string representation of `short` value.

```
String* s = valueOfShort_String (-28);
printf("%s\n",toCharArray_String(s));
```

```
-28
```

String* valueOfUShort_String(unsigned short i)

Returns string representation of `unsigned short` value.

```
String* s = valueOfUShort_String(47);
printf("%s\n",toCharArray_String(s));
```

```
47
```

String* valueOfInt_String(int i)

Returns string representation of `int` value.

```
String* s = valueOfInt_String(-28);
printf("%s\n",toCharArray_String(s));
```

```
-28
```

String* valueOfUInt_String(unsigned int i)

Returns string representation of `unsigned int` value.

```
String* s = valueOfUInt_String(47);
printf("%s\n",toCharArray_String(s));
```

```
47
```

String* valueOfLong_String(long i)

Returns string representation of `long` value.

```
String* s = valueOfLong_String(-28);
printf("%s\n",toCharArray_String(s));
```

```
-28
```

String* valueOfULong_String(unsigned long i)

Returns string representation of `unsigned long` value.

```
String* s = valueOfULong_String(47);
printf("%s\n",toCharArray_String(s));
```

```
47
```

String* valueOfFloat_String(float f)

Returns string representation of `float` value.

```
String* s = valueOfFloat_String(3.14);
printf("%s\n",toCharArray_String(s));
```

```
3.140000
```

String* valueOfDouble_String(double d)

Returns string representation of `double` value.

```
String* s = valueOfDouble_String(3.14);
printf("%s\n",toCharArray_String(s));
```

```
3.140000
```

String* valueOfPtr_String(void *p)

Returns string representation of `void*` value.

```
int i;
String* s = valueOfPtr_String((void*)&i);
printf("%s\n",toCharArray_String(s));
```

```
0012FF6C
```

String* valueOfObject_String(Object* ref)

Returns string representation of the specification data type value.

Result is the same as for `toString()` standard function:

```
Object* ref;
...
String* s = valueOfObject_String(ref);
String* s = toString(ref);
```

String* valueOfBytes_String(const char* p, int l)

Returns string hexadecimal representation of `p` byte array of `l` length.

```
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]
```

List

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

`List` data type is a container data type, implementing an ordered list of items.

Any specification references can be elements of a list. Type of elements can be constrained when creating a list. 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 list 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 a list are numbered from 0.

`List* create_List( const Type *elem_type )`

Creates a list and returns a specification reference of `List` data type. If `elem_type` parameter is `NULL`, type of elements is not restricted. Otherwise the parameter must a pointer to descriptor constant of data type of list's elements:

```
List* l1 = create_List(NULL);          // any elements
List* l2 = create_List(&type_Integer); // only elements of Integer type
```

This function is defines along with the standard `create()` function:

```
List* l1 = create(&type_List, NULL);
List* l2 = create_List(NULL);
```

The above ways of creating a specification reference are equivalent.

`Type *elemType_List(List* self)`

Returns a pointer to descriptor constant of specification data type, that constrains data type of the list's elements.

If the list is not typified, returns `NULL`.

```
List* l = create_List(&type_Integer);
Type *t = elemType_List(l);             // &type_Integer
```

`void add_List( List* self, int index, Object* ref )`

Inserts `ref` element into the list at `index` position.

If the list is typified, data type of `ref` element must coincide with data type of list's elements. Number of position must be in the range from 0 to length of the list (including the bounds), i.e. `0 ≤ index ≤ size_List(self)`. If number of position is equal to 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));      // 28
add_List(l,0,create_Integer(47));      // 47 28
add_List(l,1,create_Integer(63));      // 47 63 28
```

`void append_List( List* self, Object* ref )`

Appends `ref` element to the end of the list.

If the list is typified, data type of `ref` element must coincide with data type of list's elements.

```
List* l = create_List(&type_Integer);
append_List(l,create_Integer(28));      // 28
append_List(l,create_Integer(47));      // 28 47
append_List(l,create_Integer(63));      // 28 47 63
```

`void clear_List( List* self )`

Removes all elements from the list.

```
List* l = create_List(&type_Integer);
append_List(l,create_Integer(28));      // 28
append_List(l,create_Integer(47));      // 28 47
clear_List(l);                          //
```

The same result can be achieved by re-creating the list, but `clear_List()` function is more effective:

```
List* l;
...
l = create_List(elemType_List(l));
```

`bool contains_List( List* self, Object* ref )`

Checks whether the list contains an element, equals to the given.

If the list is typified, data type of `ref` element must coincide with data type of list's elements.

```
List* l = create_List(&type_Integer);
append_List(l,create_Integer(28));          // 28
append_List(l,create_Integer(47));          // 28 47
if (contains_List(l,create_Integer(28))) ... // истинно
```

`Object* get_List( List* self, int index )`

Returns a specification reference to the element at `index` position.

Number of position must be in the rage from 0 to length of the list – 1, i.e. `0 ≤ index < size_List(self)`.

```
List* l = create_List(&type_Integer);
Object* o;
append_List(l,create_Integer(28));      // 28
append_List(l,create_Integer(47));      // 28 47
append_List(l,create_Integer(63));      // 28 47 63
o = get_List(l,1);                      // 47
```

`int indexOf_List( List* self, Object* ref )`

Returns number of position of first element in the list, equals to `ref`.

If the list is typified, data type of `ref` element must coincide with data type of list's elements. If the list does not contain the element, returns –1.

```
List* l = create_List(&type_Integer);
append_List(l,create_Integer(28));          // 28
append_List(l,create_Integer(47));          // 28 47
append_List(l,create_Integer(28));          // 28 47 28
int pos = indexOf_List(l,create_Integer(28)); // 0
```

`bool isEmpty_List( List* self )`

Checks whether the list is empty.

```
bool empty;
List* l = create_List(&type_Integer);
empty = isEmpty_List(l);                    // true
append_List(l,create_Integer(28));          // 28
empty = isEmpty_List(l);                    // false
```

`int lastIndexOf_List( List* self, Object* ref )`

Returns number of position of the last element in the list, equals to `ref`.

If the list is typified, data type of `ref` element must coincide with data type of list's elements. If the list does not contain the element, returns –1.

```
List* l = create_List(&type_Integer);
append_List(l,create_Integer(28));              // 28
append_List(l,create_Integer(47));              // 28 47
append_List(l,create_Integer(28));              // 28 47 28
int pos = lastIndexOf_List(l,create_Integer(28)); // 2
```

`void remove_List( List* self, int index )`

Removes the element at `index` position from the list.

Number of position must be in the range from 0 to length of the list, i.e. `0 ≤ index < size_List(self)`.

```
List* l = create_List(&type_Integer);
append_List(l,create_Integer(28));      // 28
append_List(l,create_Integer(47));      // 28 47
append_List(l,create_Integer(63));      // 28 47 63
remove_List(l,1);                       // 28 63
```

`void set_List( List* self, int index, Object* ref )`

Replaces the element at `index` position in the list by `ref` element.

If the list is typified, data type of `ref` element must coincide with data type of list's elements. Number of position must be in the range from 0 to length of the list (including the bounds), i.e. `0 ≤ index ≤ size_List(self)`. If number of position is equal to length of the list, the element is appended to the end of this list.

```
List* l = create_List(&type_Integer);
append_List(l,create_Integer(28));      // 28
append_List(l,create_Integer(47));      // 28 47
set_List(l,1,create_Integer(63));       // 28 63
```

`int size_List( List* self )`

Returns length of the list.

```
int size;
List* l = create_List(&type_Integer);
size = size_List(l);                    // 0
```

```
append_List(l,create_Integer(28));      // 28
size = size_List(l);                    // 1
```

List* subList_List(List* self, int fromIndex, int toIndex)

Returns sublist of the given list, containing elements from `fromIndex` position to `toIndex-1` position.

`fromIndex` position must be not negative and must be not greater than `toIndex`. `toIndex` position must be not greater than length of the list: `0 ≤ fromIndex ≤ toIndex ≤ size_List(self)`. If `fromIndex` equals to `toIndex`, returns an empty list.

```
List* l = create_List(&type_Integer);
List* l2;
append_List(l,create_Integer(28));      // 28
append_List(l,create_Integer(47));      // 28 47
append_List(l,create_Integer(63));      // 28 47 63
l2 = subList_List(l,1,3);               // 47 63
```

void addAll_List(List* self, int index, List* other)

Adds to `seft` list all elements of `other` list, inserting them from `index` position.

If `self` list is typified, types of all elements of `other` list must coincide with type of elements of `self` list (`other` list itself is not required to be typified). Number of position must be in the range from 0 to length of self list (including the bounds), i.e. `0 ≤ index ≤ size_List(self)`. If number of position is equal to 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));      // 28
append_List(l1,create_Integer(47));      // 28 47
append_List(l2,create_Integer(63));      // 63
append_List(l2,create_Integer(85));      // 63 85
addAll_List(l1,1,l2);                    // 28 63 85 47
```

void appendAll_List(List* self, List* other)

Appends all elements of `other` list to the end of `self` list.

If `self` list is typified, types of all elements of `other` list must coincide with type of elements of `self` list (`other` list itself is not required to be typified).

```
List* l1 = create_List(&type_Integer);
List* l2 = create_List(NULL);
append_List(l1,create_Integer(28));      // 28
append_List(l1,create_Integer(47));      // 28 47
append_List(l2,create_Integer(63));      // 63
append_List(l2,create_Integer(85));      // 63 85
appendAll_List(l1,l2);                   // 28 47 63 85
```

Set* toSet_List(List* self)

Returns a *set* that consists all elements of the given list.

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));      // 28
append_List(l,create_Integer(47));      // 28 47
append_List(l,create_Integer(28));      // 28 47 28
s = toSet_List(l);                      // 28 47
t = elemType_Set(s);                    // &type_Integer
```

Set

```
#include <atl/set.h>
```

`Set` data type 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 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.

`Set* create_Set( const Type* elem_type )`

Creates a set and returns a specification reference of `Set` data type. If `elem_type` parameter is `NULL`, type of elements is not restricted. Otherwise the parameter must a pointer to descriptor constant of data type of set's elements:

```
Set* s1 = create_Set(NULL);          // any elements
Set* s2 = create_Set(&type_Integer); // only elements of Integer type
```

This function is defines along with the standard `create()` function:

```
Set* s1 = create(&type_Set, NULL);
Set* s2 = create_Set(NULL);
```

The above ways of creating a specification reference are equivalent.

`Type *elemType_Set( Set* self )`

Returns a pointer to descriptor constant of specification data type, that constrains data type of the set's elements.

If the set is not typified, returns `NULL`.

```
Set* s = create_Set(&type_Integer);
Type *t = elemType_Set(s);             // &type_Integer
```

`bool add_Set( Set* self, Object* ref )`

Adds `ref` element to the set.

If the set is typified, data type of `ref` element must coincide with data type of set's elements.

```
Set* s = create_Set(&type_Integer);
add_Set(s,create_Integer(28));      // 28
add_Set(l,create_Integer(47));      // 28 47
add_Set(l,create_Integer(28));      // 28 47
```

`void remove_Set( Set* self, Object* ref )`

Removes the element from the set.

If the set is typified, data type of `ref` element must coincide with data type of set's elements.

```
Set* s = create_Set(&type_Integer);
add_Set(s,create_Integer(28));      // 28
add_Set(s,create_Integer(47));      // 28 47
remove_Set(s,create_Integer(28));   // 47
```

`void clear_Set( Set* self )`

Removes all elements from the set.

```
Set* s = create_Set(&type_Integer);
add_Set(s,create_Integer(28));      // 28
add_Set(s,create_Integer(47));      // 28 47
clear_Set(s);                       //
```

The same result can be achieved by re-creating the set, but `clear_Set()` function is more effective:

```
Set* s;
...
s = create_Set(elemType_Set(s));
```

`bool contains_Set( Set* self, Object* ref )`

Checks whether the set contains an element, equals to the given.

If the set is typified, data type of `ref` element must coincide with data type of set's elements.

```
Set* s = create_Set(&type_Integer);
add_Set(s,create_Integer(28));              // 28
add_Set(s,create_Integer(47));              // 28 47
if (contains_Set(s,create_Integer(28))) ... // истинно
```

`bool isEmpty_Set( Set* self )`

Checks whether the set is empty.

```
bool empty;
Set* s = create_Set(&type_Integer);
empty = isEmpty_Set(s);                     // true
add_Set(s,create_Integer(28));              // 28
empty = isEmpty_Set(s);                     // false
```

`int size_Set( Set* self )`

Returns number of elements in the set.

```
int size;
Set* s = create_Set(&type_Integer);
size = size_Set(s);                     // 0
add_Set(s,create_Integer(28));          // 28
size = size_Set(s);                     // 1
```

`Object* get_Set( Set* self, int index )`

Returns a specification reference to the element with `index` number.

This function is intended for enumeration of set's elements. Since a set is not ordered, elements are numerated in a random order. Number of position must be in the range from 0 to 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;
add_Set(s1,create_Integer(28));      // 28
add_Set(s1,create_Integer(47));      // 28 47
```

```
add_Set(s1,create_Integer(63));      // 28 47 63
for(i=0; i<size_Set(s1); i++)
  add_Set(s2,get_Set(s1,i));
if (equals(s1,s2)) ...               // true
```

bool containsAll_Set(Set* self, Set* set)

Checks whether `set` is a subset of `self` set.

In other words, checks whether all elements of `set` belong to `self` set.

```
Set* s1 = create_Set(&type_Integer);
Set* s2 = create_Set(NULL);
add_Set(s1,create_Integer(28));      // 28
add_Set(s1,create_Integer(47));      // 28 47
add_Set(s2,create_Integer(28));      // 28
add_Set(s2,create_String("a"));      // 28 a
add_Set(s2,create_Integer(47));      // 28 a 47
if (containsAll_Set(s1,s2)) ...      // ложно
if (containsAll_Set(s2,s1)) ...      // истинно
```

bool addAll_Set(Set* self, Set* set)

Unions `self` set with `set`. Returns `true` if `self` set was changed during the operation, and `false` otherwise.

In other words, adds all elements of `set` to `self` set. If `self` set is typified, types of all elements of `set` must coincide with type of elements of `self` set (`set` itself is not required to be typified).

```
Set* s1 = create_Set(&type_Integer);
Set* s2 = create_Set(NULL);
add_Set(s1,create_Integer(28));      // 28
add_Set(s1,create_Integer(47));      // 28 47
add_Set(s2,create_Integer(28));      // 28
add_Set(s2,create_Integer(47));      // 28 47
add_Set(s2,create_Integer(63));      // 28 47 63
if (addAll_Set(s1,s2)) ...           // истинно; 28 47 63
```

bool retainAll_Set(Set* self, Set* set)

Intersects `self` set with `set`. Returns `true` if `self` set was changed during the operation, and `false` otherwise.

In other words, retains in `self` set only elements that belong to `set`.

```
Set* s1 = create_Set(&type_Integer);
Set* s2 = create_Set(NULL);
add_Set(s1,create_Integer(28));      // 28
add_Set(s1,create_Integer(47));      // 28 47
add_Set(s1,create_Integer(63));      // 28 47 63
add_Set(s2,create_Integer(28));      // 28
add_Set(s2,create_Integer(47));      // 28 47
if (retainAll_Set(s1,s2)) ...        // истинно; 28 47
```

bool removeAll_Set(Set* self, Set* set)

Subtracts `set` from `self` set. Returns `true` if `self` set was changed during the operation, and `false` otherwise.

In other words, retains in `self` set only elements that does not belong to `set`.

```
Set* s1 = create_Set(&type_Integer);
Set* s2 = create_Set(NULL);
add_Set(s1,create_Integer(28));      // 28
add_Set(s1,create_Integer(47));      // 28 47
add_Set(s1,create_Integer(63));      // 28 47 63
add_Set(s2,create_Integer(28));      // 28
add_Set(s2,create_Integer(47));      // 28 47
if (removeAll_Set(s1,s2)) ...        // истинно; 63
```

`List* toList_Set( Set* self )`

Returns a *list* containing all elements of the given set.

Order of elements in the list is not defined. Returned list has the same typification as the set: if elements of the set was constrained by a data type, 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));      // 28
add_Set(s,create_Integer(47));      // 28 47
l = toList_Set(s);                  // 28 47 (или 47 28)
t = elemType_Set(l);                // &type_Integer
```

Map

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

`Map` data type 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 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 actual parameter type is 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.

`Map* create_Map( const Type *key_type, const Type *val_type )`

Creates a map and returns a specification reference of `Map` data type. If `key_type` parameter is `NULL`, type of elements from range of definition is not restricted. Otherwise the parameter must a pointer to descriptor constant of data type of map's keys. In a similar manner, if `val_type` parameter is `NULL`, type of elements from range of values is not restricted. Otherwise the parameter must a pointer to descriptor constant of data type of map's values:

```
Map* m1 = create_Map(NULL, NULL);          // any elements
Map* m2 = create_Map(&type_Integer, NULL); // keys of Integer type only
// map from Integer to String
Map* m3 = create_Map(&type_Integer, &type_String);
```

This function is defines along with the standard `create()` function:

```
Map* m1 = create(&type_Set, NULL, NULL);
Map* m2 = create_Map(NULL, NULL);
```

The above ways of creating a specification reference are equivalent.

`Type *keyType_Map( Map* self )`

Returns a pointer to descriptor constant of specification data type, that constrains data type of the map's keys.

If the range of definition is not typified, returns `NULL`.

```
Map* m = create_Map(&type_Integer, &type_String);
Type *t = keyType_Map(m);                          // &type_Integer
```

`Type *valueType_Map( Map* self )`

Returns a pointer to descriptor constant of specification data type, that constrains data type of the map's values.

If the range of values is not typified, returns `NULL`.

```
Map* m = create_Map(&type_Integer, &type_String);
Type *t = valueType_Map(m);                        // &type_String
```

`void clear_Map( Map* self )`

Removes all elements from the map.

```
Map* m = create_Map(&type_Integer, &type_String);
put_Map(m,create_Integer(28),create_String("a")); // 28 → "a"
put_Map(m,create_Integer(47),create_String("b")); // 28 → "a", 47 → "b"
clear_Map(m);                                     //
```

The same result can be achieved by re-creating the map, but `clear_Map()` function is more effective:

```
Map* m;
...
m = create_Map(keyType_Map(m), valueType_Map(m));
```

`bool containsKey_Map( Map* self, Object* key )`

Checks whether the map contains a key, equals to the given.

If the range of definition is typified, data type of `key` element must coincide with data type of map's keys.

```
Map* m = create_Map(&type_Integer, &type_String);
put_Map(m,create_Integer(28),create_String("a")); // 28 → "a"
put_Map(m,create_Integer(47),create_String("b")); // 28 → "a", 47 → "b"
if (containsKey_Map(m,create_Integer(28))) ...    // истинно
```

`bool containsValue_Map( Map* self, Object* value )`

Checks whether the map contains a value, equals to the given.

If the range of values is typified, data type of `value` element must coincide with data type of map's values.

```
Map* m = create_Map(&type_Integer, &type_String);
put_Map(m,create_Integer(28),create_String("a")); // 28 → "a"
put_Map(m,create_Integer(47),create_String("b")); // 28 → "a", 47 → "b"
if (containsValue_Map(m,create_String("b"))) ...  // истинно
```

`Object* get_Map( Map* self, Object* key )`

Returns a specification reference to the value, corresponding to the given key.

If the range of definition is typified, data type of `key` element must coincide with data type of map's keys. If the map does not contain the given key, returns `NULL`.

```
Map* m = create_Map(&type_Integer, &type_String);
Object* val;
put_Map(m,create_Integer(28),create_String("a")); // 28 → "a"
put_Map(m,create_Integer(47),create_String("b")); // 28 → "a", 47 → "b"
val = get_Map(m,create_Integer(28));              // "a"
```

`Object* getKey_Map( Map* self, Object* value )`

Returns a specification reference to a key, the given value corresponds to.

If the range of definition is typified, data type of `key` element must coincide with data type of map's keys. If the map does not contain any keys, the given value corresponds to, returns `NULL`. Otherwise returns one of the appropriate keys.

```
Map* m = create_Map(&type_Integer, &type_String);
Object* key;
```

```
Object* val;
put_Map(m,create_Integer(28),create_String("a")); // 28 → "a"
put_Map(m,create_Integer(47),create_String("a")); // 28 → "a", 47 → "a"
val = create_String("a");
key = getKey_Map(m,val);                          // 28 or 47
if (equals( get_Map(m,key), val)) ...             // true
```

bool isEmpty_Map(Map* self)

Checks whether the map is empty.

Returns `true` if the map does not contain any "key–value" pair, otherwise returns `false`.

```
bool empty;
Map* m = create_Map(&type_Integer, &type_String);
empty = isEmpty_Map(m);                           // true
put_Map(m,create_Integer(28),create_String("a")); // 28 → "a"
empty = isEmpty_Map(m);                           // false
```

Object* put_Map(Map* self, Object* key, Object* value)

Adds the "key–value" pair to the map.

If the range of definition is typified, data type of `key` element must coincide with data type of map's keys. If the range of values is typified, data type of `value` element must coincide with data type of map's values.

If the map does not contain `key`, the `key` and the corresponding `value` are added to the map; the function returns `NULL`. If the map already contains `key`, the function returns the old corresponding value, and the old value is changed to the new one (`value`).

```
Map* m = create_Map(&type_Integer, &type_String);
put_Map(m,create_Integer(28),create_String("a"));
// returns NULL;  28 → "a"
put_Map(m,create_Integer(47),create_String("b"));
// returns NULL;  28 → "a", 47 → "b"
put_Map(m,create_Integer(28),create_String("c"));
// returns "a";   28 → "c", 47 → "b"
```

void putAll_Map(Map* self, Map* t)

Adds all "key–value" pairs from `t` map to `self` map.

If the range of definition is typified, data type of all keys from `t` map must coincide with data type of map's keys. In a same manner, if the range of values is typified, data type of all values from `t` map must coincide with data type of map's values. (Range of definition and range of values of `t` map are not required to be typified.)

If `self` map already contains a key from `t` map, the corresponding value is changed to the new.

```
Map* m1 = create_Map(&type_Integer, &type_String);
Map* m2 = create_Map(NULL, NULL);
put_Map(m1,create_Integer(28),create_String("a")); // 28 → "a"
put_Map(m1,create_Integer(63),create_String("b")); // 28 → "a", 63 → "b"
put_Map(m2,create_Integer(28),create_String("c")); // 28 → "c"
put_Map(m2,create_Integer(47),create_String("d")); // 28 → "c", 47 → "d"
putAll_Map(m1,m2);     // m1: 28 → "c", 47 → "d", 63 → "b"
```

Object* remove_Map(Map* self, Object* key)

Removes the pair with the key `key` from the map `self`. Returns the value of the removed pair or `NULL` if there is no the pair with the key `key` in the map `self`.

```
Map* m = create_Map(&type_Integer, &type_String);
put_Map(m,create_Integer(28),create_String("a"));
// returns NULL;  28 → "a"
remove_Map(m,create_Integer(28));
// returns "a";
remove_Map(m,create_Integer(28));
// returns NULL;
```

`void clear_Map(Map* self)`

Removes all pairs from the map `self`.

```
Map* m = create_Map(&type_Integer, &type_String);
put_Map(m,create_Integer(28),create_String("a"));
// returns NULL;  28 → "a"
put_Map(m,create_Integer(47),create_String("b"));
// returns NULL;  28 → "a", 47 → "b"
clear_Map(m);
// returns void;
```

`int size_Map( Map* self )`

Returns size of the map.

Size of the map is the number of "key–value" pairs, contained in the map.

```
int size;
Map* m = create_Map(&type_Integer, &type_String);
size = size_Map(m);                                  // 0
put_Map(m,create_Integer(28),create_String("a")); // 28 → "a"
size = size_Map(m);                                  // 1
```

`Object* key_Map( Map* self, int index )`

Returns the key of the map at `index` position.

This function is intended for enumeration of map's keys. Since a set is not ordered, elements are numerated in a random order. Number of position must be in the range from 0 to 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;
put_Map(m1,create_Integer(28),create_String("a")); // 28 → "a"
put_Map(m1,create_Integer(47),create_String("b")); // 28 → "a", 47 → "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);
}
if (equals(m1,m2)) ...                // true
```

SeC grammatics

```
translation_unit ::= ( external_declaration )+ ;

external_declaration ::=   function_definition
                         | declaration
                         | se_invariant_definition
                       ;

function_definition ::= declaration_specifiers
                        declarator
                        ( declaration )*
                        compound_statement
                      ;

se_invariant_definition ::=   "invariant" "(" <ID> ")" compound_statement
                            | "invariant" "(" parameter_declaration ")"
compound_statement
                        ;

/*
 * A.2.2 Declarations
 */

declaration ::= declaration_specifiers ( init_declarator  ( "," init_declarator
)* )? ";" ;

/*
 * Modification of standard syntax:
 *    GCC declaration specifiers have been added.
 */
declaration_specifiers ::= (   storage_class_specifier
                             | type_specifier
                             | type_qualifier
                             | function_specifier
                             | gcc_declaration_specifier
                             | se_declaration_specifier
                           )+
```

```
                        ;

init_declarator ::= declarator ( "=" initializer )? ;

/*
 * Modification of standard syntax:
 *    1. MSVS declspec specifier has been added.
 *    2. se_storage_class_specifier has been added.
 */
storage_class_specifier ::=   "typedef"
                            | "extern"
                            | "static"
                            | "auto"
                            | "register"
                            | msvs_declspec
                            | se_storage_class_specifier
                         ;

se_storage_class_specifier ::= "stable" ;

/*
 * Modification of standard syntax:
 *    Sized integer types (from MSVS) have been added.
 */
type_specifier ::=   "void"
                   | "char"
                   | "short"
                   | "int"
                   | "long"
                   | "float"
                   | "double"
                   | "signed"
                   | "unsigned"
                   | "_Bool"
                   | "_Complex"
                   | "_Imaginary"
                   | struct_or_union_specifier
                   | enum_specifier
                   | typedef_name
                   | msvs_builtin_type
               ;

/*
 * Modification of standard syntax:
 *    1. GCC attributes have been added.
 *    2. GCC allows empty list of struct_declarations.
 */
struct_or_union_specifier ::=   struct_or_union
                                ( gcc_attribute )*
                                ( <ID> )?
                                "{"
                                ( struct_declaration )*
                                "}"
                              | struct_or_union
                                ( gcc_attribute )*
                                <ID>
                          ;

struct_or_union ::= "struct" | "union" ;
```

```
/*
 * Modification of standard syntax:
 *    MSVS supports struct declarations without declarators.
 */
struct_declaration ::= ( specifier_qualifier )+
                       ( struct_declarator ( "," struct_declarator )* )?
                       ";"
                   ;

/*
 * Modification of standard syntax:
 *    GCC declaration specifier has been added.
 */
specifier_qualifier ::=   type_specifier
                        | type_qualifier
                        | gcc_declaration_specifier
                    ;

struct_declarator ::=   declarator
                      | ( declarator )? ":" constant_expr
                  ;

/*
 * Modification of standard syntax:
 *  1. MSVS supports a comma at the end of enumerators list.
 *  2. GCC attributes have been added.
 */
enum_specifier ::=   "enum"
                     ( gcc_attribute )*
                     ( <ID> )?
                     "{"
                     enumerator
                     ( "," enumerator )*
                     ( "," )?
                     "}"
                   | "enum"
                     ( gcc_attribute )*
                     <ID>
               ;

enumerator ::= <ID> ( "=" constant_expr )? ;

type_qualifier ::= "const" | "restrict" | "volatile" ;

function_specifier ::= "inline" ;

se_declaration_specifier ::=   "invariant"
                             | "specification"
                             | "reaction"
                             | "mediator" <ID> "for"
                             | "iterator"
                             | "scenario"
                         ;

/*
 * Modification of standard syntax:
 *   1. GCC attributes optional list has been added.
 *   2. MSVS attributes optional lists have been added.
 */
declarator ::= ( pointer )? ( msvs_attribute )* direct_declarator (
gcc_attribute )* ;
```

```
direct_declarator ::=   <ID>
                      | "(" declarator ")"
                      | direct_declarator "[" ( assignment_expr )? "]"
                      | direct_declarator "[" "*" "]"
                      | direct_declarator
                        "("
                        parameter_type_list
                        ")"
                        ( se_access_description )*
                      | direct_declarator
                        "("
                        ( <ID> ( "," <ID> )* )?
                        ")"
                        ( se_access_description )*
                  ;

/*
 * Modification of standard syntax:
 *   1. GCC attributes have been added.
 *   2. MSVS attributes optional lists have been added.
 */
pointer ::= ( ( msvs_attribute )* "*" ( pointer_qualifier )* )+ ;

pointer_qualifier ::= type_qualifier | gcc_attribute ;

parameter_type_list ::= parameter_declaration
                        ( "," parameter_declaration )*
                        ( "," "..." )?
                    ;

parameter_declaration ::=   declaration_specifiers declarator
                          | declaration_specifiers ( abstract_declarator )?
                      ;

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

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

se_access ::= ( se_access_alias )? assignment_expr ;

se_access_alias ::= <ID> "=" ;

type_name ::= ( specifier_qualifier )+ ( abstract_declarator )? ;

/*
 * Modification of standard syntax:
 *   MSVS attributes optional lists have been added.
 */
abstract_declarator ::=     pointer    ( msvs_attribute )*
                        | ( pointer )? ( msvs_attribute )*
direct_abstract_declarator
                    ;
```

```
direct_abstract_declarator ::=   "(" abstract_declarator ")"
                               | ( direct_abstract_declarator )?
                                 "["
                                 ( assignment_expr )?
                                 "]"
                               | ( direct_abstract_declarator )? "[" "*" "]"
                               | ( direct_abstract_declarator )?
                                 "("
                                 ( parameter_type_list )?
                                 ")"
                           ;

typedef_name ::= <ID> ;

/*
 * Modification of standard syntax:
 *    initializer_list is optional for specification typedef SEC construction.
 */
initializer ::=   assignment_expr
                | "{" ( initializer_list )? ( "," )? "}"
            ;

initializer_list ::= ( designation )?
                     initializer
                     ( "," ( designation )? initializer )*
                 ;

designation ::= ( designator )+ "=" ;

designator ::= "[" constant_expr "]"  |  "." <ID> ;

/*
 * A.2.3 Statements
 */

/*
 * Modification of standard syntax:
 *    MSVS inline assembler statements have been added.
 */
statement ::=   labeled_statement
              | compound_statement
              | expression_statement
              | selection_statement
              | iteration_statement
              | jump_statement
              | msvs_asm_statement
              | se_iteration_statement
              | se_block_statement
          ;

labeled_statement ::=   <ID> ":" statement
                      | "case" constant_expr ":" statement
                      | "default" ":" statement
                  ;

compound_statement ::= "{" ( block_item )* "}" ;

block_item ::= declaration | statement ;

expression_statement ::= ( expression )? ";" ;
```

```
selection_statement ::=   "if"
                          "("
                          expression
                          ")"
                          statement
                          ( "else" statement )?
                        | "switch" "(" expression ")" statement
                    ;

iteration_statement ::=   "while" "(" expression ")" statement
                        | "do" statement "while" "(" expression ")" ";"
                        | "for"
                          "("
                          ( declaration | ( expression )? ";" )
                          ( expression )?
                          ";"
                          ( expression )?
                          ")"
                          statement
                    ;

jump_statement ::=   "goto" <ID> ";"
                   | "continue" ";"
                   | "break" ";"
                   | "return" ( se_return_expression )? ";"
               ;

se_return_expression ::= expression | "{" expression "}" ;

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

se_block_statement ::=   se_pre_block_statement
                       | se_coverage_block_statement
                       | se_post_block_statement
                       | se_call_block_statement
                       | se_state_block_statement
                   ;

se_pre_block_statement ::= "pre" compound_statement ;

se_coverage_block_statement ::=  ( "default" )? "coverage" <ID>
compound_statement ;

se_post_block_statement ::= "post" compound_statement ;

se_call_block_statement ::= "call" compound_statement ;

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

```
/*
 * A.2.2 Expressions
 */

constant ::=   <INTEGER_CONSTANT>
             | <FLOATING_CONSTANT>
             | <ENUMERATION_CONSTANT>
             | <CHARACTER_CONSTANT>
             ;

primary_expr ::= <ID> | constant | <STRING_LITERAL> | "(" expression ")" |
se_primary_expr ;

se_primary_expr ::=   "invariant"
                    | "pre" ( <ID> )?
                    | "coverage" ( <ID> )?
                    | "scenario" <ID>
                ;

postfix_expr ::=   primary_expr
                 | postfix_expr "[" expression "]"
                 | postfix_expr
                   "("
                   ( assignment_expr ( "," assignment_expr )* )?
                   ")"
                 | postfix_expr "."  <ID>
                 | postfix_expr "->" <ID>
                 | postfix_expr "++"
                 | postfix_expr "--"
                 | "(" type_name ")" "{" initializer_list ( "," )? "}"
             ;
/*
 * Modification of standard syntax:
 *    GCC extension specifier has been added.
 */
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
              ;

multiplicative_expr ::=   cast_expr
                        | multiplicative_expr ( "*" | "/" | "%" ) cast_expr
                    ;

additive_expr ::=   multiplicative_expr
                  | additive_expr ( "+" | "-" ) multiplicative_expr
              ;

shift_expr ::=   additive_expr
               | shift_expr ( "<<" | ">>" ) additive_expr
           ;
```

```
relational_expr ::=   shift_expr
                    | relational_expr ( "<" | ">" | "<=" | ">=" ) shift_expr
                  ;

equality_expr ::=   relational_expr
                  | equality_expr ( "==" | "!=" ) relational_expr
                ;

AND_expr ::=   equality_expr
             | AND_expr "&" equality_expr
           ;

exclusive_OR_expr ::=   AND_expr
                      | exclusive_OR_expr "^" AND_expr
                    ;

inclusive_OR_expr ::=   exclusive_OR_expr
                      | inclusive_OR_expr "|" exclusive_OR_expr
                    ;

logical_AND_expr ::=   inclusive_OR_expr
                     | logical_AND_expr "&&" inclusive_OR_expr
                   ;

logical_OR_expr ::=   logical_AND_expr
                    | logical_OR_expr "||" logical_AND_expr
                  ;

se_logical_impl_expr ::=   logical_OR_expr
                         | se_logical_impl_expr "=>" logical_OR_expr
                       ;

/*
 * Modification of standard syntax:
 *    The else branch of conditional_expr makes optional for conditional
access_descs.
 */
conditional_expr ::=   se_logical_impl_expr
                     | se_logical_impl_expr "?" expression ( ":"
conditional_expr )?
                   ;

assignment_expr ::=   conditional_expr
                    | unary_expr assignment_operator assignment_expr
                  ;

assignment_operator ::=   "="
                        | "*="
                        | "/="
                        | "%="
                        | "+="
                        | "-="
                        | "<<="
                        | ">>="
                        | "&="
                        | "^="
                        | "|="
                      ;

expression ::= assignment_expr ( "," assignment_expr )* ;

constant_expr ::= conditional_expr ;
```

```
/*
 * Microsoft Extensions
 */
msvs_attribute ::= "__asm" | "__fastcall" | "__based" | "__inline" | "__cdecl" |
"__stdcall" ;

msvs_builtin_type ::=  "__int8"
                     | "__int16"
                     | "__int32"
                     | "__int64"
                  ;

msvs_declspec ::= "__declspec" "(" ( msvs_extended_decl_modifier )* ")" ;

msvs_extended_decl_modifier ::= "thread" | "naked" | "dllimport" | "dllexport" |
"noreturn" ;

msvs_asm_statement ::=   "__asm" msvs_asm_directive
                       | "__asm"
                         "{"
                         ( msvs_asm_directive )+
                         "}"
                 ;

msvs_asm_directive ::=   ( msvs_asm_label_def )? msvs_asm_segment_directive
                       | msvs_asm_label_def
                 ;

msvs_asm_label_def ::=   <ID> ":"
                       | <ID> ":" ":"
                       | "@" "@" ":"
                 ;

/*
 * msvs_asm_segment_directive ::=   msvs_asm_instruction
 *                                | msvs_asm_data_directive
 *                                | msvs_asm_control_directive
 *                                | msvs_asm_startup_directive
 *                                | msvs_asm_exit_directive
 *                                | msvs_asm_offset_directive
 *                                | msvs_asm_label_directive
 *                                | msvs_asm_proc_directive
 *                                  ( msvs_asm_local_directive )*
 *                                  ( msvs_asm_directive )*
 *                                  msvs_asm_endp_directive
 *                                | msvs_asm_invoke_directive
 *                                | msvs_asm_general_directive
 *                           ;
 *
 * The full MASM instruction set is not supported.
 */
msvs_asm_segment_directive ::=   msvs_asm_instruction ;

msvs_asm_instruction ::= ( msvs_asm_instr_prefix )?
                         msvs_asm_mnemonic
                         ( msvs_asm_expr ( "," msvs_asm_expr )* )?
                   ;

msvs_asm_instr_prefix ::= "REP" | "REPE" | "REPZ" | "REPNE" | "REPNZ" | "LOCK" ;

msvs_asm_mnemonic ::= <ID> | "AND" | "MOD" | "NOT" | "OR" | "SEG" | "SHL" |
"SHR" | "XOR" ;
```

```
/*
 * msvs_asm_expr ::=   "SHORT"  msvs_asm_expr05
 *                   | ".TYPE"  msvs_asm_expr01
 *                   | "OPATTR" msvs_asm_expr01
 *                   | msvs_asm_expr01
 *               ;
 *
 * The full MASM instruction set is not supported.
 */
msvs_asm_expr ::= msvs_asm_expr01 ;

msvs_asm_expr01 ::=   msvs_asm_expr01 "OR"  msvs_asm_expr02
                    | msvs_asm_expr01 "XOR" msvs_asm_expr02
                    | msvs_asm_expr02
                ;

msvs_asm_expr02 ::=   msvs_asm_expr02 "AND" msvs_asm_expr03
                    | msvs_asm_expr03
                ;

msvs_asm_expr03 ::=   "NOT" msvs_asm_expr04
                    | msvs_asm_expr04
                ;

msvs_asm_expr04 ::=   msvs_asm_expr04 "EQ" msvs_asm_expr05
                    | msvs_asm_expr04 "NE" msvs_asm_expr05
                    | msvs_asm_expr04 "LT" msvs_asm_expr05
                    | msvs_asm_expr04 "LE" msvs_asm_expr05
                    | msvs_asm_expr04 "GT" msvs_asm_expr05
                    | msvs_asm_expr04 "GE" msvs_asm_expr05
                    | msvs_asm_expr05
                ;

msvs_asm_expr05 ::=   msvs_asm_expr05 "+" msvs_asm_expr06
                    | msvs_asm_expr05 "-" msvs_asm_expr06
                    | msvs_asm_expr06
                ;

msvs_asm_expr06 ::=   msvs_asm_expr06 "*"   msvs_asm_expr07
                    | msvs_asm_expr06 "/"   msvs_asm_expr07
                    | msvs_asm_expr06 "MOD" msvs_asm_expr07
                    | msvs_asm_expr06 "SHR" msvs_asm_expr07
                    | msvs_asm_expr06 "SHL" msvs_asm_expr07
                    | msvs_asm_expr07
                ;

msvs_asm_expr07 ::=   "+" msvs_asm_expr08
                    | "-" msvs_asm_expr08
                    | msvs_asm_expr08
                ;

msvs_asm_expr08 ::=   "HIGH"     msvs_asm_expr09
                    | "LOW"      msvs_asm_expr09
                    | "HIGHWORD" msvs_asm_expr09
                    | "LOWWORD"  msvs_asm_expr09
                    | msvs_asm_expr09
                ;
```

```
msvs_asm_expr09 ::=   "OFFSET"   msvs_asm_expr10
                    | "SEG"      msvs_asm_expr10
                    | "LROFFSET" msvs_asm_expr10
                    | "TYPE"     msvs_asm_expr10
                    | "THIS"     msvs_asm_expr10
                    | msvs_asm_expr09 "PTR" msvs_asm_expr10
                    | msvs_asm_expr09 ":"   msvs_asm_expr10
                    | msvs_asm_expr10
                ;

msvs_asm_expr10 ::=   msvs_asm_expr10 "." msvs_asm_expr11
                    | msvs_asm_expr10 "[" msvs_asm_expr "]"
                    | msvs_asm_expr11
                ;

/*
 * msvs_asm_expr11 ::=   "(" msvs_asm_expr ")"
 *                     | "[" msvs_asm_expr "]"
 *                     | "WIDTH" <ID>
 *                     | "MASK"  <ID>
 *                     | "SIZE"    msvs_asm_size_arg
 *                     | "SIZEOF"  msvs_asm_size_arg
 *                     | "LENGTH"   <ID>
 *                     | "LENGTHOF" <ID>
 *                     | msvs_asm_record_const
 *                     | msvs_asm_string
 *                     | msvs_asm_constant
 *                     | msvs_asm_type
 *                     | <ID>
 *                     | "$"
 *                     | msvs_asm_segment_register
 *                     | msvs_asm_register
 *                     | "ST"
 *                     | "ST" "(" msvs_asm_expr ")"
 *                 ;
 *
 * The full MASM instruction set is not supported.
 */
msvs_asm_expr11 ::=   "(" msvs_asm_expr ")"
                    | "[" msvs_asm_expr "]"
                    | msvs_asm_constant
                    | msvs_asm_type
                    | <ID>
                    | "$"
                    | msvs_asm_segment_register
                    | msvs_asm_register
                ;

msvs_asm_type ::=   <ID>
                  | msvs_asm_distance
                  | msvs_asm_data_type
              ;

msvs_asm_distance ::=   msvs_asm_nearfar
                      | "NEAR16"
                      | "NEAR32"
                      | "FAR16"
                      | "FAR32"
                  ;

msvs_asm_nearfar ::= "NEAR" | "FAR" ;
```

```
msvs_asm_data_type ::=   "BYTE"
                       | "SBYTE"
                       | "WORD"
                       | "SWORD"
                       | "DWORD"
                       | "SDWORD"
                       | "FWORD"
                       | "QWORD"
                       | "TBYTE"
                       | "REAL4"
                       | "REAL8"
                       | "REAL10"
                   ;

msvs_asm_segment_register ::= "CS" | "DS" | "ES" | "FS" | "GS" | "SS" ;

msvs_asm_register ::=   msvs_asm_special_register
                      | msvs_asm_gp_register
                      | msvs_asm_byte_register
                  ;

msvs_asm_special_register ::=   "CR0" | "CR2" | "CR3"
                              | "DR0" | "DR1" | "DR2" | "DR3" | "DR6" | "DR7"
                              | "TR3" | "TR4" | "TR5" | "TR6" | "TR7"
                          ;

msvs_asm_gp_register ::=   "AX" | "EAX" | "BX" | "EBX" | "CX" | "ECX" | "DX" |
"EDX"
                         | "BP" | "EBP" | "SP" | "ESP" | "DI" | "EDI" | "SI" |
"ESI"
                     ;

msvs_asm_byte_register ::= "AL" | "AH" | "BL" | "BH" | "CL" | "CH" | "DL" | "DH"
;

msvs_asm_constant ::= <INTEGER_CONSTANT> ;


/*
 * GCC Extensions
 */
gcc_declaration_specifier ::=   gcc_attribute
                              | gcc_extension_specifier
                          ;

gcc_attribute ::= "__attribute__" "(" "(" gcc_attribute_parameter ( ","
gcc_attribute_parameter )* ")" ")" ;

gcc_attribute_parameter ::=   ( gcc_any_word )?
                            | gcc_any_word "(" ( assignment_expr ( ","
assignment_expr )* )? ")"
                        ;

gcc_any_word ::=   <ID>
                 | storage_class_specifier
                 | type_specifier
                 | type_qualifier
                 | function_specifier
             ;
```

gcc_extension_specifier ::= "__extension__" ;