CTesK 2.2 Community Edition: Описание языка SeC

Содержание

Алфавитный указатель

Введение

Язык SeC является расширением языка программирования C и специально разработан для поддержки технологии тестирования UniTesK. Ниже кратко рассмотрены основные особенности указанной технологии.

Тестирование должно демонстрировать, что тестируемая система удовлетворяет требованиям, предъявляемым к ней, на некотором достаточно показательном наборе ситуаций. Для этой цели при разработке тестов необходимо иметь полное и точное описание требований к тестируемой системе. Дополнительно нужно определить набор тестовых ситуаций, на котором будет проводиться тестирование.

Технология тестирования UniTesK сфокусирована на функциональном тестировании, нацеленным на проверку соответствия поведения тестируемой системы функциональным требованиям, то есть требованиям, формулирующим, что система должна делать. Одна из основных целей UniTesK – автоматизировать процесс разработки тестов для функционального тестирования.

Функциональные требования описываются в форме спецификаций, которые состоят из описания интерфейса тестируемой системы и определения корректного результата для каждого элемента интерфейса. Спецификации, представляемые в форме, обеспечивающей возможность автоматической обработки компьютером, называются формальными спецификациями. Из формальных спецификаций можно автоматически генерировать оракулы – компоненты тестовой системы, проверяющие соответствие поведения тестируемой системы требованиям, записанным в спецификации.

Инструмент CTesK поддерживает автоматическую генерацию оракулов из спецификаций, написанных на языке SeC.

Как описать достаточный набор тестовых ситуаций?

Обычный способ сделать это – определить некоторую метрику тестового покрытия и значение этой метрики, достаточное для того, чтобы считать, что система качественно протестирована. Примеры широко используемых на практике метрик: “процент исполненных во время тестирования инструкций” и “процент строк исходного кода, покрываемых тестом”.

При функциональном тестировании высокий уровень покрытия строк исходного кода еще недостаточен, чтобы утверждать, что тестируемая система качественно протестирована, хотя и важен для демонстрации того, что протестированы почти все части ее реализации.

Поэтому рассматриваются метрики тестового покрытия в терминах функциональности тестируемой системы, а не в терминах исходного кода. Поскольку спецификации формально представляют функциональные требования, метрики, определенные в терминах структуры спецификации (например, “процент функциональных ветвей, покрываемых тестом”) гораздо лучше соответствуют функциональности, чем метрики, основанные на исходном коде.

Инструмент CTesK поддерживает автоматическое вычисление значений метрик тестового покрытия для спецификаций, написанных на языке SeC.

Когда есть описание функциональности тестируемой системы и значение метрики тестового покрытия, которое нужно достичь, единственная проблема — генерация последовательности тестовых воздействий для его достижения.

Для генерации тестовой последовательности технология тестирования UniTesK предлагает специальную технику, основанную на понятии конечных автоматов. Язык CTesK позволяет пользователю описывать автоматную модель тестируемой системы в компактной, удобной для повторного использования форме. По данному описанию автоматически строится обход графа состояний конечного автомата, в результате которого генерируется требуемая тестовая последовательность.

Общие сведения о языке SeC

Язык SeC является расширением языка программирования C и предназначен для разработки тестов на основе формальных спецификаций. В нем добавлены специальные конструкции, которые позволяют компактным и удобным образом описывать требования к тестируемой системе и другие компоненты тестовой системы. Это делает разработку тестов максимально удобной, а также позволяет сократить затраты на обучение специалистов, уже знакомых с языком программирования C.

Язык SeC дополнительно вводит спецификацинные типы (см. *Спецификационные типы*), инварианты типов данных и глобальных переменных, тестовые сценарии (см. *Тестовые сценарии*) и три вида функций: спецификационные (см. *Спецификационные функции, Отложенные реакции*), медиаторные (см. *Медиаторные функции*) и сценарные (см. *Сценарные функции*). Эти типы, инварианты и функции определяются в спецификационных файлах с расширением **`sec`**. Спецификационные заголовочные файлы, содержащие декларации спецификационных типов и функций должны находится в файлах с расширением **`seh`**. Спецификационные заголовочные файлы включаются в спецификационные файлы при помощи директивы препроцессора C **`#include`**. Спецификационные файлы могут содержать и обычные функции C, требуемые для различных вспомогательных целей. При необходимости использования специальных данных или функций используется включение соответствующих обычных заголовочных файлов языка C при помощи директивы препроцессора C **`#include`**.

Для удобства записи и чтения логических выражений в язык SeC дополнительно введен оператор импликации **`=>`**, являющийся бинарным инфиксным оператором, приоритет которого меньше приоритета оператора дизьюнкции **`||`**, но больше приоритета условного оператора **`?:`**. Выражение **`x => y`** эквивалентно выражению **`!x || y`**, и при его вычислении, также как при вычислении других логических операторов, действуют правила короткой логики. Оператор импликации ассоциативен слева направо, то есть выражение **`x => y => z`** эквивалентно выражению **`(x => y) => z`**.

Спецификации

Спецификации представляют собой формальное описание требований к тестируемой системе в форме инвариантов данных и спецификации поведения тестируемой системы.

Для описания требований спецификации могут использовать как непосредственно данные тестируемой системы, так и собственную спецификационную модель данных. Привязка данных спецификационной модели к данным тестируемой системы, спецификационных функций к функциям тестируемой системы и отложенных реакций к реакциям реализации осуществляется при помощи медиаторов (см. *Медиаторы*).

Спецификационные типы

Назначение

Спецификационные типы объединяют тип языка C с базовыми функциями работы с ним: создание, инициализация, копирование, сравнение, построение строкового представления, уничтожения.

Описание

```
specification typedef базовый_тип новый_тип =
{
  .init = указатель_на_функцию_инициализации
, .copy = указатель_на_функцию_копирования
, .compare = указатель_на_функцию_сравнения
, .to_string = указатель_на_функцию_преобразования_в_строку
, .enumerator =
      указатель_на_функцию_перечисляющую_содержащиеся_объекты
, .destroy = указатель_на_функцию_освобождающую_ресурсы
};
```

Спецификационные типы вводятся с помощью обычной конструкции языка C typedef, помеченной ключевым словом SeC specification. В данной конструкции может встречаться несколько деклараций с инициализаторами. Каждая из них вводит новый спецификационный тип со своим именем. Если в декларации инициализатор отсутствует, то это предварительная декларация спецификационного типа, в противном случае — это определение спецификационного типа.

Значения спецификационных типов размещаются в динамической памяти, управление памятью автоматизировано — при изменении количества ссылок на значение, счетчик ссылок на данное значение автоматически изменяется, значение удаляется автоматически после уничтожения последней ссылки.

В спецификационном расширении языка C существует встроенный спецификационный тип Object, который является неполным спецификационным типом (incomplete specification type). Он является базовым для всех спецификационных типов, но объектов этого типа быть не может. Тип ссылки на спецификационный тип Object, то есть Object*, используется по аналогии с void*. Любая ссылка на спецификационный тип может быть преобразована к ссылке на Object и наоборот. Если при обратном преобразовании тип объекта по ссылке не совместим с типом, к которому осуществляется преобразование, то поведение системы не определено.

Синтаксис

```
declaration ::= (  (( declaration_specifiers )?
                    "specification"
                    ( declaration_specifiers )?
                    "typedef"
                    ( declaration_specifiers )?
                   )
                 | (( declaration_specifiers )?
                    "typedef"
                    ( declaration_specifiers )?
                    "specification"
                    ( declaration_specifiers )?
                   )
```

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

        ;
```

Семантические ограничения

- Спецификационные типы не могут быть локальными.
- Имена спецификационных типов входят в тоже пространство имен, что и typedef-имена.
- Определение спецификационного типа с данным именем (декларатор с инициализатором) может встречаться только один раз во всех единицах трансляции (translation_unit), собираемых в единую систему.
- Инициализатор в определении спецификационного типа должен иметь вид:
  = { .<field(1)> = <expr>, .<field(2)> = <expr>, ... },
  где между фигурными скобками находится (возможно пустой) набор конструкций вида .<field(i)> = <expr>, разделенных запятыми. <field(i)> может принимать одно из следующих значений: init, copy, compare, to_string, enumerate или destroy. <expr> должно иметь тип, соответствующий указанному полю:

```
/* 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 );
```

  Подробную информацию о функциях, указателями на которые должны быть инициализированы поля при определении спецификационного типа, можно найти в главе «Библиотека спецификационных типов».

- Если инициализация поля в определении спецификационного типа отсутствует, то используется функция по умолчанию. В функциях по умолчанию указатели на все типы кроме спецификационных, функциональных и void интерпретируются как указатели на единственное значение этого типа. То есть случае когда базовый тип является указателем на несколько значений (строка, массив), в определении спецификационного типа должны быть инициализированы все поля.
- В качестве базового типа при определении спецификационного типа запрещается использовать
  1. спецификационные, функциональные и неполные типы,

2. объединения, массивы, и структуры, содержащие вышеперечисленные типы.

Если в определении спецификационного типа отсутствует инициализация хотя бы одного поля, то кроме вышеперечисленных типов запрещается использовать в качестве базового типа

3. объединения и массивы переменной длины,
4. структуры и массивы, содержащие все вышеперечисленные типы,
5. указатели на типы, перечисленные в пунктах 2, 3 и 4.

- Если в декларации спецификационного типа присутствует ключевое слово invariant, то для данного типа должен быть определен инвариант, ограничивающий множество его значений (см. Инварианты типов).
- Если в хотя бы в одной декларации спецификационного типа присутствует ключевое слово invariant, то оно должно присутствовать во всех декларациях, а также в определении данного типа.
- Спецификационные типы аналогично неполным типам языка C могут использоваться только через указатели, то есть не допускаются объявления переменных спецификационных типов, определения объединений, структур и массивов, содержащих поля и элементы спецификационных типов и т. д.

Пример

Предварительная декларация:

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

Определение:

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

Инварианты типов

Назначение

Инварианты типов предназначены для описания ограничений на типы данных, используемые в спецификациях. Их можно рассматривать как общие части всех пред- и постусловий функций, результаты которых зависят от данных этих типов. С другой стороны, инвариант типа можно рассматривать как ограничения подтипа, определяющие целостность данных типа.

Описание

Определение типа с инвариантом:

```
invariant typedef базовый_тип подтип;
```

Определение инварианта:

```
invariant (подтип параметр)
{
  ...
  return значение_булевского_типа;
  ...
}
```

Вызов инварианта:

```
invariant(проверяемое_значение)
```

Для декларации инвариантов типов используется обычная конструкция языка C typedef, помеченная ключевым словом SeC invariant. Данная конструкция, также как и обычный typedef, вводит новые имена типов. Основное ее отличие состоит в том, что множество значений определяемого типа не совпадает со множеством значений базового типа, а является его подмножеством. Таким образом, определяется не синоним для базового типового выражения, а новый тип с собственным множеством значений.

Ограничения на множество значений базового типа описываются в составном операторе (compound_statement), который синтаксически и семантически эквивалентен телу функции без побочных эффектов, возвращающей значение булевского типа, с одним параметром, который должен быть:

- определяемого подтипа, если базовый тип является обычным типом языка C;
- указателем на определяемый подтип спецификационного типа, если базовый тип является спецификационным типом.

Составной оператор инварианта помечается модификатором invariant, после которого в скобках указывается формальный параметр соответствующего типа. Поскольку тип возвращаемого значения фиксирован, он не указывается.

Инвариант типа может быть вызван для выражения соответствующего типа. Выражение вызова инварианта типа состоит из ключевого слова invariant, за которым в скобках следует выражение, для значения которого проверяется выполнение инварианта. Значением данного выражения вызова инварианта типа является true, если значение проверяемого выражения в точке вызова удовлетворяет ограничениям инварианта, и false — в обратном случае.

Синтаксис

Определение типа с инвариантом:

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

          ;
```

Определение инварианта типа:

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

Вызов инварианта типа:

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

Семантические ограничения

- Инвариант типа не может быть определен для локального типа.
- До точки определения или вызова инварианта тип должен быть определен при помощи конструкции typedef, причем среди списка спецификаторов его декларации (declaration_specifiers) должен встречаться спецификатор SeC (se_declaration_specifier) invariant.
- Если в одной из единиц трансляции (translation_unit) при определении типа среди спецификаторов декларации (declaration_specifiers) встречается спецификатор SeC (se_declaration_specifier) invariant, то он должен встречаться в определениях этого типа во всех единицах трансляции (translation_unit), собираемых в единую систему, и определение инварианта этого типа должно встречаться ровно один раз во всех единицах трансляции (translation_unit), собираемых в единую систему.
- В качестве базового типа при определении типа с инвариантом запрещается использовать функциональные типы и void.
- Если базовый тип в определении типа с инвариантом является спецификационным типом, то в определении инварианта параметр декларируется как указатель на определяемый тип. При этом пользователь может быть уверен, что в теле инварианта эта ссылка имеет ненулевое значение. При определении инвариантов остальных типов (в том числе указателя на спецификационный тип), параметр декларируется с определяемым типом.
- В выражении вызова инварианта типа в скобках должно содержатся выражение типа, выполнение инварианта которого проверяется.

Пример

Декларация:

```
invariant typedef int Nat;
```

или

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

Определение:

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

или

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

Вызов:

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

Инварианты переменных

Назначение

Инварианты переменных предназначены для описания ограничений на значения глобальных переменных, используемых в спецификациях. Их можно рассматривать как общие части всех пред- и постусловий функций, результаты которых зависят от значений этих переменных. С другой стороны, инвариант переменной можно рассматривать как ограничения, определяющие целостность данных, содержащихся в переменной.

Описание

Декларация переменной с инвариантом:

```
invariant тип переменная;
```

Определение инварианта:

```
invariant (переменная)
{
  ...
  return значение_булевского_типа;
  ...
}
```

Вызов инварианта:

```
invariant(имя_переменной)
```

Если глобальная переменная не может принимать все возможные значения своего типа, то необходимо ограничить множество ее допустимых значений с помощью инварианта переменной. Для этого во всех декларациях переменной в ее определении указывается ключевое слово SeC invariant. Это слово говорит о том, что все переменные вводимые данной декларацией имеют ограничение на множество допустимых значений.

Ограничения на множество значений переменной описываются в составном операторе (compound_statement), который синтаксически и семантически эквивалентен телу функции без побочных эффектов, без параметров, и возвращающей значение булевского типа.

Составной оператор инварианта помечается модификатором invariant, после которого в скобках указывается идентификатор переменной. Поскольку тип возвращаемого значения фиксирован, он не указывается.

Инвариант переменной может быть вызван для проверки его выполнения. Выражение вызова инварианта переменной состоит из ключевого слова invariant, за которым следует имя переменной в скобках. Значением результата вычисления выражения вызова инварианта является true, если значение переменной в точке вызова удовлетворяет ограничениям инварианта, и false — в обратном случае.

Синтаксис

Предварительная декларация и определение переменной с инвариантом:

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

Определение инварианта переменной:

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

Вызов инварианта переменной:

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

Семантические ограничения

- Инварианты переменных определяются только для глобальных переменных.
- До точки определения или вызова инварианта переменная уже должна быть декларирована, причем среди списка спецификаторов ее декларации (declaration_specifiers) должен встречаться спецификатор SeC (se_declaration_specifier) invariant.
- Если в определении или в одной из деклараций переменной среди списка спецификаторов ее декларации (declaration_specifiers) встречается спецификатор SeC (se_declaration_specifier) invariant, то он должен встречаться в декларациях и определении этой переменной, и определение инварианта этой переменной должно встречаться ровно один раз во всех единицах трансляции (translation_unit), собираемых в единую систему.
- В определении инварианта и выражении вызова инварианта переменной в скобках содержится единственный идентификатор — имя переменной, для которой определен или продекларирован инвариант.
- Тело инварианта должно быть синтаксически и семантически эквивалентено телу функции без побочных эффектов, без параметров, и возвращающей значение булевского типа.

Пример

Декларация переменной с инвариантом:

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

Определение:

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

Вызов:

```
invariant (EvenNum);
```

Спецификационные функции

Назначение

Спецификационные функции предназначены для спецификации поведения тестируемой системы при внешнем воздействии на нее через некоторую часть интерфейса. Спецификационные функции описывают поведение в форме ограничений доступа к данным, предусловий, критериев покрытия и постусловий.

Описание

Декларация:

```
specification сигнатура ограничения_доступа;
```

Определение:

```
specification сигнатура ограничения_доступа
{
  дополнительный_код
  pre {...}
  {
    дополнительный_код
    coverage имя_1 {...}
    ...
    coverage имя_n {...}
    {
      дополнительный_код
      post {...}
      дополнительный_код
    }
    дополнительный_код
  }
  дополнительный_код
}
```

Вызов:

```
имя_спецификационной_функции(аргументы)
```

Спецификационные функции декларируются и определяются в спецификационных файлах и помечаются ключевым словом SeC specification. Они могут содержать следующие элементы:

- Описание ограничений доступа данной функции к глобальным переменным и параметрам (см. Ограничения доступа)
- Предусловие, описывающее ситуации, в которых определено поведение тестируемой системы (см. Предусловия)
- Критерии покрытия, описывающие разбиение на ветви функциональности поведения тестируемой системы при взаимодействии с ней через часть интерфейса, описываемого данной спецификационной функцией (см. Критерии покрытия)
- Постусловие, описывающее ограничения, которым должны удовлетворять результаты работы тестируемой системы, описываемой данной спецификационной функцией (см. Постусловия)

- Дополнительный код на SeC вне предусловия, критериев покрытия и постусловия

Спецификационные функции вызываются так же как обычные функции. При тестировании в точке вызова спецификационной функции происходит проверка инвариантов типов выражений с доступом reads и updates, проверка инвариантов переменных с доступом reads и updates, проверка значений переданных аргументов на выполнение предусловия, вычисление элементов покрытий для значений переданных аргументов, обращение к медиатору, установленному для вызываемой спецификационной функции, проверка неизменности выражений с доступом reads, проверка инвариантов типов выражений с доступом writes и updates, проверка инвариантов переменных с доступом writes и updates, проверка выполнения постусловия.

Синтаксис

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

Семантические ограничения

- Имена спецификационных функций входят в то же пространство имен, что и имена обычных функций языка C.
- Спецификационная функция должна быть определена ровно один раз среди всех единиц трансляции (translation_unit), собираемых в одну систему.
- Для всех глобальных переменных и параметров (или их составных частей), которые используются в спецификационной функции, в декларатore определения функции должны быть указаны ограничения доступа (se_access_description, см. Ограничения доступа).
- В составном операторе спецификационной функции должно присутствовать ровно одно постусловие (se_post_block_statement, см. Постусловия) после блоков критериев покрытия и предусловия, если они присутствуют.
- В составном операторе спецификационной функции может присутствовать не больше одного предусловия (se_pre_block_statement, см. Предусловия) до блоков критериев покрытия, если они присутствуют, и до блока постусловия, после предусловия должен следовать либо составной оператор, либо первый критерий покрытия, либо постусловие.
- В составном операторе спецификационной функции могут присутствовать несколько критериев покрытия (se_coverage_block_statement, см. Критерии покрытия), следующих друг за другом после блока предусловия, если он присутствует, и до блока постусловия, между блоками критериев покрытия не допускается никакого кода, после последнего критерия покрытия должен следовать либо составной оператор, либо постусловие
- Дополнительный код в составном операторе спецификационной функции вне предусловия, критериев покрытия и постусловия допускается только:
    - до предусловия и в конце после критериев покрытия, постусловия и окончания составных операторов, содержащих критерии покрытия и постусловие,

  - в составном операторе после предусловия — в начале до первого критерия покрытия или постусловия (если критерии покрытия отсутствуют) и в конце после критериев покрытия, постусловия и окончания составного оператора, содержащего постусловие,
  - в составном операторе после последнего критерия покрытия — в начале до постусловия и в конце после него.

- Спецификационная функция должна быть без видимых снаружи побочных эффектов:
  - значения глобальных переменных и значения данных, передаваемых по указателям, не должны изменяться
  - динамическая память, выделяемая в спецификационной функции, должна освобождаться, причем на том же уровне вложенности (в том же составном операторе), на котором она выделялась:
    - если память выделяется в предусловие, критерии покрытия или постусловие, она должна освобождаться в том же блоке,
    - если память выделяется в начале спецификационной функции, она должна освобождаться в конце после предусловия, критериев покрытия, постусловия и окончания составных операторов, содержащих критерии покрытия и постусловие,
    - если память выделяется в составном операторе после предусловия, то она должна освобождаться в конце этого составного оператора после критериев покрытия, постусловия и окончания составного оператора, содержащего постусловие,
    - если память выделяется в составном операторе перед постусловием, то она должна освобождаться в его конце после постусловия.

- Предварительная декларация спецификационной функции кроме сигнатуры, помеченной ключевым словом specification, должна включать описание ограничений доступа (см. Ограничения доступа), причем во всех декларациях ограничения доступа должны быть одинаковы.

Пример

Декларация:

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

Определение:

```
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 );
   }
}
```

Вызов:

```
sqrt_spec(5.2)
```

Отложенные реакции

Назначение

Отложенные реакции предназначены для описания поведения тестируемой системы при отложенном реагирования системы на внешние воздействия. Отложенные реакции описывают поведение в форме ограничений доступа к данным, предусловий и постусловий.

Описание

```
reaction сигнатура ограничения_доступа
{
  дополнительный_код
  pre {...}
  {
    дополнительный_код
    post {...}
    дополнительный_код
  }
  дополнительный_код
}
```

Отложенные реакции декларируются и определяются в спецификационных файлах в виде функций без параметров, помеченных ключевым словом SeC reaction и возвращающих указатели на спецификационные типы. Они могут содержать следующие элементы:

- Описание ограничений доступа данной реакции к глобальным переменным (см. Ограничения доступа).
- Предусловие, описывающее ситуации, в которых возможно возникновение данной реакции (см. Предусловия).
- Постусловие, описывающее ограничения на значения глобальных переменных, которые должны выполняться в случае возникновения данной реакции (см. Постусловия).
- Дополнительный код на SeC вне предусловия и постусловия.

Синтаксис

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

Семантические ограничения

- Имена отложенных реакций входят в то же пространство имен, что и имена обычных функций языка C.
- Отложенная реакция должна быть определена ровно один раз среди всех единиц трансляции (translation_unit), собираемых в одну систему.

- У отложенных реакций не бывает параметров, типом возвращаемого значения отложенной реакции может быть только указатель на спецификационный тип.
- Для всех глобальных переменных (или их составных частей), которые используются в отложенной реакции должны быть указаны ограничения доступа (se_access_description, см. Ограничения доступа).
- В составном операторе отложенной реакции должно присутствовать ровно одно постусловие (se_post_block_statement, см. Постусловия) после предусловия, если оно присутствует.
- В составном операторе отложенной реакции может присутствовать не больше одного предусловия (se_pre_block_statement, см. Предусловия) до постусловия, после предусловия должен следовать либо составной оператор, либо постусловие.
- Дополнительный код вне предусловия и постусловия допускается только:
    - в составном операторе реакции — в начале до предусловия и в конце после постусловия и окончания составного оператора, содержащего постусловие
    - в составном операторе после предусловия — в начале до постусловия и в конце после него
- Отложенная реакция должна быть без видимых снаружи побочных эффектов:
    - значения глобальных переменных не должны изменяться,
    - динамическая память, выделяемая в реакции должна освобождаться, причем на том же уровне вложенности (в том же составном операторе), на котором она выделялась:
        - если память выделяется в предусловии или постусловии, она должна освобождаться в том же блоке,
        - если память выделяется в начале реакции, она должна освобождаться в конце после предусловия, постусловия и окончания составного оператора, содержащего постусловие,
        - если память выделяется в составном операторе после предусловия, то она должна освобождаться в конце этого составного оператора после постусловия
- Предварительная декларация спецификационной функции кроме сигнатуры, помеченной ключевым словом reaction, должна включать описание ограничений доступа (см. Ограничения доступа), причем во всех декларациях ограничения доступа должны быть одинаковы

Пример

```
int last_message_id = 1;

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

Ограничения доступа

Назначение

Ограничения доступа предназначены для описания способа использования внутри спецификационных функций и отложенных реакций глобальных переменных и параметров, а так же выражений с ними, результаты которых могут быть правой частью выражений с оператором присваивания. Ограничения доступа используются для проверки правильности работы поведения системы. Язык SeC поддерживает три вида ограничений доступа: чтение, запись и изменение.

Описание

```
reads   выражение_1, ..., имя_псевдонима = выражение_n, ...
writes  выражение_1, ..., имя_псевдонима = выражение_n, ...
updates выражение_1, ..., имя_псевдонима = выражение_n, ...
```

Ограничения доступа описываются после сигнатуры функции или реакции. Для указания доступа на чтение используется модификатор reads, на запись — writes, на изменение — updates. Действие модификатора доступа распространяется на все перечисленные вслед за ним через запятую идентификаторы до следующего модификатора или начала тела функции или реакции. В ограничениях доступа можно объявлять псевдонимы (см. Псевдонимы) выражений, на которые описываются ограничения доступа.

Синтаксис

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


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


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

Грамматику элемента se_access_alias можно найти в разделе Псевдонимы.

Семантические ограничения

- В пределах одного описания не должно быть двух ограничений доступа, имеющих разные модификаторы доступа и одно и то же выражение, к которому указывается доступ.
- Если для выражения указано ограничение доступа на запись (модификатор writes), то внутри функции или реакции не допускается использование этого выражения до ключевого слова post.
- Если для выражения указано ограничение доступа на изменение (модификатор updates), то внутри функции или реакции до ключевого слова post выражение имеет пре-значение — значение до взаимодействия с тестируемой системой, описываемого данной спецификационной функцией, или значение до возникновения данной реакции, после ключевого слова post выражение имеет

пост-значение — значение после взаимодействия с тестируемой системой или после возникновения реакции, пре-значение выражения после ключевого слова post доступно через оператор SeC `@` (см. раздел Превыражения).

- Если для выражения указано ограничение доступа на чтение (модификатор reads), то внутри функции или реакции его значение доступно везде и не меняется.

Пример

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

Псевдонимы

Назначение

Псевдонимы предназначены для упрощения доступа внутри спецификационных функций и отложенных реакций к выражениям, на которые описываются ограничения доступа.

Описание

Псевдонимы объявляются простым присваиванием в ограничениях доступа. Внутри функции или реакции псевдонимы используются аналогично локальным переменным.

Синтаксис

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

Семантические ограничения

- Идентификатор псевдонима не должен совпадать с идентификаторами параметров и должен быть уникальным в пределах ограничений доступа спецификационной функции или отложенной реакции.

Пример

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

Предусловия

Назначение

Предусловие спецификационной функции служит для выделения ситуаций, в которых определено поведение тестируемой системы при взаимодействиях с ней через часть интерфейса, описываемого данной функцией. Во время тестирования предусловие проверяется всякий раз, когда вызывается соответствующая функция тестируемой системы. Если предусловие нарушено, то это значит, что тест составлен некорректно.

Предусловие отложенной реакции описывает условия, при которых возможно возникновение данной реакции. Во время тестирования предусловие реакции проверяется всякий раз, когда она возникает. Если предусловие нарушается, то это значит, что обнаружено несоответствие поведения системы спецификации.

Описание

```
pre
{
  ...
  return значение_булевского_типа;
  ...
}
```

Предусловие на языке SeC — это набор инструкций, который синтаксически и семантически эквивалентен телу функции, имеющей те же параметры, что и спецификационная функция (у реакций параметры всегда отсутствуют) и возвращающей результат типа bool. Этот набор инструкций заключается в фигурные скобки и помечается ключевым словом pre.

Синтаксис

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

Семантические ограничения

- В спецификационной функции или отложенной реакции может быть не больше одного предусловия, которое должно быть определено до критериев покрытия (если они определяются в функции) и постусловия.
- Предусловие может быть опущено, отсутствие предусловия эквивалентно наличию предусловия, всегда возвращающего true.
- Предусловие не должно иметь побочных эффектов, то есть оно не должно изменять значения глобальных переменных и данных, переданных через указатели.
- Набор инструкций в предусловии должен быть синтаксически и семантически эквивалентно телу функции, имеющей ту же сигнатуру, что и спецификационная функция или отложенная реакция, частью определения которой оно является, и возвращающей результат типа bool.
- В предусловиях нельзя использовать выражения с ограничением доступа на запись, то есть описанные в ограничениях доступа со спецификатором writes.

Пример

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

Критерии покрытия

Назначение

Критерии покрытия предназначены для описания критериев покрытия требований. Каждый критерий покрытия разбивает на ветви функциональности поведение тестируемой системы при взаимодействии с ней через часть интерфейса, описываемого спецификационной функцией, частью определения которой он является. При выполнении тестирования критерии покрытия используются для оценки достигнутой полноты тестирования.

Описание

```
coverage имя
{
  ...
  return {идентификатор, строковый_литерал};
  ...
}
```

Критерий покрытия на языке SeC — это набор инструкций, синтаксически и семантически эквивалентный телу функции, имеющей те же параметры, что и спецификационная функция и возвращающей специальную конструкцию аналогичную конструкции инициализации в декларации переменной структурного типа из двух полей — заключенные в фигурные скобки идентификатор и строковый литерал, разделенные запятой. Этот набор инструкций заключается в фигурные скобки, помечается ключевым словом coverage и именуется.

При необходимости повторных вычислений выражений, задающих разбиение на ветви функциональности, в постусловии или в критериях покрытия, определяемых после данного критерия покрытия, могут использоваться конструкции coverage(*имя_покрытия*), значением которой является идентификатор покрываемой ветви функциональности указанного критерия покрытия. Конструкция может использоваться в условных операторах if-else и в операторах-переключателях switch языка C.

Синтаксис

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

Семантические ограничения

- В спецификационной функции может быть несколько критериев покрытия следующих друг за другом.
- Имена разных критериев покрытия должны быть уникальны внутри спецификационной функции.
- Критерии покрытия должны быть определены после предусловия (если оно не опущено) и до постусловия.

- Критерии покрытия могут быть опущены, их отсутствие эквивалентно наличию одного критерия покрытия с единственной ветвью функциональности.
- Набор инструкций в критерии покрытия должен быть синтаксически и семантически эквивалентен телу функции, имеющей ту же сигнатуру, что и спецификационная функция, частью определения которой он является, и возвращающей специальную конструкцию аналогичную конструкции инициализации в декларации переменной структурного типа из двух полей — заключенные в фигурные скобки идентификатор и строковый литерал, разделенные запятой, один и тот же идентификатор или строковый литерал не может использоваться в несовпадающих парах в одном и том же критерии покрытия.
- Критерий покрытия не должен иметь побочных эффектов, то есть он не должен изменять значения глобальных переменных и данных, переданных через указатели.
- В критериях покрытия нельзя использовать выражения с ограничением доступа на запись, то есть описанные в ограничениях доступа со спецификатором writes.
- Любой набор допустимых значений параметров и глобальных переменных должен соответствовать одной из ветвей функциональности, определенной в критерии покрытия.

Пример

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

Постусловия

Назначение

Постусловие спецификационной функции служит для описания ограничений, которым должны удовлетворять результаты работы тестируемой системы при взаимодействиях с ней через часть интерфейса, описываемого данной функцией. Во время тестирования постусловие проверяется всякий раз после работы тестируемой системы при соответствующих взаимодействиях, и, если постусловие нарушено, значит обнаружено несоответствие поведения системы спецификации.

Постусловие отложенной реакции описывает ограничения, которым должны удовлетворять значения самой реакции и глобальных переменных после возникновения данной реакции. Если во время тестирования после возникновения реакции и работы ее медиатора постусловие нарушено, значит обнаружено несоответствие поведения системы спецификации.

Описание

```
post
{
  ...
  return значение_булевского_типа;
  ...
}
```

Постусловие на языке SeC — это набор инструкций, который синтаксически и семантически эквивалентен телу функции, имеющей те же параметры, что и спецификационная функция (у реакций параметры всегда отсутствуют) и возвращающей результат типа bool. Этот набор инструкций заключается в фигурные скобки и помечается ключевым словом post.

Синтаксис

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

Семантические ограничения

- В спецификационной функции или отложенной реакции всегда должно быть ровно одно постусловие.
- Постусловие определяется после предусловия (если оно не опущено) и после последнего критерия покрытия (если критерии покрытия определяются).
- Набор инструкций в постусловие должен быть синтаксически и семантически эквивалентен телу функции, имеющей те же параметры, что и спецификационная функция или реакция (у реакций параметры всегда отсутствуют), частью определения которой оно является, и возвращающей результат типа bool.
- Постусловие не должно иметь побочных эффектов, то есть оно не должно изменять значения глобальных переменных и данных, переданных через указатели.
- В постусловии и после него в коде функции или реакции можно использовать следующие дополнительные конструкции:

- Превыражения с оператором `@` (см. Превыражения),
- Идентификатор, совпадающим с именем отложенной реакции или спецификационной функции (только для функций и реакций, имеющих тип результата не void), через который можно получить доступ к значению возникшей отложенной реакциии или результата, возвращаемого спецификационной функцией,
- Псевдопеременная timestamp, имеющая тип TimeInterval, которая содержит временные метки начала и конца исполнения вызова реализации медиатора данной спецификационной функции или временные метки указанные при регистрации данной отложенной реакции.

Пример

```
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 возвращает 0, при равенстве параметров */
    else return (   0
                 == compare(stck,
                            subList_List(@stck,1,size_List(@stck))
                           )
                 && i == value_Integer(get_List(@stck, 0))
                 && pop_spec
                );
  }
}
```

Превыражения

Назначение

Превыражения используются при спецификации совместных ограничений на состояния тестируемой системы до и после тестового воздействия или до и после возникновения отложенной реакции.

Описание

```
@выражение
```

В спецификационной функции вхождение ключевого слова post трактуется как тестовое воздействие. В отложенной реакции вхождение ключевого слова post трактуется как возникновение отложенной реакции. Все инструкции, написанные до слова post по потоку управления, выполняются до тестового воздействия или возникновения реакции. Инструкции после слова post выполняются после тестового воздействия или возникновения реакции.

Исключения составляют превыражения — выражения, помеченные при помощи префиксного оператора @. Внутри превыражений не могут использоваться идентификаторы объектов, доступных только на запись.

При записи превыражений следует руководствоваться следующими правилами:

- Приоритет оператора @ такой же как у других унарных операторов
- Вычислимость превыражения должна быть обеспечена непосредственно перед ключевым словом post по потоку управления, в котором используется превыражение, инструкциями, находящимся до него по потоку управления, в частности, внутри превыражения можно использовать идентификаторы только тех локальных переменных, которые объявлены до слова post по потоку управления

Синтаксис

```
"@" 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
              ;
```

Семантические ограничения

- Оператор `@` может использоваться только после ключевого слова post по потоку управления.
- Превыражение должно быть вычислимо до постусловия по потоку управления, в котором оно используется.

Пример

```
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;
  }
}
```

Медиаторы

Медиаторы предназначены для связывания спецификации с реализацией тестируемой системы или со спецификацией другого уровня абстрактности. В дальнейшем для удобства изложения под тестируемой системой будем понимать либо непосредственно реализацию тестируемой системы, либо ее спецификацию другого уровня абстрактности.

Медиаторы решают следующие задачи: преобразование модельного представления тестового воздействия в реализационное представление и преобразование реализационного представления реакции в модельное представление, а также синхронизацию состояния спецификационной модели данных с состоянием тестируемой системы. Медиаторы отложенных реакций реализуют только синхронизацию состояния спецификационной модели данных с состоянием тестируемой системы.

В SeC медиаторы реализуются как медиаторные функции и сборщики отложенных реакций.

Медиаторные функции

Назначение

Медиаторные функции предназначены для реализации медиаторов. Каждая медиаторная функция связывает спецификационную функцию или отложенную реакцию с частью реализации тестируемой системы или ее спецификации другого уровня абстрактности.

Описание

```
mediator медиатор for сигнатура ограничения_доступа
{
  дополнительный_код
  call { ... }
  state { ... }
  дополнительный_код
}
```

Медиаторные функции помечаются ключевыми словами SeC mediator for, между которыми должен содержаться уникальный идентификатор — имя медиаторной функции. Каждая медиаторная функция соответствует некоторой спецификационной функции или отложенной реакции, сигнатура и ограничения доступа которой должны указываться в предварительных декларациях и определении медиаторной функции.

Медиаторная функция может содержать:

- Блок воздействия, помеченный ключевым словом **`call`**, в котором реализуется поведение, описанное в соответствующей спецификационной функции, посредством оказания воздействий на тестируемую систему.
- Блок синхронизации, помеченный ключевым словом **`state`**, в котором реализуется синхронизация состояния спецификационной модели данных с состоянием тестируемой системы после оказанного воздействия или возникновения отложенной реакции.
- Дополнительный код до первого именованного блока и после последнего.

Синтаксис

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

Семантические ограничения

- Имена медиаторных функций входят в то же пространство имен, что и имена обычных функций языка C.
- Медиаторная функция должна быть определена ровно один раз среди всех единиц трансляции (translation_unit), собираемых в одну систему.

- Спецификационная функция или отложенная реакция, для которой определяется медиаторная функция, должна быть декларирована до определения или декларации медиаторной функции.
- Сигнатура и ограничения доступа спецификационной функции или отложенной реакции, для которой описывается медиатор, должны присутствовать среди спецификаторов (declaration_specifiers) и в декларатоpе (declarator) всех предварительных декларациях и определении медиаторной функции.
- В медиаторе спецификационной функции должен присутствовать блок воздействия, блок синхронизации может отсутствовать.
- Медиатор отложенной реакции состоит из обязательного блока синхронизации, блок воздействия в медиаторе реакции не допускается.
- Если в медиаторе есть и блок вызова, и блок синхронизации, то блок синхронизации должен следовать сразу после блока вызова.
- Дополнительный код допускается до первого именованного блока и после последнего.

Пример

```
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]))
        );
  }
}
```

Блоки воздействия

Назначение

Блоки воздействия предназначены для реализации посредством оказания воздействий на тестируемую систему поведения, описанного в спецификационных функциях, частью определения медиаторов которых они являются.

Описание

```
call
{
  ...
  return значение;
  ...
}
```

Блок воздействия на языке SeC — это набор инструкций синтаксически и семантически эквивалентный телу функции, имеющей в точности ту же сигнатуру — порядок параметров и их типы и тип возвращаемого результата, что и спецификационная функция, для которой определяется медиатор. Этот набор инструкций заключается в фигурные скобки и помечается ключевым словом call.

Синтаксис

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

Семантические ограничения

- Блок воздействия является обязательным блоком медиаторов спецификационных функций.
- В медиаторах отложенных реакций блок воздействия не допускается.
- Набор инструкций в блоке воздействия должен быть синтаксически и семантически эквивалентен телу функции, имеющей в точности ту же сигнатуру — порядок параметров и их типы и тип возвращаемого результата, что и спецификационная функция, для которой определяется медиатор.

Пример

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

Блоки синхронизации

Назначение

Блоки синхронизации предназначены для реализации синхронизации состояния спецификационной модели данных с состоянием тестируемой системы после оказанных воздействий на тестируемую систему или возникновения отложенных реакций.

Описание

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

Блок синхронизации на языке SeC — это набор инструкций синтаксически и семантически эквивалентный телу функции без возвращаемого результата, сигнатура которой в части параметров совпадает с сигнатурой спецификационной функции или отложенной реакции, для которой определяется медиатор. Этот набор инструкций заключается в фигурные скобки и помечается ключевым словом **`state`**.

Синтаксис

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

Семантические ограничения

- Набор инструкций в блоке синхронизации должен быть синтаксически и семантически эквивалентен телу функции без возвращаемого результата, сигнатура которой в части параметров совпадает с сигнатурой спецификационной функции или отложенной реакции, для которой определяется медиатор.
- В блоке синхронизации медиатора реакции или функции с типом результата не void можно получать доступ к значению возникшей отложенной реакции или результата, возвращаемого блоком воздействия того же самого медиатора, через идентификатор, совпадающим с именем отложенной реакции или спецификационной функции .
- В блоке синхронизации допускается использовать псевдопеременную timestamp, имеющую тип TimeInterval, которая содержит временные метки начала и конца исполнения блока воздействия того же самого медиатора или временные метки, указанные при регистрации отложенной реакции.

Пример

```
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]))
          );
    }
  }
```

Тестовые сценарии

Тестовые сценарии определяют исходные данные для построения тестов. Каждый тест представляет собой последовательность тестовых воздействий, выполнение которой обеспечивает решение некоторой задачи тестирования. Обычно в качестве такой задачи выступает тестирование поведения системы при воздействиях на нее через некоторый набор интерфейсных функций реализации до достижения заданного уровня покрытия в соответствии с критериями покрытия спецификации.

Тестовый сценарий

Назначение

Тестовый сценарий предназначен для построения теста, на основе указанного механизма построения тестов и необходимых для этого данных.

Описание

Декларация:

```
scenario тип идентификатор;
```

Определение:

```
scenario тип идентификатор = инициализатор;
```

Вызов:

```
идентификатор( argc, argv );
```

В языке SeC определение тестового сценария аналогично определению глобальной переменной, среди спецификаторов которой встречается ключевое слово SeC scenario.

Тип тестового сценария задается идентификатором, определенным с помощью конструкции typedef, и определяет механизм построения теста.

Каждому механизму построения теста соответствует:

- тип данных, необходимых для построения теста с помощью данного механизма, который является базовым типом в конструкции typedef, определяющей идентификатор типа сценария,
- функция запуска теста, построенного с помощью данного механизма.

При определении тестовый сценарий инициализируется значением типа, соответствующего данному механизму построения теста.

Для запуска тестового сценария используется конструкция вызова функции, в которой вместо имени функции используется имя тестового сценария, а параметры аналогичны по своей семантике параметрам стандартной функции main(int argc, char** argv).

Конструкция вызова тестового сценария возвращает значение типа bool, равное true, если тест отработал корректно и в ходе тестирования никаких ошибок обнаружено не было, и false — в противном случае.

Синтаксис

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

Семантические ограничения

- Имена тестовых сценариев входят в то же пространство имен, что и имена обычных глобальных переменных языка C.

- Тестовый сценарий должен быть определен ровно один раз среди всех единиц трансляции (translation_unit), собираемых в одну систему.
- Не может быть определено локальных тестовых сценариев.
- Тип тестового сценария должен задаваться идентификатором, определенным с помощью конструкции typedef.
- Идентификатор типа тестового сценария является именем механизма построения теста.
- Для полного определения механизма построения теста, необходимо, чтобы была определена функция запуска теста, построенного на основе данного механизма, с сигнатурой

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

- Тип инициализатора должен быть совместим с типом тестового сценария.
- Если тип тестового сценария является структурой, то в инициализаторе допускается использовать синтаксис разыменователей стандарта C99, независимо от используемого стандарта языка C.
- Вызов тестового сценария аналогичен вызову функции с двумя параметрами: первый типа int и второй типа char**. При этом второй параметр должен указывать на массив строк, оканчивающихся нулем, последний и только последний элемент массива должен быть нулевым указателем, а значение первого параметра должно равняться длине массива без последнего элемента.
- Вызов тестового сценария возвращает значение типа bool.

Пример

Предварительная декларация:

```
scenario dfsm stack_scenario;
```

Определение:

```
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;
}
```

Сценарные функции

Назначение

Сценарные функции предназначены для описания последовательностей тестовых воздействий, которые выполняются в каждой тестовой ситуации, достигнутой во время тестирования. Так же они могут производить проверку корректности поведения вызываемых функций тестируемой системы.

Описание

```
scenario bool сценарная_функция ()
{
  ...
  return значение_булевского_типа;
  ...
}
```

В языке SeC сценарная функция представляется как функция без параметров, возвращающая булевский результат, помеченная ключевым словом scenario. Сценарные функции могут производить проверки, основываясь на результатах выполнения функций тестируемой системы. Если проверка показывает, что тестируемая система ведет себя корректно, сценарная функция должна возвратить true, иначе — false. Так как результаты проверок постусловий вызываемых спецификационных функций или возникающих отложенных реакций автоматически учитываются тестовой системой, в возвращаемом результате сценарной функции не требуется их учитывать.

Синтаксис

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

Семантические ограничения

- Имена сценарных функций имена входят в то же пространство имен, что и имена обычных функций языка C
- Сценарная функция должна быть определена ровно один раз среди всех единиц трансляции (translation_unit), собираемых в одну систему
- Сценарные функции не должны иметь параметров и должны возвращать булевский результат
- Сценарные функции нельзя вызывать
- В сценарных функциях допускается использование следующих конструкций:
  - Операторов итерации (см. Операторы итерации),
  - Переменных состояния (см. Переменные состояния).

Пример

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

Операторы итерации

Назначение

Итераторы предназначены для записи перебора тестовых воздействий в сценарных функциях.

Описание

```
iterate                    (                                    выражение_1
         ;                                                      выражение_2
         ;                                                      выражение_3
         ;                                                      выражение_4
       )
```

Синтаксис итератора похож на синтаксис цикла **`for(;;)`** языка C. Итератор начинается с ключевого слова **`iterate`**, после которого в скобках через точку с запятой указаны:

- Декларация и инициализация итерационной переменной
- Условие продолжения итерации
- Способ итерации
- Условие фильтрации

Все указанные части, за исключением первой, являются необязательными и могут быть опущены. При этом следует учитывать, что при наличие фильтрации и отсутствии способа итерации может произойти зацикливание. Декларация заканчивается телом итератора.

Оператор итерации:

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

в некотором смысле подобен циклу:

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

Итерационные переменные не являются локальными, они являются частным случаем переменных состояния (см. Переменные состояния). Их значения запоминаются в специальной структуре данных, связанной с текущим обобщенным состоянием модели. Эти значения становятся доступными как только модель снова окажется в этом обобщенном состоянии.

Синтаксис

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

```
                    ")"
                    statement
                ;
```

Семантические ограничения

- Итераторы могут использоваться только в сценарных функциях.
- Выражения, задающие условия продолжения итерации или условия фильтрации, если они присутствуют в итераторе, должны иметь тип bool.
- Переменная, объявленная в стартовой декларации итератора, не должна быть неполного или локального типа, и должна быть инициализирована.

Пример

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

Переменные состояния

Назначение

Переменные состояния предназначены для хранения данных, связанных с обобщенным состоянием модели. Значения таких переменных становятся доступными, как только модель оказывается в этом обобщенном состоянии снова.

Описание

```
stable тип переменная = значение;
```

Декларация переменных состояния начинается с модификатора stable, после которого следует декларация локальных переменных. Код вида:

```
operator_1;
stable int i = 1;
operator_2;
```

эквивалентен следующему:

```
operator_1;
iterate(int i = 1;false;;)
{
    operator_2;
}
```

Синтаксис

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

Семантические ограничения

- Переменные состояния могут использоваться только в сценарных функциях.
- Переменные состояния не должны быть неполного или локального типа, и должны быть инициализированы при объявлении.

Пример

```
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;
}
```

Библиотека поддержки тестовой системы CTesK

Инструмент CTesK включает в себя библиотеку поддержки разрабатываемых тестов. Эта библиотека предоставляет интерфейс для организации взаимодействия с тестовой системой, а также ряд вспомогательных типов данных и функций. Заголовочные файлы библиотеки располагаются в каталоге include дистрибутива CTesK.

Данный раздел описывает часть интерфейса библиотеки, предназначенную для непосредственного использования разработчиками тестов. Другая часть интерфейса, определенная в заголовочных файлах, предназначена только для использования автоматически сгенерированными компонентами тестовой системы CTesK.

Базовые сервисы тестовой системы

Основные сервисы, предоставляемые тестовой системой, используются с помощью конструкций спецификационного расширения языка C.

Базовые сервисы тестовой системы CTesK, включенные в библиотеку поддержки, состоят из набора типов данных и функций, определяющих модель времени тестовой системы, и небольшого числа системных функций языка SeC.

Системные функции

Системными функциями языка SeC являются следующие функции:

- setBadVerdict
- assertion

setBadVerdict

Функция setBadVerdict устанавливает отрицательный вердикт текущего вызова медиатора.

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

Параметры

msg

Комментарий, поясняющий причины отрицательного вердикта медиатора. Этот комментарий попадает в трассу и используется только для упрощения анализа результатов тестирования.

Параметр может принимать нулевое значение. В этом случае никакой комментарий в трассу не попадет.

Описание

Функция setBadVerdict устанавливает отрицательный вердикт текущего вызова медиатора. Если медиатор по каким-либо причинам не может выполнить свою задачу, то он должен оповестить об этом тестовую систему. Для этого в ходе работы медиатора должна быть вызвана функция setBadVerdict.

Функция setBadVerdict может быть вызвана несколько раз в течении работы одного медиатора.

Если функция setBadVerdict была вызвана вне вызова медиатора, то комментарий попадает в трассу в качестве *пользовательского сообщения* без какого-либо другого побочного эффекта.

Дополнительная информация

Заголовочный файл: ts/ts.h

Библиотека: ts

Смотрите также

Системные функции

Пример

```
/*
 * Пример использования функции setBadVerdict.
 */


/*
 * Реализация содержит переменную состояния errMsg,
 * которая может принимать ограниченное число значений.
 */
const char* errMsg;

/*
 * Переменная errMsg моделируется переменной errno
 * перечислимого типа.
 */
enum ErrorKind { NoError, ErrorKind1, ErrorKind2 } errno;

/*
 * Функция updateSystemState синхронизирует значение модельной
 * переменной errno со зачением переменной реализации errMsg.
```

```
 */
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

Функция assertion проверяет выполнение предполагаемого ограничения и, если оно не выполнено, то завершает работу приложения.

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

Параметры

expr

Выражение, значение которого предполагается отличным от 0. Если это предположение нарушено, то работа приложения завершается.

format

Строка, определяющая формат сообщения о нарушении предположения. Сообщение формируется на основании строки формата и дополнительных параметров, семантика которых совпадает с семантикой функции printf из стандартной библиотеки языка C.

Описание

Функция assertion проверяет, что выполняется предполагаемое ограничение. Если это ограничение не выполнено, то работа приложения завершается.

При этом формируется сообщение о нарушении предположения, которое помещается либо в трассу, если проверка происходит во время работы тестового сценария, либо в stderr, в противном случае. Затем трасса корректно закрывается и приложение завершает свою работу посредством вызова системной функции exit с параметром 1.

Любое аварийное завершение работы тестового сценария должно выполняться посредством вызова функции assertion. В противном случае целостность трассы может быть нарушена.

Дополнительная информация

Заголовочный файл: utils/assertion.h

Библиотека: utils

Смотрите также

Системные функции, Сервисы трассировки

Модель времени

Тестовая система CTesK поддерживает три режима работы со временем:

- Без учета времени,
- Линейная модель времени,
- Распределенная модель времени.

Для характеризации моментов времени в тестовой системе используются *временные метки*. Временная метка является абстрактной величиной, которая по воли разработчика может быть привязана к реальному времени, а может использоваться только для упорядочивания определенных моментов времени.

Каждая временная метка принадлежит некоторой *системе отсчета времени*. Все временные метки принадлежащие одной системе отсчета линейно упорядочены между собой. Однако порядок между временными метками различных систем отсчета не определен.

При работе в режиме линейной модели времени считается, что все временные метки принадлежат одной единственной системе отсчета времени. Поэтому все временные метки являются линейно упорядоченными.

При работе в режиме распределенной модели времени временные метки могут принадлежать различным системам отсчета. Этот режим является наиболее общим, однако алгоритмы работы тестовой системы с распределенными временными метками наименее эффективны.

Управление моделью времени тестовой системы CTesK осуществляется посредством следующих функций:

setTSTimeModel

getTSTimeModel

Временные метки определяются с помощью следующих типов данных, констант и функций:

Типы

LinearTimeMark

TimeFrameOfReferenceID

TimeMark

TimeInterval

Константы

systemTimeFrameOfReferenceID

minTimeMark

maxTimeMark

Функции

getTimeFrameOfReferenceID

setSystemTimeFrameOfReferenceName

createTimeMark

createDistributedTimeMark

createTimeInterval

Для определения временной метки соответствующей текущему моменту времени используются следующие функции:

getCurrentTimeMark

setDefaultCurrentTimeMarkFunction

TSTimeModel

Перечислимый тип TSTimeModel определяет возможные режимы работы со временем тестовой системы CTesK.

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

Описание

При использовании механизма тестирования dfsm, тестовая система по умолчанию работает

- в режиме без учета времени, если свойство обходчика «поддержка отложенных реакций» отключено,
- в режиме линейного времени, если свойство обходчика «поддержка отложенных реакций» включено.

Режим работы со временем может быть изменен в функции инициализации сценария.

Дополнительная информация

Заголовочный файл: ts/timemark.h

Библиотека: ts

Смотрите также

Модель времени, setTSTimeModel, getTSTimeModel

setTSTimeModel

Функция setTSTimeModel изменяет режим работы со временем тестовой системы CTesK.

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

Параметры

time_model

Новый режим работы со временем, в котором будет работать тестовая система.

Возвращаемое значение

Предыдущий режим работы со временем тестовой системы.

Описание

При использовании механизма тестирования dfsm, тестовая система по умолчанию работает

- в режиме без учета времени, если хотя бы одно из полей saveModelState, restoreModelState или isStationaryState тестового сценария не определено или инициализировано нулевым указателем
- в режиме линейного времени, если поля saveModelState, restoreModelState и isStationaryState тестового сценария инициализированы ненулевыми указателями

Режим работы со временем может быть изменен в функции инициализации сценария.

Дополнительная информация

Заголовочный файл: ts/timemark.h

Библиотека: ts

Смотрите также

Модель времени, TSTimeModel, setTSTimeModel, Механизм построения тестов dfsm

getTSTimeModel

Функция getTSTimeModel возвращает текущий режим работы со временем тестовой системы CTesK.

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

Возвращаемое значение

Текущий режим работы со временем тестовой системы CTesK.

Дополнительная информация

Заголовочный файл: ts/timemark.h

Библиотека: ts

Смотрите также

Модель времени, TSTimeModel, getTSTimeModel

LinearTimeMark

Тип LinearTimeMark используется для идентификации временных меток внутри определенной системы отсчета времени.

```
typedef unsigned long LinearTimeMark;
```

Описание

Значения этого типа могут отражать любые характеристики моментов времени внутри определенной системы отсчета времени. Например, число секунд (или миллисекунд) прошедших с определенного момента времени.

Если одно значение типа LinearTimeMark больше другого значения, то считается, что оно соответствует моменту времени гарантировано более позднему, чем другое. Если два значения равны, то считается, что о взаимном расположении соответствующих им моментов времени ничего не известно. Они могут совпадать, а могут и быть различными.

Дополнительная информация

Заголовочный файл: ts/timemark.h

Библиотека: ts

Смотрите также

Модель времени, TimeMark

TimeFrameOfReferenceID

Тип TimeFrameOfReferenceID определяет идентификаторы систем отсчета времени.

```
typedef int TimeFrameOfReferenceID;
```

Описание

Тестовая система работает в выделенной системе отсчета времени, которая имеет предопределенный идентификатор systemTimeFrameOfReferenceID. В режиме линейного времени разрешается использовать только этот идентификатор системы отсчета.

Другие идентификаторы строятся в режиме распределенного времени функцией getTimeFrameOfReferenceID. Эта функция по имени системы отсчета возвращает ее идентификатор. При двух обращениях к функции с одним и тем же именем будет получен один и тот же идентификатор. Если вместо имени передать нулевой указатель, то функция вернет уникальный идентификатор системы отсчета, который гарантировано не будет использован ею дважды.

Обычно каждому компьютеру соответствует собственная система отсчета времени. В этом случае, системы отсчета можно идентифицировать сетевым именем соответствующего им компьютера.

Для единообразия работы с идентификаторами систем отсчета, предопределенному идентификатору также можно присвоить символическое имя. Для этого предназначена функция setSystemTimeFrameOfReferenceName.

Если системе отсчета, в которой работает тестовая система, было присвоено имя, то любой вызов функции getTimeFrameOfReferenceID с этим именем вернет systemTimeFrameOfReferenceID.

Дополнительная информация

Заголовочный файл: ts/timemark.h

Библиотека: ts

Смотрите также

Модель времени, systemTimeFrameOfReferenceID, getTimeFrameOfReferenceID, setSystemTimeFrameOfReferenceName

TimeMark

Структура TimeMark определяет тип временной метки, используемой в тестовой системе.

```
typedef struct TimeMark TimeMark;

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

Описание

Временная метка характеризуется идентификатором системы отсчета времени и значением идентифицирующим момент времени внутри данной системы отсчета.

Считается, что одна временная метка меньше другой только в том случае, когда обе метки принадлежат одной системе отсчета и значение поля timemark первой метки меньше значения этого поля другой.

Дополнительная информация

Заголовочный файл: ts/timemark.h

Библиотека: ts

Смотрите также

Модель времени, TimeFrameOfReferenceID, LinearTimeMark, minTimeMark, maxTimeMark, createTimeMark, createDistributedTimeMark

TimeInterval

Тип TimeInterval определяет интервал времени между двумя временными метками.

```
typedef struct TimeInterval TimeInterval;

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

Описание

Тип TimeInterval определяет интервал времени между двумя временными метками minMark и maxMark. При этом требуется, чтобы эти метки принадлежали одной системе отсчета времени и значение первой не было больше значения второй. Граничные временные метки включаются внутрь интервала.

Для обозначения минимально и максимально возможных временных меток необходимо использовать специальные константы minTimeMark и maxTimeMark соответственно.

Дополнительная информация

Заголовочный файл: ts/timemark.h

Библиотека: ts

Смотрите также

Модель времени, TimeMark, createTimeInterval, minTimeMark, maxTimeMark

systemTimeFrameOfReferenceID

Константа systemTimeFrameOfReferenceID определяет идентификатор системы отсчета времени, в которой работает тестовая система.

```
extern const TimeFrameOfReferenceID systemTimeFrameOfReferenceID;
```

Описание

Идентификатору системы отсчета времени, в которой работает тестовая система, systemTimeFrameOfReferenceID можно присвоить символическое имя с помощью функции setSystemTimeFrameOfReferenceName. В этом случае любой вызов функции getTimeFrameOfReferenceID с этим символическим именем будет возвращать systemTimeFrameOfReferenceID.

Дополнительная информация

Заголовочный файл: ts/timemark.h

Библиотека: ts

Смотрите также

Модель времени, TimeFrameOfReferenceID, setSystemTimeFrameOfReferenceName

minTimeMark

Константа minTimeMark равняется временной метке гарантировано меньшей любой другой временной метки из любой системы отсчета.

```
extern const TimeMark minTimeMark;
```

Описание

Константа minTimeMark является выделенным значением типа TimeMark, которое гарантировано меньше любого другого значения этого типа независимо от его системы отсчета. Константа minTimeMark равняется только себе самой.

Дополнительная информация

Заголовочный файл: ts/timemark.h

Библиотека: ts

Смотрите также

Модель времени, TimeMark, maxTimeMark

maxTimeMark

Константа maxTimeMark равняется временной метке гарантировано большей любой другой временной метки из любой системы отсчета.

```
extern const TimeMark maxTimeMark;
```

Описание

Константа maxTimeMark является выделенным значением типа TimeMark, которое гарантировано больше любого другого значения этого типа независимо от его системы отсчета. Константа maxTimeMark равняется только себе самой.

Дополнительная информация

Заголовочный файл: ts/timemark.h

Библиотека: ts

Смотрите также

Модель времени, TimeMark, minTimeMark

getTimeFrameOfReferenceID

Функция getTimeFrameOfReferenceID возвращает идентификатор системы отсчета, соответствующий указанному имени.

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

Параметры

name

Имя системы отсчета.

Параметр может быть нулевым указателем. В этом случае возвращается уникальный идентификатор системы отсчета, который гарантировано не будет возвращен данной функцией дважды.

Возвращаемое значение

Идентификатор системы отсчета с указанным именем. Функция гарантированно возвращает один и тот же идентификатор для любого числа запросов с заданным именем системы отсчета.

Описание

Функция getTimeFrameOfReferenceID возвращает идентификатор системы отсчета по ее имени. При двух обращениях к функции с одним и тем же именем будет получен один и тот же идентификатор. Если вместо имени передать нулевой указатель, то функция вернет уникальный идентификатор системы отсчета.

Функцию getTimeFrameOfReferenceID допускается использовать только в режиме работы с распределенным временем.

Обычно каждому компьютеру соответствует собственная система отсчета времени. В этом случае, системы отсчета можно идентифицировать сетевым именем соответствующего им компьютера.

Для единообразия работы с идентификаторами систем отсчета, идентификатору системы отсчета, в которой работает тестовая система, также можно присвоить символическое имя с помощью функции setSystemTimeFrameOfReferenceName.

Если системе отсчета, в которой работает тестовая система, было присвоено имя, то любой вызов функции getTimeFrameOfReferenceID с этим именем вернет systemTimeFrameOfReferenceID.

Дополнительная информация

Заголовочный файл: ts/timemark.h

Библиотека: ts

Смотрите также

Модель времени, TSTimeModel, TimeFrameOfReferenceID, setSystemTimeFrameOfReferenceName, systemTimeFrameOfReferenceID

setSystemTimeFrameOfReferenceName

Функция setSystemTimeFrameOfReferenceName устанавливает имя системы отсчета, в которой работает тестовая система.

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

Параметры

name

Имя системы отсчета, в которой работает тестовая система.

Параметр не должен быть нулевым указателем.

Возвращаемое значение

Функция возвращает false, если данное имя уже использовалось для идентификации системы отсчета отличной от системной, и true — иначе.

Описание

Функция setSystemTimeFrameOfReferenceName устанавливает имя системы отсчета, в которой работает тестовая система. После этого любой вызов функции getTimeFrameOfReferenceID с данным именем вернет systemTimeFrameOfReferenceID.

Система отсчета, в которой работает тестовая система, может иметь несколько имен одновременно.

Дополнительная информация

Заголовочный файл: ts/timemark.h

Библиотека: ts

Смотрите также

Модель времени, TSTimeModel, TimeFrameOfReferenceID, getTimeFrameOfReferenceID, systemTimeFrameOfReferenceID

createTimeMark

Функция createTimeMark создает временную метку в системе отсчета, в которой работает тестовая система.

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

Параметры

timemark

Метка, идентифицирующая момент времени внутри системы отсчета, в которой работает тестовая система.

Возвращаемое значение

Функция возвращает временную метку, принадлежащую системе отсчета времени, в которой работает тестовая система, и идентифицирующуюся внутренней меткой timemark.

Описание

Функция createTimeMark создает временную метку с идентификатором системы отсчета systemTimeFrameOfReferenceID и внутренней меткой timemark.

Дополнительная информация

Заголовочный файл: ts/timemark.h

Библиотека: ts

Смотрите также

Модель времени, LinearTimeMark, TimeMark, systemTimeFrameOfReferenceID, createDistributedTimeMark

createDistributedTimeMark

Функция createDistributedTimeMark создает временную метку в указанной системе отсчета.

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

Параметры

frame

Идентификатор системы отсчета, которой принадлежит создаваемая временная метка.

timemark

Метка идентифицирующая момент времени внутри системы отсчета frame.

Возвращаемое значение

Функция возвращает временную метку с идентификатором системы отсчета frame и внутренней меткой timemark.

Дополнительная информация

Заголовочный файл: ts/timemark.h

Библиотека: ts

Смотрите также

Модель времени, TimeFrameOfReferenceID, LinearTimeMark, TimeMark, createTimeMark

createTimeInterval

Функция createTimeInterval создает временной интервал с указанными границами.

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

Параметры

minMark

Временная метка нижней границы интервала.

maxMark

Временная метка верхней границы интервала.

Возвращаемое значение

Функция возвращает временной интервал с указанными границами.

Описание

Функция createTimeInterval создает интервал времени между двумя временными метками minMark и maxMark. При этом требуется, чтобы эти метки принадлежали одной системе отсчета времени и значение первой не было больше значения второй. Граничные временные метки включаются внутрь интервала.

Для обозначения минимально и максимально возможных временных меток необходимо использовать специальные константы minTimeMark и maxTimeMark соответственно.

Дополнительная информация

Заголовочный файл: ts/timemark.h

Библиотека: ts

Смотрите также

Модель времени, TimeInterval, TimeMark, createTimeMark, createDistributedTimeMark, minTimeMark, maxTimeMark

GetCurrentTimeMarkFuncType

Тип GetCurrentTimeMarkFuncType используется для установки функции, вычисляющей временную метку текущего момента времени.

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

Описание

Тип GetCurrentTimeMarkFuncType используется для установки функции, вычисляющей временную метку текущего момента времени.

Дополнительная информация

Заголовочный файл: ts/timemark.h

Библиотека: ts

Смотрите также

Модель времени, TimeMark, setDefaultCurrentTimeMarkFunction

getCurrentTimeMark

Функция getCurrentTimeMark возвращает временную метку, соответствующую текущему моменту времени внутри данного процесса.

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

Возвращаемое значение

Функция возвращает временную метку, соответствующую текущему моменту времени внутри данного процесса.

Описание

Функция возвращает временную метку, соответствующую текущему моменту времени внутри данного процесса. Данная функция используется тестовой системой для автоматического определения временного интервала, в котором выполняется вызов спецификационной функции.

По умолчанию, текущая временная метка принадлежит системе отсчета, в которой работает тестовая система. Метка внутри системы отсчета вычисляется как число секунд прошедших с полуночи (00:00:00) 1 января 1970 года, полученное путем вызова системной функции time.

Поведение функции может быть переопределено пользователем с помощью функции setDefaultCurrentTimeMarkFunction.

Дополнительная информация

Заголовочный файл: ts/timemark.h

Библиотека: ts

Смотрите также

Модель времени, TimeMark, createTimeMark, createDistributedTimeMark, setDefaultCurrentTimeMarkFunction, systemTimeFrameOfReferenceID

setDefaultCurrentTimeMarkFunction

Функция setDefaultCurrentTimeMarkFunction устанавливает пользовательскую функцию вычисления временной метки, соответствующей текущему моменту времени внутри данного процесса.

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

Параметры

new_func

Указатель на функцию, которая будет использоваться для вычисления временной метки, соответствующей текущему моменту времени внутри данного процесса.

Параметр не должен быть нулевым указателем.

Возвращаемое значение

Функция возвращает указатель на предыдущую функцию, вычислявшую текущую временную метку.

Описание

Функция setDefaultCurrentTimeMarkFunction устанавливает пользовательскую функцию вычисления временной метки, соответствующей текущему моменту времени внутри активного процесса. После этого функция getCurrentTimeMark будет возвращать временную метку, полученную путем вызова данной пользовательской функции.

Таким образом, пользовательская функция будет использоваться для автоматического вычисления интервала времени, соответствующего вызову каждой спецификационной функции.

Дополнительная информация

Заголовочный файл: ts/timemark.h

Библиотека: ts

Смотрите также

Модель времени, GetCurrentTimeMarkFuncType, getCurrentTimeMark

Стандартные механизмы построения тестов

В CTesK 2.2 Community Edition включены два механизма построения тестов dfsm и ndfsm. Благодаря своей гибкости они позволяют тестировать очень широкий класс программного обеспечения, начиная с простых систем без внутреннего состояния и кончая распределенными системами с асинхронным интерфейсом.

dfsm

Механизм построения тестов dfsm основывается на понятии обхода детерминированного конечного автомата. Конечный автомат, используемый для построения теста, задается в неявном виде. Пользователь определяет функцию вычисления текущего состояния сценария и набор тестовых воздействий.

Во время тестирования dfsm применяет тестовые воздействия, которые могут изменять состояние сценария. dfsm автоматически отслеживает все изменения состояния и строит конечный автомат, соответствующий процессу тестирования. Состояниями автомата являются все достижимые состояния сценария, а переходы автомата помечаются тестовыми воздействиями, которые их инициировали.

Механизм тестирования dfsm заканчивает тестирование только тогда, когда подаст все определенные пользователем тестовые воздействия во всех состояниях автомата достижимых из начального.

Чтобы достижение этой цели было возможно, необходимо выполнение следующих условий:

- **Конечность**

  Число состояний, достижимых из начального путем применения тестовых воздействий из заданного набора, должно быть конечным.

- **Детерминированность**

  Для любого состояния системы применение одного и того же тестового воздействия всегда должно переводить систему в одно и тоже состояние.

- **Сильная связность**

  Из любого состояния сценария достижимо любое другое состояние сценария путем применения последовательности тестовых воздействий.

Набор тестовых воздействий определяется с помощью сценарных функций (см. Сценарные функции).

Тип описания тестового сценария dfsm используется при инициализации тестового сценария, построенного на основе механизма dfsm. Этот тип является структурой со следующими полями:

init

finish

getState

actions

saveModelState

restoreModelState

isStationaryState

observeState

Обязательным для инициализации является только поле actions, которое содержит массив сценарных функций, определяющих тестовые воздействия.

Дополнительные параметры механизма тестирования dfsm могут быть настроены с помощью следующих функций:

setFinishMode

setDeferredReactionsMode

setWTime

setFindFirstSeriesOnly

setFindFirstSeriesOnlyBound

При запуске тестового сценария ему передается список параметров. Механизм тестирования dfsm имеет стандартные параметры, которые настраивают его работу. Все стандартные параметры должны следовать до пользовательских параметров. Механизм тестирования dfsm обрабатывает их и передает оставшиеся в функцию инициализации тестового сценария. В CTesK 2.2 Community Edition механизм тестирования dfsm поддерживает следующие стандартные параметры:

-t <file-name>

-tc

-tt

-nt

-uerr

-uend

--trace-accidental

--find-first-series-only

ndfsm

Механизм построения тестов ndfsm по сравнению с dfsm позволяет корректно работать на более широком классе автоматов, а именно на конечных атоматах, имеющих детерминированный сильносвязный полный остовный подавтомат.

- **Остовный подавтомат**

  Остовный подавтомат содержит все состояния сценария, достижимые в процессе тестирования.

- **Полный подавтомат**

  Для каждого состояния сценария и допустимого в нем тестового воздействия подавомат либо содержит все переходы из этим состояния, помеченные этим тестовым воздействием, либо не содержит таких переходов вовсе.

Набор тестовых воздействий определяется с помощью сценарных функций (см. Сценарные функции).

Тип описания тестового сценария ndfsm используется при инициализации тестового сценария, построенного на основе механизма ndfsm. Этот тип является структурой со следующими полями:

- init
- finish
- getState
- actions

Обязательным для инициализации является только поле actions, которое содержит массив сценарных функций, определяющих тестовые воздействия.

Дополнительные параметры механизма тестирования ndfsm могут быть настроены с помощью следующих функций:

- setFinishMode
- setDeferredReactionsMode
- setWTime
- setFindFirstSeriesOnly
- setFindFirstSeriesOnlyBound

При запуске тестового сценария ему передается список параметров. Механизм тестирования ndfsm имеет стандартные параметры, которые настраивают его работу. Все стандартные параметры должны следовать до пользовательских параметров. Механизм тестирования ndfsm обрабатывает их и передает оставшиеся в функцию инициализации тестового сценария. В CTesK 2.2 Community Edition механизм тестирования ndfsm поддерживает следующие стандартные параметры:

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

Типы и параметры механизмов построения тестов

Ниже описаны типы и параметры, используемые стандартными механизмами построения тестов dfsm и ndfsm.

PtrInit

Тип PtrInit определяет тип функции инициализации тестового сценария.

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

Описание

Функция инициализации получает на вход массив параметров, согласно семантике принятой для параметров argc, argv стандартной функции main, и возвращает булевскую величину, которая должна принимать значение true, если инициализация завершилась успешно, и false – в противном случае.

Дополнительная информация

Заголовочный файл: ts/engine.h

Библиотека: ts

Смотрите также

Механизм построения тестов dfsm, механизм построения тестов ndfsm, поле init

init

Поле init содержит указатель на функцию инициализации тестового сценария.

```
PtrInit init;
```

Описание

Функция инициализации получает на вход массив параметров, согласно семантике принятой для параметров argc, argv стандартной функции main. Она может использовать эти параметры для настройки тестового сценария.

Обычно функция инициализации выполняет следующие действия:

- инициализация тестируемой системы,
- инициализация спецификационной модели данных,
- инициализация данных сценария,
- установка медиаторов для спецификационных функций и реакций, используемых в данном тестовом сценарии.

Функция инициализации возвращает булевскую величину, которая должна принимать значение true, если инициализация завершилась успешно, и false – в противном случае. В последнем случае работа тестового сценария на этом завершится. Функция завершения сценария вызвана не будет.

Функция инициализация может содержать вызовы спецификационных функций, осуществляющих инициализацию тестируемой системы, спецификационной модели данных или данных сценария. При работе в режиме с отложенными реакциями механизм построения тестов осуществляет сериализацию всех поданных при вызове функции инициализации стимулов и всех полученных реакций.

Поле init может быть инициализировано нулевым указателем или не инициализировано вовсе. Это будет эквивалентно инициализации поля init функцией, не выполняющей никаких действий и возвращающей true.

Дополнительная информация

Заголовочный файл: ts/dfsm.h, ts/ndfsm.h

Библиотека: ts

Смотрите также

Механизм построения тестов dfsm, механизм построения тестов ndfsm, PtrInit, поле finish

PtrFinish

Тип PtrFinish определяет тип функции завершения тестового сценария.

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

Описание

Функция завершения сценария не имеет параметров и возвращаемого значения. Она предназначена для освобождения ресурсов после завершения работы сценария.

Дополнительная информация

Заголовочный файл: ts/engine.h

Библиотека: ts

Смотрите также

Механизм построения тестов dfsm, механизм построения тестов ndfsm, поле finish

finish

Поле finish содержит указатель на функцию завершения тестового сценария.

```
PtrFinish finish;
```

Описание

Функция завершения сценария не имеет параметров и возвращаемого значения. Она предназначена для освобождения ресурсов после завершения работы сценария.

Обычно функция завершения сценария выполняет следующие действия:

- освобождение ресурсов тестируемой системы, занятых во время работы тестового сценария,
- освобождение ресурсов спецификационной модели данных,
- освобождение ресурсов тестового сценария.

Функция завершения сценария может содержать вызовы спецификационных функций, осуществляющих освобождение ресурсов тестируемой системы, ресурсов спецификационной модели данных или ресурсов сценария. При работе в режиме с отложенными реакциями механизм построения тестов осуществляет сериализацию всех поданных при вызове функции завершения сценария стимулов и всех полученных реакций.

Поле finish может быть инициализировано нулевым указателем или не инициализировано вовсе. В этом случае, никаких действий по завершению сценария выполняться не будет.

Дополнительная информация

Заголовочный файл: ts/dfsm.h, ts/ndfsm.h

Библиотека: ts

Смотрите также

Механизм построения тестов dfsm, механизм построения тестов ndfsm, PtrFinish, поле init

PtrGetState

Тип PtrGetState определяет тип функции, вычисляющей текущее состояние тестового сценария.

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

Описание

Функция вычисления состояния сценария не имеет параметров и возвращает объект спецификационного типа.

Дополнительная информация

Заголовочный файл: ts/engine.h

Библиотека: ts

Смотрите также

Механизм построения тестов dfsm, механизм построения тестов ndfsm, поле getState

getState

Поле getState содержит указатель на функцию, вычисляющую текущее состояние тестового сценария.

```
PtrGetState getState;
```

Описание

Функция вычисления состояния сценария не имеет параметров и возвращает объект спецификационного типа.

При определении состояния сценария важно учитывать требование детерминированности механизма построения тестов dfsm или наличие детерминированного сильносвязного полного остовного подавтомата для механизма построения тестов ndfsm.

Поле getState может быть инициализировано нулевым указателем или не инициализировано вовсе. В этом случае механизм тестирования dfsm будет считать, что состояние сценария всегда одно и тоже.

Дополнительная информация

Заголовочный файл: ts/dfsm.h, ts/ndfsm.h

Библиотека: ts

Смотрите также

Механизм построения тестов dfsm, механизм построения тестов ndfsm, PtrGetState

actions

Поле actions содержит массив сценарных функций, завершающийся нулевым указателем.

```
ScenarioFunctionID actions[];
```

Описание

Поле actions содержит массив сценарных функций, завершающийся нулевым указателем (см. Сценарные функции).

Поле actions является обязательным для инициализации. Последнее значение массива должно быть нулевым указателем.

Дополнительная информация

Заголовочный файл: ts/dfsm.h, ts/ndfsm.h

Библиотека: ts

Смотрите также

Механизм построения тестов dfsm, механизм построения тестов ndfsm

PtrSaveModelState

Тип PtrSaveModelState определяет тип функции, возвращающей состояние спецификационной модели данных.

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

Описание

Функция сохранения состояния спецификационной модели данных не имеет параметров и возвращает объект спецификационного типа.

Дополнительная информация

Заголовочный файл: ts/engine.h

Библиотека: ts

Смотрите также

Механизм построения тестов dfsm, механизм построения тестов ndfsm, поле saveModelState

saveModelState

Поле saveModelState содержит указатель на функцию, возвращающую состояние спецификационной модели данных.

```
PtrSaveModelState saveModelState;
```

Описание

Функция сохранения состояния спецификационной модели данных не имеет параметров и возвращает объект спецификационного типа. Этот объект должен содержать все состояние спецификационной модели данных, чтобы функция восстановления состояния спецификационной модели данных могла полностью его восстановить по данному объекту.

Поле saveModelState может быть инициализировано нулевым указателем или не инициализировано вовсе. В этом случае тестирование систем с отложенными реакциями с помощью данного тестового сценария будет невозможно.

Дополнительная информация

Заголовочный файл: ts/dfsm.h, ts/ndfsm.h

Библиотека: ts

Смотрите также

Механизм построения тестов dfsm, механизм построения тестов ndfsm, PtrSaveModelState, поле restoreModelState

PtrRestoreModelState

Тип PtrRestoreModelState определяет тип функции, восстанавливающей состояние спецификационной модели данных.

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

Описание

Функция получает объект спецификационного типа и восстанавливает по нему состояние спецификационной модели данных. Функция не имеет возвращаемого значения.

Дополнительная информация

Заголовочный файл: ts/engine.h

Библиотека: ts

Смотрите также

Механизм построения тестов dfsm, механизм построения тестов ndfsm, поле restoreModelState

restoreModelState

Поле restoreModelState содержит указатель на функцию, восстанавливающую состояние спецификационной модели данных.

```
PtrRestoreModelState restoreModelState;
```

Описание

Функция восстановления состояния спецификационной модели данных получает объект спецификационного типа и восстанавливает по нему состояние спецификационной модели данных. Получаемый объект заведомо был построен ранее с помощью функции сохранения состояния спецификационной модели данных. Функция восстановления состояния не имеет возвращаемого значения.

Поле restoreModelState может быть инициализировано нулевым указателем или не инициализировано вовсе. В этом случае тестирование систем с отложенными реакциями с помощью данного тестового сценария будет невозможно.

Дополнительная информация

Заголовочный файл: ts/dfsm.h, ts/ndfsm.h

Библиотека: ts

Смотрите также

Механизм построения тестов dfsm, механизм построения тестов ndfsm, PtrRestoreModelState, поле saveModelState

PtrIsStationaryState

Тип PtrIsStationaryState определяет тип функции проверки стационарности модельного состояния.

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

Описание

Функция проверки стационарности модельного состояния не имеет параметров и возвращает значение типа bool.

Дополнительная информация

Заголовочный файл: ts/engine.h

Библиотека: ts

Смотрите также

Механизм построения тестов dfsm, механизм построения тестов ndfsm, поле isStationaryState

isStationaryState

Поле isStationaryState содержит указатель на функцию проверки стационарности модельного состояния.

```
PtrIsStationaryState isStationaryState;
```

Описание

Функция проверки стационарности модельного состояния не имеет параметров и возвращает true, если текущее модельное состояние является стационарным, и false – иначе.

Модельное состояние называется *стационарным*, если в этом состоянии целевая система, удовлетворяющая данной модели, не может инициировать взаимодействие.

Поле isStationaryState может быть инициализировано нулевым указателем или не инициализировано вовсе. В этом случае тестирование систем с асинхронным интерфейсом с помощью данного тестового сценария будет невозможно.

Дополнительная информация

Заголовочный файл: ts/dfsm.h, ts/ndfsm.h

Библиотека: ts

Смотрите также

Механизм построения тестов dfsm, механизм построения тестов ndfsm, PtrIsStationaryState

PtrObserveState

Тип PtrObserveState определяет тип функции, синхронизирующей модельное состояние с состоянием тестируемой системы по истечении времени стабилизации.

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

Описание

Функция синхронизации модельного состояния не имеет параметров и возвращаемого значения.

Дополнительная информация

Заголовочный файл: ts/engine.h

Библиотека: ts

Смотрите также

Механизм построения тестов dfsm, механизм построения тестов ndfsm, поле observeState

observeState

Поле observeState содержит указатель на функцию, синхронизирующую модельное состояние с состоянием тестируемой системы по истечении времени стабилизации.

```
PtrObserveState observeState;
```

Описание

Функция синхронизации модельного состояния не имеет параметров и возвращаемого значения. Она вызывается после того как тестовая система подаст очередное тестовое воздействие и подождет в течении времени стабилизации того, чтобы целевая система перешла в стационарное состояние. Во время вызова функция синхронизации может обратиться к одной или нескольким спецификационным функциям, имеющим доступ к состоянию тестируемой системы, но не изменяющие его. Взаимодействия, инициированные из функции синхронизации, будут учитываться во время сериализации наравне с другими взаимодействиями предшествовавшего ему тестового воздействия.

Поле observeState может быть инициализировано нулевым указателем или не инициализировано вовсе. В этом случае никакой синхронизации в стационарном состоянии проводиться не будет.

Дополнительная информация

Заголовочный файл: ts/dfsm.h, ts/ndfsm.h

Библиотека: ts

Смотрите также

Механизм построения тестов dfsm, механизм построения тестов ndfsm, PtrObserveState

FinishMode

Перечислимый тип FinishMode определяет возможные режимы завершения работы механизма тестирования dfsm.

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

Описание

Перечислимый тип FinishMode определяет возможные режимы завершения работы механизма тестирования dfsm.

Первый режим — UNTIL_ERROR означает, что тестирование завершается сразу после обнаружения первой ошибки.

Второй режим — UNTIL_END означает, что тестирование после обнаружения не критичной ошибки будет продолжено и завершится только после достижения стандартного условия завершения тестирования.

По умолчания, механизм тестирования dfsm работает в режиме до первой ошибки.

Дополнительная информация

Заголовочный файл: ts/engine.h

Библиотека: ts

Смотрите также

Механизм построения тестов dfsm, механизм построения тестов ndfsm, setFinishMode, getFinishMode, стандартный параметр «–uerr», стандартный параметр «–uend»

setFinishMode

Функция setFinishMode устанавливает значение режима завершения работы тестового сценария.

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

Параметры

finish_mode

Новый режим завершения работы тестового сценария.

Возвращаемое значение

Предыдущий режим завершения работы тестогово сценария.

Описание

Функция setFinishMode устанавливает значение режима завершения работы механизма тестирования dfsm. По умолчания, механизмы тестирования dfsm и ndfsm работают в режиме до первой ошибки.

Изменять режим работы разрешается в любой момент работы тестовой системы. При этом изменение режима завершения работы повлияет только на вновь обнаруживаемые ошибки, и ни как не скажется на обнаруженных ранее.

Для доступа к значению текущего режима завершения работы предназначена функция getFinishMode.

Дополнительная информация

Заголовочный файл: ts/engine.h

Библиотека: ts

Смотрите также

Механизм построения тестов dfsm, механизм построения тестов ndfsm, FinishMode, getFinishMode, стандартный параметр «–uerr», стандартный параметр «-uend»

getFinishMode

Функция getFinishMode возвращает текущий режим завершения работы тестового сценария.

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

Возвращаемое значение

Текущий режим завершения работы тестового сценария.

Описание

Функция getFinishMode возвращает текущий режим завершения работы тестового сценария.

Для изменения текущего режима завершения работы предназначена функция setFinishMode.

Дополнительная информация

Заголовочный файл: ts/engine.h

Библиотека: ts

Смотрите также

Механизм построения тестов dfsm, механизм построения тестов ndfsm, FinishMode, setFinishMode, стандартный параметр «–uerr», стандартный параметр «–uend»

setDeferredReactionsMode

Функция setDeferredReactionsMode устанавливает режим поддержки отложенных реакций.

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

Параметры

enable

Значение параметра true соответствует включенному режиму поддержки отложенных реакций, значение false – выключенному.

Возвращаемое значение

Функция возвращает предыдущее значение режима поддержки отложенных реакций.

Описание

Функция setDeferredReactionsMode устанавливает режим поддержки отложенных реакций. Режим поддержки отложенных реакций не может быть включен, если хотя бы одно из полей saveModelState, restoreModelState или isStationaryState тестового сценария не определено или инициализировано нулевым указателем.

По умолчанию режим поддержки отложенных реакций принимает следующие значения:

- «выключен», если хотя бы одно из полей saveModelState, restoreModelState или isStationaryState тестового сценария не определено или инициализировано нулевым указателем,
- «включен», если поля saveModelState, restoreModelState и isStationaryState тестового сценария инициализированы ненулевыми указателями.

Изменять режим поддержки отложенных реакций разрешается только в функции инициализации тестового сценария.

Для доступа к текущему значению режима поддержки отложенных реакций предназначена функция areDeferredReactionsEnabled.

Дополнительная информация

Заголовочный файл: ts/engine.h

Библиотека: ts

Смотрите также

Механизм построения тестов dfsm, механизм построения тестов ndfsm, areDeferredReactionsEnabled, поле saveModelState, поле restoreModelState, поле isStationaryState

areDeferredReactionsEnabled

Функция areDeferredReactionsEnabled возвращает текущий режим поддержки отложенных реакций.

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

Возвращаемое значение

Функция areDeferredReactionsEnabled возвращает текущий режим поддержки отложенных реакций.

Описание

Функция areDeferredReactionsEnabled возвращает текущий режим поддержки отложенных реакций.

Для изменения режима поддержки отложенных реакций предназначена функция setDeferredReactionsMode.

Дополнительная информация

Заголовочный файл: ts/engine.h

Библиотека: ts

Смотрите также

Механизм построения тестов dfsm, механизм построения тестов ndfsm, setDeferredReactionsMode

setWTime

Функция setWTime устанавливает время ожидания стабилизации целевой системы.

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

Параметры

secs

Время ожидания стабилизации целевой системы в секундах.

Значение параметра должно быть неотрицательным целым числом.

Возвращаемое значение

Функция возвращает предыдущее значение времени ожидания стабилизации целевой системы.

Описание

Функция `setWTime` устанавливает время ожидания стабилизации целевой системы. Это время, которое ждет механизм тестирования после осуществления каждого тестового воздействия, для того, чтобы вся информация о реакциях была собрана и целевая система перешла в стационарное состояние.

По умолчанию, значение времени ожидания равно 0.

Изменять время ожидания стабилизации целевой системы разрешается только в функции инициализации тестового сценария.

Для получения текущего значения времени ожидания предназначена функция getWTime.

Дополнительная информация

Заголовочный файл: ts/engine.h

Библиотека: ts

Смотрите также

Механизм построения тестов dfsm, механизм построения тестов ndfsm, getWTime

getWTime

Функция getWTime возвращает время ожидания стабилизации целевой системы.

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

Возвращаемое значение

Функция getWTime возвращает время ожидания стабилизации целевой системы.

Описание

Функция getWTime возвращает время ожидания стабилизации целевой системы.

Для изменения времени ожидания стабилизации целевой системы предназначена функция setWTime.

Дополнительная информация

Заголовочный файл: ts/engine.h

Библиотека: ts

Смотрите также

Механизм построения тестов dfsm, механизм построения тестов ndfsm, setWTime

setFindFirstSeriesOnly

Функция setFindFirstSeriesOnly устанавливает свойство FindFirstSeriesOnly.

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

Параметры

new_value

Устанавливаемое значение свойства FindFirstSeriesOnly.

Возвращаемое значение

Функция возвращает предыдущее значение свойства FindFirstSeriesOnly.

Описание

Функция setFindFirstSeriesOnly устанавливает свойство FindFirstSeriesOnly. Если свойство принимает значение false, то во время сериализации взаимодействий с целевой системой, тестовый механизм строит все допустимые последовательности взаимодействий и проверяет, что все они проводят в одно и тоже состояние спецификационной модели данных. То есть проверяется детерминированность модели. Если из дополнительных знаний о модели известно, что все допустимые цепочки всегда приводят в одно и тоже стационарное состояние, то допускается установить значение свойства FindFirstSeriesOnly true и, таким образом, оптимизировать работу тестовой системы. Например, если в модели определено одно-единственное стационарное состояние спецификационной модели данных, то вышеуказанное условие заведомо выполняется и свойству FindFirstSeriesOnly допускается присвоить значение true.

По умолчанию, значение свойства равно false.

Изменять свойство FindFirstSeriesOnly разрешается только во время работы тестового сценария, в том числе в функции инициализации тестового сценария. Все изменения этого свойства до начала работы тестового сценария на его работу не повлияют.

Для получения текущего значения свойства FindFirstSeriesOnly предназначена функция isFindFirstSeriesOnly.

Дополнительная информация

Заголовочный файл: ts/engine.h

Библиотека: ts

Смотрите также

Механизм построения тестов dfsm, механизм построения тестов ndfsm, isFindFirstSeriesOnly

isFindFirstSeriesOnly

Функция isFindFirstSeriesOnly возвращает текущее значение свойства FindFirstSeriesOnly.

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

Возвращаемое значение

Функция isFindFirstSeriesOnly возвращает текущее значение свойства FindFirstSeriesOnly.

Описание

Функция isFindFirstSeriesOnly возвращает текущее значение свойства FindFirstSeriesOnly.

Для изменения значения свойства FindFirstSeriesOnly предназначена функция setFindFirstSeriesOnly.

Дополнительная информация

Заголовочный файл: ts/engine.h

Библиотека: ts

Смотрите также

Механизм построения тестов dfsm, механизм построения тестов ndfsm, setFindFirstSeriesOnly

setFindFirstSeriesOnlyBound

Функция setFindFirstSeriesOnlyBound устанавливает свойство FindFirstSeriesOnlyBound.

```
int setFindFirstSeriesOnly( int bound );
```

Параметры

bound

Устанавливаемое значение свойства FindFirstSeriesOnlyBound.

Возвращаемое значение

Функция возвращает предыдущее значение свойства FindFirstSeriesOnlyBound.

Описание

Функция setFindFirstSeriesOnlyBound устанавливает свойство FindFirstSeriesOnlyBound.

Если значение свойства FindFirstSeriesOnlyBound равно нулю, то во время сериализации взаимодействий с целевой системой, тестовый механизм будет строить все допустимые последовательности взаимодействий и проверять, что все они проводят в одно и тоже состояние спецификационной модели данных.

Если значение свойства FindFirstSeriesOnlyBound положительно, то во время сериализации взаимодействий с целевой системой, тестовый механизм будет только в том случае строить все допустимые последовательности взаимодействий, если число взаимодействий меньше значения свойства FindFirstSeriesOnlyBound, в противном случае, тестовый механизм рассмотрим единственную допустимую последовательность.

Вызов setFindFirstSeriesOnlyBound(0) эквивалентен вызову setFindFirstSeriesOnly(false). Вызов setFindFirstSeriesOnlyBound(1) эквивалентен вызову setFindFirstSeriesOnly(true).

По умолчанию, значение свойства равно 0.

Изменять свойство FindFirstSeriesOnlyBound разрешается только во время работы тестового сценария, в том числе в функции инициализации тестового сценария. Все изменения этого свойства до начала работы тестового сценария на его работу не повлияют.

Для получения текущего значения свойства FindFirstSeriesOnlyBound предназначена функция getFindFirstSeriesOnlyBound.

Дополнительная информация

Заголовочный файл: ts/engine.h

Библиотека: ts

Смотрите также

Механизм построения тестов dfsm, механизм построения тестов ndfsm, setFindFirstSeriesOnly, getFindFirstSeriesOnlyBound

getFindFirstSeriesOnlyBound

Функция getFindFirstSeriesOnlyBound возвращает текущее значение свойства FindFirstSeriesOnlyBound.

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

Возвращаемое значение

Функция getFindFirstSeriesOnlyBound возвращает текущее значение свойства FindFirstSeriesOnlyBound.

Описание

Функция getFindFirstSeriesOnlyBound возвращает текущее значение свойства FindFirstSeriesOnlyBound.

Для изменения значения свойства FindFirstSeriesOnlyBound предназначена функция setFindFirstSeriesOnlyBound.

Дополнительная информация

Заголовочный файл: ts/engine.h

Библиотека: ts

Смотрите также

Механизм построения тестов dfsm, механизм построения тестов ndfsm, setFindFirstSeriesOnlyBound

Стандартный параметр «–t»

Стандартный параметр «–t <file-name>» добавляет файл, указанный в следующим за ним параметре, в набор устройств для сохранения трассы. Если следующий параметр отсутствует или файл с указанным в нем именем невозможно открыть на запись, то работа тестового сценария аварийно завершается.

В ходе работы тестового сценария можно изменять набор устройств для сохранения трассы с помощью функций управления сохранением трассы.

Если среди параметров сценария присутствует несколько стандартных параметров управления сохранением трассы, то трасса будет направляться во все указанные в них устройства.

Если среди параметров сценария отсутствуют стандартные параметры управления сохранением трассы, это эквивалентно наличию стандартного параметра «–tt».

Все устройства, добавленные в набор устройств для сохранения трассы с помощью стандартных параметров сценария, автоматически удаляются оттуда после завершения работы этого сценария.

Смотрите также

Механизм построения тестов dfsm, механизм построения тестов ndfsm, стандартный параметр «–tc», стандартный параметр «–tt», стандартный параметр «-nt», Сервисы трассировки, Управление трассировкой, addTraceToFile, removeTraceToFile

Стандартный параметр «–tc»

Стандартный параметр «-tc» добавляет консоль в набор устройств для сохранения трассы. Под консолью понимается стандартный поток вывода процесса, в котором работает тестовая система.

В ходе работы тестового сценария можно изменять набор устройств для сохранения трассы с помощью функций управления сохранением трассы.

Если среди параметров сценария присутствует несколько стандартных параметров управления сохранением трассы, то трасса будет направляться во все указанные в них устройства.

Если среди параметров сценария отсутствуют стандартные параметры управления сохранением трассы, это эквивалентно наличию стандартного параметра «–tt».

Все устройства, добавленные в набор устройств для сохранения трассы с помощью стандартных параметров сценария, автоматически удаляются оттуда после завершения работы этого сценария.

Смотрите также

Механизм построения тестов dfsm, механизм построения тестов ndfsm, стандартный параметр «–t <file-name>», стандартный параметр «–tt», стандартный параметр «-nt», Сервисы трассировки, Управление трассировкой, addTraceToConsole, removeTraceToConsole

Стандартный параметр «-tt»

Стандартный параметр «-tt» добавляет файл со сгенерированным именем «<scenario_name>-YYYY-MM-DD--HH-MM-SS.utt» в набор устройств для сохранения трассы. Если файл с таким именем невозможно открыть на запись, то работа тестового сценария аварийно завершается.

В ходе работы тестового сценария можно изменять набор устройств для сохранения трассы с помощью функций управления сохранением трассы.

Если среди параметров сценария присутствует несколько стандартных параметров «–tt», то это эквивалентно присутствию только одного такого параметра.

Если среди параметров сценария присутствует несколько стандартных параметров управления сохранением трассы, то трасса будет направляться во все указанные в них устройства.

Если среди параметров сценария отсутствуют стандартные параметры управления сохранением трассы, это эквивалентно наличию стандартного параметра «–tt».

Все устройства, добавленные в набор устройств для сохранения трассы с помощью стандартных параметров сценария, автоматически удаляются оттуда после завершения работы этого сценария.

Смотрите также

Механизм построения тестов dfsm, механизм построения тестов ndfsm, стандартный параметр «-t <file-name>», стандартный параметр «-tc», стандартный параметр «-nt», Сервисы трассировки, Управление трассировкой, addTraceToFile, removeTraceToFile

Стандартный параметр «–nt»

Стандартный параметр «-nt» выключает трассировку.

Стандартный параметр «-nt» нельзя использовать вместе со стандартными параметрами «-t», «-tt» или «-tc».

В ходе работы тестового сценария можно изменять набор устройств для сохранения трассы с помощью функций управления сохранением трассы.

Если среди параметров сценария отсутствуют стандартные параметры управления сохранением трассы, это эквивалентно наличию стандартного параметра «–tt».

Смотрите также

Механизм построения тестов dfsm, механизм построения тестов ndfsm, стандартный параметр «–t <file-name>», стандартный параметр «–tt», стандартный параметр «-tc», Сервисы трассировки, Управление трассировкой, addTraceToConsole, removeTraceToConsole

Стандартный параметр «–uerr»

Стандартный параметр «-uerr» устанавливает значение режима завершения работы. Значение режима завершения может быть переопределено с помощью функции setFinishMode во время инициализации тестового сценария или дальнейшей его работы.

Если среди параметров сценария присутствует несколько стандартных параметров устанавливающих режим завершения работы, то механизм тестирования использует значение последнего из них.

Для механизма построения тестов ndfsm у стандартного параметра «-uerr» можно указать значение — число ошибок, после возникновения которого тестирование прекращается. Для этого используется формат -uerr=*число_ошибок*.

Смотрите также

Механизм построения тестов dfsm, механизм построения тестов ndfsm, стандартный параметр «–uend», setFinishMode

Стандартный параметр «–uend»

Стандартный параметр «-uend» устанавливает значение режима завершения работы. Значение режима завершения может быть переопределено с помощью функции setFinishMode во время инициализации тестового сценария или дальнейшей его работы.

Если среди параметров сценария присутствует несколько стандартных параметров устанавливающих режим завершения работы, то механизм тестирования использует значение последнего из них.

Смотрите также

Механизм построения тестов dfsm, механизм построения тестов ndfsm, стандартный параметр «–uerr», setFinishMode

Стандартный параметр «--trace-accidental»

Стандартный параметр «--trace-accidental» включает трассировку информации о несущественных переходах.

Трассировка информации о несущественных переходах может быть включена/отключена с помощью функции setTraceAccidental во время инициализации тестового сценария или дальнейшей его работы.

По умолчанию трассировка информации о несущественных переходах отключена.

Смотрите также

Механизм построения тестов dfsm, механизм построения тестов ndfsm, setTraceAccidental

Стандартный параметр «--find-first-series-only»

Стандартный параметр «--find-first-series-only» (в сокращенной форме «-ffso») устанавливает значение свойства FindFirstSeriesOnly в true. Это означает, что во время сериализации взаимодействий с целевой системой, тестовый механизм не будет строить все допустимые последовательности взаимодействий и проверять, что все они проводят в одно и тоже состояние спецификационной модели данных, а рассмотрит только одну допустимую последовательность взаимодействий.

Значение свойства FindFirstSeriesOnly может быть изменено с помощью функции setFindFirstSeriesOnly во время инициализации тестового сценария или дальнейшей его работы.

По умолчанию, значение свойства равно false.

Смотрите также

Механизм построения тестов dfsm, механизм построения тестов ndfsm, setFindFirstSeriesOnly

Сервисы трассировки

Трассировщик тестовой системы CTesK предоставляет возможность сохранять информацию о процессе тестирования для ее последующего анализа. Для этого все компоненты тестовой системы автоматически трассируют данные о своей работе в специальном формате. Впоследствии трасса используется генератором отчетов для анализа результатов тестирования и построения различных отчетов. Подробнее о генераторе отчетов можно прочитать в «*CTesK 2.2 Community Edition: Руководство пользователя*».

Для пользователя трассировщик предоставляет интерфейс управления трассировкой и интерфейс трассировки сообщений.

Управление трассировкой

Функции управления трассировкой делятся на два класса: функции управления сохранением трассы и функции управления содержанием трассы.

Функции управления сохранением трассы, работают по следующим принципам. Трасса теста может одновременно сохраняется на нескольких устройствах. CTesK 2.2 Community Edition поддерживается два вида устройств: консоль и файл. Под консолью понимается стандартный поток вывода процесса, в котором работает тестовая система.

Трассировщик сохраняет трассу на устройствах, входящих в *набор устройств для сохранения трассы*. Чтобы добавить устройство в этот набор используются функции из группы addTraceTo... Если добавляемое устройство уже принадлежит набору, то счетчик вхождения этого устройства в набор увеличивается.

Чтобы удалить устройство из набора устройств для сохранения трассы используются функции из группы removeTraceTo... Если удаляемое устройство было добавлено в набор несколько раз, то функция удаления только уменьшает счетчик вхождения этого устройства в набор. Реальное удаление устройства из набора происходит только тогда, когда счетчик вхождения обнуляется.

Изменять состав набора устройств для сохранения трассы разрешается только тогда, когда в тестовой системе не запущен ни один тестовый сценарий. В частности, в функции инициализации тестового сценария изменять этот набор не допускается.

Функции управления сохранением трассы:

- addTraceToConsole
- removeTraceToConsole
- addTraceToFile
- removeTraceToFile

Функции управления содержанием трассы определяют набор и формат сообщений трассировщика. В CTesK 2.2 Community Edition доступна только одна функция управления содержанием трассы:

- setTraceAccidental

Функция установки кодировки трассы:

setTraceEnconding

addTraceToConsole

Функция addTraceToConsole добавляет консоль в набор устройств для сохранения трассы.

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

Описание

Функция addTraceToConsole добавляет консоль в набор устройств для сохранения трассы. Если консоль уже принадлежит этому набору, то счетчик вхождения консоли в набор увеличивается.

Под консолью понимается стандартный поток вывода процесса, в котором работает тестовая система.

Функция addTraceToConsole может быть вызвана только тогда, когда в тестовой системе не запущен ни один тестовый сценарий.

Дополнительная информация

Заголовочный файл: ts/c_tracer.h

Библиотека: tracer

Смотрите также

Сервисы трассировки, Управление трассировкой, removeTraceToConsole, addTraceToFile, removeTraceToFile

removeTraceToConsole

Функция removeTraceToConsole удаляет консоль из набора устройств для сохранения трассы.

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

Описание

Функция removeTraceToConsole удаляет консоль из набора устройств для сохранения трассы. Если счетчик вхождения консоли в этот набор больше единицы, то счетчик уменьшается, а реального удаления консоли из набора не происходит.

Под консолью понимается стандартный поток вывода процесса, в котором работает тестовая система.

Функция removeTraceToConsole может быть вызвана только тогда, когда в тестовой системе не запущен ни один тестовый сценарий.

Дополнительная информация

Заголовочный файл: ts/c_tracer.h

Библиотека: tracer

Смотрите также

Сервисы трассировки, Управление трассировкой, addTraceToConsole, addTraceToFile, removeTraceToFile

addTraceToFile

Функция addTraceToFile добавляет указанный файл в набор устройств для сохранения трассы.

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

Параметры

name

Имя файла, добавляемого в набор устройств для сохранения трассы.

Параметр не должен быть нулевым указателем.

Возвращаемое значение

Функция возвращает true, если файл был добавлен успешно, и false — иначе. Причиной неудачи, например, может быть невозможность открыть файл с данным именем для записи.

Описание

Функция addTraceToFile добавляет указанный файл в набор устройств для сохранения трассы. Если этот файл уже принадлежит набору, то счетчик вхождения файла в набор увеличивается.

Если файл с данным именем невозможно открыть на запись, то функция возвращает false.

Функция addTraceToFile может быть вызвана только тогда, когда в тестовой системе не запущен ни один тестовый сценарий.

Дополнительная информация

Заголовочный файл: ts/c_tracer.h

Библиотека: tracer

Смотрите также

Сервисы трассировки, Управление трассировкой, removeTraceToFile, addTraceToConsole, removeTraceToConsole

removeTraceToFile

Функция removeTraceToFile удаляет указанный файл из набора устройств для сохранения трассы.

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

Параметры

name

Имя файла, удаляемого из набор устройств для сохранения трассы.

Параметр не должен быть нулевым указателем.

Возвращаемое значение

Функция возвращает false, если файл с данным именем не принадлежит набору устройств для сохранения трассы, и true — иначе.

Описание

Функция removeTraceToFile удаляет указанный файл из набора устройств для сохранения трассы. Если счетчик вхождения этого файла в набор больше единицы, то счетчик уменьшается, а реального удаления файла из набора не происходит.

Функция removeTraceToFile может быть вызвана только тогда, когда в тестовой системе не запущен ни один тестовый сценарий.

Дополнительная информация

Заголовочный файл: ts/c_tracer.h

Библиотека: tracer

Смотрите также

Сервисы трассировки, Управление трассировкой, addTraceToFile, addTraceToConsole, removeTraceToConsole

setTraceAccidental

Функция setTraceAccidental включает/отключает трассировку информации о *несущественных переходах*. Под несущественными переходами понимаются переходы, соответствующие вызовам сценарных функций, в течении которых не происходит вызова ни одной спецификационной функции.

```
bool setTraceAccidental( bool enable );
```

Параметры

enable

Если параметр принимает значение true, то функция включает трассировку информации о несущественных переходах, иначе — отключает.

Возвращаемое значение

Функция возвращает предыдущее значение свойства сохранения информации о несущественных переходах.

Описание

Функция setTraceAccidental включает/отключает трассировку информации о несущественных переходах.

По умолчанию трассировщик не сохраняет информацию о несущественных переходах.

Функция setTraceAccidental не может быть вызвана во время работы сценарной функции.

Дополнительная информация

Заголовочный файл: ts/c_tracer.h

Библиотека: tracer

Смотрите также

Сервисы трассировки, Управление трассировкой

setTraceEncoding

Функция setTraceEncoding устанавливает кодировку трассы.

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

Параметры

encoding

Идентификатор кодировки трассы.

Описание

Функция setTraceEncoding устанавливает кодировку трассы. По умолчанию используется кодировка UTF-8.

Для того чтобы локализованные строки, используемые в качестве имен ветвей функциональности, названий подсистем или пользовательских сообщений, корректно отображались в отчетах о проведенном тестировании, необходимо установить соответствующую кодировку трассы.

Дополнительная информация

Заголовочный файл: ts/c_tracer.h

Библиотека: tracer

Смотрите также

Сервисы трассировки, Управление трассировкой

Трассировка сообщений

В CTesK 2.2 Community Edition существует только один вид сообщений трассировщика доступный для пользователя. Эти сообщения так и называются: сообщения пользователя. Они играют вспомогательную роль и используются в основном для ручного анализа трассы. Но в некоторых отчетах об ошибках пользовательские сообщения отображаются для облегчения анализа ошибочных ситуаций. Подробнее о различных видах отчетов можно прочитать в «*CTesK 2.2 Community Edition: Руководство пользователя»*.

Для трассировки сообщений пользователя используются функции:

traceUserInfo

traceFormattedUserInfo

traceUserInfo

Функция traceUserInfo записывает в трассу сообщение пользователя.

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

Параметры

info

Указатель на строку, завершающуюся нулевым символом, которая содержит пользовательское сообщение.

Описание

Функция traceUserInfo записывает сообщение пользователя в трассу. Сообщения пользователя играют вспомогательную роль и используются в основном для ручного анализа трассы. Но в некоторых отчетах об ошибках пользовательские сообщения отображаются для облегчения анализа ошибочных ситуаций. Подробнее о различных видах отчетов можно прочитать в «*CTesK 2.2 Community Edition: Руководство пользователя*».

Дополнительная информация

Заголовочный файл: ts/c_tracer.h

Библиотека: tracer

Смотрите также

Сервисы трассировки, Трассировка сообщений

traceFormattedUserInfo

Функция traceFormattedUserInfo записывает в трассу форматированное сообщение пользователя.

```
void traceFormattedUserInfo( const char* format, ... );
```

Параметры

format

Указатель на строку, завершающуюся нулевым символом, которая содержит формат пользовательского сообщения. В строке допускаются все спецификаторы преобразований, допустимые в стандартной функции printf(), а также спецификатор $(obj) для преобразования в строку спецификационного объекта. Спецификаторы $(obj) должны располагаться перед спецификаторами функции printf().

Описание

Функция traceFormattedUserInfo форматирует и записывает сообщение пользователя в трассу. Сообщения пользователя играют вспомогательную роль и используются в основном для ручного анализа трассы. Но в некоторых отчетах об ошибках пользовательские сообщения отображаются для облегчения анализа ошибочных ситуаций. Подробнее о различных видах отчетов можно прочитать в «*CTesK 2.2 Community Edition: Руководство пользователя*».

Дополнительная информация

Заголовочный файл: ts/c_tracer.h

Библиотека: tracer

Смотрите также

Сервисы трассировки, Трассировка сообщений

Сервисы регистрации отложенных реакций

Тестовая система CTesK поддерживает тестирование *систем с отложенными реакциями*. Системами с отложенными реакциями называются такие системы, которые могут участвовать в нескольких взаимодействиях одновременно или которые могут сами инициировать взаимодействия с окружением.

Одной из важнейших задач при тестировании систем с отложенными реакциями является получение всей необходимой информации о взаимодействиях с целевой системой. Эта информация требуется тестовой системе CTesK для проверки соответствия реального поведения целевой системы требованиям, описанным в спецификациях, поскольку именно эти взаимодействия и отражают поведение целевой системы с отложенными реакциями.

Тестовая система автоматически регистрирует все взаимодействия, которые инициируются посредством вызова спецификационной функции внутри процесса тестовой системы. Все остальные взаимодействия должны быть зарегистрированы разработчиком теста в специальном компоненте тестовой системы — регистраторе взаимодействий.

Каждое взаимодействие с целевой системой характеризуется каналом, в котором оно произошло. Для идентификации каналов в тестовой системе используются идентификаторы каналов взаимодействия.

Дополнительно, для удобства регистрации отложенных взаимодействий, тестовая система предоставляет сервис регистрации функций-сборщиков реакций.

Каналы взаимодействия

Каждое взаимодействие с целевой системой характеризуется каналом, в котором оно произошло. Все взаимодействия внутри одного канала линейно упорядочены. Поэтому тестовая система считает, что, если два взаимодействия характеризуются одним каналом, то первое зарегистрированное в ней произошло раньше второго.

Идентификаторы каналов взаимодействия, использующиеся для их идентификации внутри тестовой системы, имеют тип ChannelID.

Существует две предопределенные константы этого типа:

- WrongChannel
- UniqueChannel

Для получения свободного идентификатора канала и последующего его освобождения предназначены функции:

- getChannelID
- releaseChannelID

ChannelID

Тип ChannelID используется для идентификации каналов взаимодействия внутри тестовой системы.

```
typedef long ChannelID;
```

Описание

Тип ChannelID используется для идентификации каналов взаимодействия внутри тестовой системы. Существует две константы этого типа WrongChannel и UniqueChannel.

Функция getChannelID возвращает свободный идентификатор канала. После того как идентификатор канала становится ненужным, его можно освободить с помощью функции releaseChannelID.

Корректными идентификаторами каналов считаются константа UniqueChannel и все идентификаторы, полученные с помощью функции getChannelID и отличные от WrongChannel.

Дополнительная информация

Заголовочный файл: ts/register.h

Библиотека: ts

Смотрите также

Сервисы регистрации асинхронных взаимодействий, Каналы взаимодействия, WrongChannel, UniqueChannel, getChannelID, releaseChannelID

WrongChannel

Константа WrongChannel обозначает некорректный идентификатор канала взаимодействия.

```
extern const ChannelID WrongChannel;
```

Описание

Константа WrongChannel обозначает некорректный идентификатор канала взаимодействия. Эта константа используется для вспомогательных целей. Например, функция getChannelID возвращает эту константу, если не может выделить свободный идентификатор канала.

Дополнительная информация

Заголовочный файл: ts/register.h

Библиотека: ts

Смотрите также

Сервисы регистрации асинхронных взаимодействий, Каналы взаимодействия, ChannelID, UniqueChannel, getChannelID, releaseChannelID

UniqueChannel

Константа UniqueChannel используется для идентификации уникального канала, в котором произошло одно взаимодействие и других взаимодействий произойти не может в принципе.

```
extern const ChannelID UniqueChannel;
```

Описание

Константа UniqueChannel используется для идентификации уникального канала, в котором произошло одно взаимодействие и других взаимодействий произойти не может в принципе. Эта константа часто используется, когда понятие каналов взаимодействия не применяется при моделировании целевой системы.

Дополнительная информация

Заголовочный файл: ts/register.h

Библиотека: ts

Смотрите также

Сервисы регистрации асинхронных взаимодействий, Каналы взаимодействия, ChannelID, WrongChannel, getChannelID, releaseChannelID

getChannelID

Функция getChannelID возвращает свободный идентификатор канала взаимодействия.

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

Возвращаемое значение

Функция возвращает свободный идентификатор канала взаимодействия, если таковой существует, и WrongChannel — в противном случае.

Описание

Функция getChannelID возвращает свободный идентификатор канала взаимодействия. Если свободных идентификаторов не осталось, то возвращается константа WrongChannel.

Если идентификатор канала становится ненужным, его можно освободить с помощью функции releaseChannelID.

Дополнительная информация

Заголовочный файл: ts/register.h

Библиотека: ts

Смотрите также

Сервисы регистрации асинхронных взаимодействий, Каналы взаимодействия, ChannelID, WrongChannel, UniqueChannel, releaseChannelID

releaseChannelID

Функция releaseChannelID освобождает указанный идентификатор канала взаимодействия.

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

Параметры

chid

Освобождаемый идентификатор канала взаимодействия.

Параметр должен идентификатором канала, полученный ранее с помощью функции getChannelID.

Описание

Функция releaseChannelID освобождает указанный идентификатор канала взаимодействия, который был ранее получен с помощью функции getChannelID.

Дополнительная информация

Заголовочный файл: ts/register.h

Библиотека: s

Смотрите также

Сервисы регистрации асинхронных взаимодействий, Каналы взаимодействия, ChannelID, WrongChannel, UniqueChannel, getChannelID

Регистратор взаимодействий

Тестовая система автоматически регистрирует все взаимодействия, которые инициируются посредством вызова спецификационной функции внутри процесса тестовой системы. При этом считается, что взаимодействие происходит в канале с идентификатором, заданным свойством StimulusChannel. Для управления этим свойством предназначены функции:

- setStimulusChannel
- getStimulusChannel

По умолчанию, это свойство принимает значение UniqueChannel.

Все остальные взаимодействия должны быть зарегистрированы разработчиком теста в регистраторе взаимодействий. Для этого предназначены следующие функции:

- registerReaction
- registerReactionWithTimeMark
- registerReactionWithTimeInterval
- registerWrongReaction
- registerStimulusWithTimeInterval

setStimulusChannel

Функция setStimulusChannel устанавливает значение свойства StimulusChannel.

```
ChannelID setStimulusChannel( ChannelID chid );
```

Параметры

chid

Идентификатор канала взаимодействия, который будет использоваться тестовой системой при автоматической регистрации взаимодействий.

Параметр должен быть корректным идентификатором канала.

Возвращаемое значение

Функция возвращает предыдущее значение свойства StimulusChannel.

Описание

Функция setStimulusChannel устанавливает значение свойства StimulusChannel. Свойство StimulusChannel содержит идентификатор канала, используемый тестовой системой для идентификации канала, в котором происходят взаимодействия, инициированные посредством вызова спецификационной функции внутри процесса тестовой системы. По умолчанию, это свойство принимает значение UniqueChannel.

Для доступа к текущему значению свойства StimulusChannel предназначена функция getStimulusChannel.

Дополнительная информация

Заголовочный файл: ts/register.h

Библиотека: ts

Смотрите также

Сервисы регистрации асинхронных взаимодействий, Регистратор взаимодействий, Каналы взаимодействия, ChannelID, UniqueChannel, getStimulusChannel

getStimulusChannel

Функция getStimulusChannel возвращает значение свойства StimulusChannel.

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

Возвращаемое значение

Функция возвращает значение свойства StimulusChannel.

Описание

Функция getStimulusChannel возвращает значение свойства StimulusChannel. Свойство StimulusChannel содержит идентификатор канала, используемый тестовой системой для идентификации канала, в котором происходят взаимодействия, инициированные посредством вызова спецификационной функции внутри процесса тестовой системы. По умолчанию, это свойство принимает значение UniqueChannel.

Для изменения значения свойства StimulusChannel предназначена функция setStimulusChannel.

Дополнительная информация

Заголовочный файл: ts/register.h

Библиотека: ts

Смотрите также

Сервисы регистрации асинхронных взаимодействий, Регистратор взаимодействий, Каналы взаимодействия, ChannelID, UniqueChannel, setStimulusChannel

registerReaction

Функция registerReaction предназначена для регистрации реакций, полученных от целевой системы. Реакцией называется взаимодействие, инициированное целевой системой.

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

Параметры

chid

Идентификатор канала взаимодействия, в котором была получена данная реакция.

Параметр должен быть корректным идентификатором канала.

name

Имя реакции, используемое только для трассировки.

Параметр может быть нулевым указателем. В этом случае, считается, что имя данного взаимодействия совпадает с именем реакции *reactionID*.

reactionID

Идентификатор реакции, которому соответствует регистрируемое взаимодействие.

data

Данные полученные от целевой системы в модельном представлении.

Тип параметра должен совпадать с типом возвращаемого значения реакции *reactionID*.

Описание

Функция registerReaction предназначена для регистрации реакций, полученных от целевой системы. Реакцией называется взаимодействие, инициированное целевой системой.

Основными свойствами взаимодействия являются имя реакции *reactionID* и данные *data*, полученные во время взаимодействия. Тип полученных данных должен совпадать с типом возвращаемого значения реакции.

Временные метки и интервалы, а также каналы взаимодействий используются тестовой системой для упорядочивания всех зарегистрированных взаимодействий между собой.

Если данные, полученные от целевой системы, не удается преобразовать в модельное представление, то необходимо уведомить тестовую систему о получении некорректной реакции с помощью функции registerWrongReaction.

Дополнительная информация

Заголовочный файл: ts/register.h

Библиотека: ts

Смотрите также

Сервисы регистрации асинхронных взаимодействий, Регистратор взаимодействий, Каналы взаимодействия, ChannelID, registerReactionWithTimeMark, registerReactionWithTimeInterval, registerWrongReaction

registerReactionWithTimeMark

Функция registerReactionWithTimeMark предназначена для регистрации реакций, полученных от целевой системы, с указанием временной метки момента получения. Реакцией называется взаимодействие, инициированное целевой системой.

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

Параметры

chid

Идентификатор канала взаимодействия, в котором была получена данная реакция.

Параметр должен быть корректным идентификатором канала.

name

Имя реакции, используемое только для трассировки.

Параметр может быть нулевым указателем. В этом случае, считается, что имя данного взаимодействия совпадает с именем реакции *reactionID*.

reactionID

Идентификатор реакции, которому соответствует регистрируемое взаимодействие.

data

Данные полученные от целевой системы в модельном представлении.

Тип параметра должен совпадать с типом возвращаемого значения реакции *reactionID*.

mark

Временная метка момента взаимодействия.

Описание

Функция registerReactionWithTimeMark предназначена для регистрации реакций, полученных от целевой системы, с указанием временной метки момента получения. Реакцией называется взаимодействие, инициированное целевой системой.

Основными свойствами взаимодействия являются имя реакции *reactionID* и данные *data*, полученные во время взаимодействия. Тип полученных данных должен совпадать с типом возвращаемого значения реакции.

Временные метки и интервалы, а также каналы взаимодействий используются тестовой системой для упорядочивания всех зарегистрированных взаимодействий между собой.

Если данные, полученные от целевой системы, не удается преобразовать в модельное представление, то необходимо уведомить тестовую систему о получении некорректной реакции с помощью функции registerWrongReaction.

Дополнительная информация

Заголовочный файл: ts/register.h

Библиотека: ts

Смотрите также

Сервисы регистрации асинхронных взаимодействий, Регистратор взаимодействий, Каналы взаимодействия, ChannelID, TimeMark, registerReaction, registerReactionWithTimeInterval, registerWrongReaction

registerReactionWithTimeInterval

Функция registerReactionWithTimeInterval предназначена для регистрации реакций, полученных от целевой системы, с указанием временного интервала, в котором оно происходило. Реакцией называется взаимодействие, инициированное целевой системой.

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

Параметры

chid

Идентификатор канала взаимодействия, в котором была получена данная реакция.

Параметр должен быть корректным идентификатором канала.

name

Имя реакции, используемое только для трассировки.

Параметр может быть нулевым указателем. В этом случае, считается, что имя данного взаимодействия совпадает с именем реакции *reactionID*.

reactionID

Идентификатор реакции, которому соответствует регистрируемое взаимодействие.

data

Данные полученные от целевой системы в модельном представлении.

Тип параметра должен совпадать с типом возвращаемого значения реакции *reactionID*.

interval

Временной интервал, в котором происходило взаимодействие. При этом имеется ввиду, что взаимодействие происходило где-то внутри интервала, а не занимало весь интервал целиком.

Описание

Функция registerReactionWithTimeInterval предназначена для регистрации реакций, полученных от целевой системы, с указанием временного интервала, в котором оно происходило. Реакцией называется взаимодействие, инициированное целевой системой.

Основными свойствами взаимодействия являются имя реакции *reactionID* и данные *data*, полученные во время взаимодействия. Тип полученных данных должен совпадать с типом возвращаемого значения реакции.

Временные метки и интервалы, а также каналы взаимодействий используются тестовой системой для упорядочивания всех зарегистрированных взаимодействий между собой.

Если данные, полученные от целевой системы, не удается преобразовать в модельное представление, то необходимо уведомить тестовую систему о получении некорректной реакции с помощью функции registerWrongReaction.

Дополнительная информация

Заголовочный файл: ts/register.h

Библиотека: ts

Смотрите также

Сервисы регистрации асинхронных взаимодействий, Регистратор взаимодействий, Каналы взаимодействия, ChannelID, TimeInterval, registerReaction, registerReactionWithTimeMark, registerWrongReaction

registerWrongReaction

Функция registerWrongReaction предназначена для уведомления тестовой системы о получении некорректной реакции, которую невозможно преобразовать в модельное представление. Реакцией называется взаимодействие, инициированное целевой системой.

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

Параметры

info

Описание некорректной реакции, используемое при анализе результатов тестирования.

Параметр может быть нулевым указателем.

Описание

Функция registerWrongReaction предназначена для уведомления тестовой системы о получении некорректной реакции, которую невозможно преобразовать в модельное представление. Реакцией называется взаимодействие, инициированное целевой системой.

После регистрации некорректной реакции тестовая система завершит анализ текущего тестового воздействия с отрицательным вердиктом.

Дополнительная информация

Заголовочный файл: ts/register.h

Библиотека: ts

Смотрите также

Сервисы регистрации асинхронных взаимодействий, Регистратор взаимодействий, registerReaction, registerReactionWithTimeMark, registerReactionWithTimeInterval, registerStimulusWithTimeInterval

registerStimulusWithTimeInterval

Функция registerStimulusWithTimeInterval предназначена для регистрации стимула, не зарегистрированного тестовой системой автоматически. Стимулом называется взаимодействие с целевой системой, инициированное тестом.

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

Параметры

chid

Идентификатор канала взаимодействия, по которому был подан данный стимул.

Параметр должен быть корректным идентификатором канала.

name

Имя стимула, используемое только для трассировки.

Параметр может быть нулевым указателем. В этом случае, считается, что имя данного взаимодействия совпадает с именем спецификационной функции *stimulusID*.

stimulusID

Идентификатор спецификационной функции, которому соответствует регистрируемое взаимодействие.

interval

Временной интервал, в котором происходило взаимодействие. При этом имеется ввиду, что взаимодействие происходило где-то внутри интервала, а не занимало весь интервал целиком.

arguments

Дополнительные аргументы, которые должны передаваться строго в следующем порядке:

- Список значений параметров спецификационной функции до ее вызова.
- Список значений параметров спецификационной функции после ее вызова.
- Значение, которое вернула данная спецификационная функция (если тип возвращаемого значения отличен от void).

Описание

Функция registerStimulusWithTimeInterval предназначена для регистрации стимула, не зарегистрированного тестовой системой автоматически. Так как все стимулы, которые инициируются посредством вызова спецификационной функции внутри процесса тестовой системы, регистрируются автоматически, то регистрировать вручную необходимо только те стимулы, которые инициируются либо извне процесса тестовой системы, либо не посредством вызова спецификационной функции.

Основными свойствами взаимодействия являются имя спецификационной функции *stimulusID* и данные, передаваемые через дополнительные аргументы.

Временные метки и интервалы, а также каналы взаимодействий используются тестовой системой для упорядочивания всех зарегистрированных взаимодействий между собой.

Если данные, полученные во время взаимодействия от целевой системы, не удается преобразовать в модельное представление, то необходимо уведомить тестовую систему о некорректном поведении целевой системы во время взаимодействия, вызвав функцию registerWrongReaction в основном процессе тестовой системы.

Дополнительная информация

Заголовочный файл: ts/register.h

Библиотека: ts

Смотрите также

Сервисы регистрации асинхронных взаимодействий, Регистратор взаимодействий, Каналы взаимодействия, ChannelID, TimeInterval, registerWrongReaction

Сервис регистрации функций-сборщиков реакций

Для удобства регистрации отложенных взаимодействий тестовая система предоставляет сервис регистрации функций-сборщиков реакций. Сервис устроен следующим образом. В тестовой системе регистрируются функции, которые вызываются ей по истечении времени стабилизации после каждого тестового воздействия. За время стабилизации целевая система должна выдать все ожидаемые от нее реакции и перейти в стационарное состояние. Во время вызова функции-сборщики должны собрать все данные о полученных реакциях и зарегистрировать их в Регистраторе Взаимодействий.

Для регистрации функций-сборщиков предусмотрены следующие функции:

- registerReactionCatcher
- unregisterReactionCatcher
- unregisterReactionCatchers

ReactionCatcherFuncType

Тип ReactionCatcherFuncType используется для регистрации функций-сборщиков реакций в тестовой системе.

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

Описание

Тип ReactionCatcherFuncType используется в качестве типа функций-сборщиков реакций.

Дополнительная информация

Заголовочный файл: ts/timemark.h

Библиотека: ts

Смотрите также

Сервисы регистрации асинхронных взаимодействий, Сервис регистрации функций-сборщиков реакций, registerReactionCatcher, unregisterReactionCatcher, unregisterReactionCatchers

registerReactionCatcher

Функция registerReactionCatcher регистрирует в тестовой системе функцию-сборщик реакций и ее вспомогательные данные.

```
void registerReactionCatcher(
        ReactionCatcherFuncType  catcher,
        void*                    par
                            );
```

Параметры

catcher

Указатель на функцию-сборщик реакций.

Параметр не должен быть нулевым указателем.

par

Вспомогательные данные регистрируемой функции.

Параметр может быть нулевым указателем.

Описание

Функция registerReactionCatcher регистрирует в тестовой системе функцию-сборщик реакций и ее вспомогательные данные.

Когда тестовая система вызывает функцию-сборщик, то передает ей в качестве параметра вспомогательные данные.

Можно зарегистрировать в тестовой системе одну функцию-сборщик реакций несколько раз с различными вспомогательными данными. При этом функция будет вызываться соответствующее число раз с различными значениями параметра.

Дополнительная информация

Заголовочный файл: ts/register.h

Библиотека: ts

Смотрите также

Сервисы регистрации асинхронных взаимодействий, Сервис регистрации функций-сборщиков реакций, ReactionCatcherFuncType, unregisterReactionCatcher, unregisterReactionCatchers

unregisterReactionCatcher

Функция unregisterReactionCatcher удаляет из тестовой системы регистрационную запись о данной функции-сборщике реакций, зарегистрированной с указанными вспомогательными данными.

```
bool unregisterReactionCatcher(
            ReactionCatcherFuncType    catcher,
            void*                      par
                                );
```

Параметры

catcher

Указатель на функцию-сборщик реакций.

Параметр не должен быть нулевым указателем.

par

Вспомогательные данные, переданные при регистрации этой функции.

Возвращаемое значение

Функция возвращает false, если данная функция с данными вспомогательными данными не была ранее зарегистрирована, и true — иначе.

Описание

Функция unregisterReactionCatcher удаляет из тестовой системы регистрационную запись о данной функции-сборщике реакций, зарегистрированной с указанными вспомогательными данными.

Для одновременного удаления всех регистрационных записей о данной функции-сборщике реакций используется функция unregisterReactionCatchers.

Дополнительная информация

Заголовочный файл: ts/register.h

Библиотека: ts

Смотрите также

Сервисы регистрации асинхронных взаимодействий, Сервис регистрации функций-сборщиков реакций, ReactionCatcherFuncType, registerReactionCatcher, unregisterReactionCatchers

unregisterReactionCatchers

Функция unregisterReactionCatchers удаляет из тестовой системы все регистрационные записи о данной функции-сборщике реакций.

```
bool unregisterReactionCatchers( ReactionCatcherFuncType catcher );
```

Параметры

catcher

Указатель на функцию-сборщик реакций.

Параметр не должен быть нулевым указателем.

Возвращаемое значение

Функция возвращает false, если данная функция-сборщик не была ранее зарегистрирована, и true — иначе.

Описание

Функция unregisterReactionCatchers удаляет из тестовой системы все регистрационные записи о данной функции-сборщике реакций.

Для удаления только одной регистрационной записи о данной функции-сборщике с конкретными вспомогательными данными используется функция unregisterReactionCatcher.

Дополнительная информация

Заголовочный файл: ts/register.h

Библиотека: ts

Смотрите также

Сервисы регистрации асинхронных взаимодействий, Сервис регистрации функций-сборщиков реакций, ReactionCatcherFuncType, registerReactionCatcher, unregisterReactionCatcher

Библиотека спецификационных типов

Библиотека спецификационных типов содержит стандартные функции для работы со спецификационными типами (создание, копирование, сравнение, построение строкового представления), а также предопределенные спецификационные типы для стандартных типов языка C (char, short, int, long, float, double, void*, строки char*), для комплексных чисел и типа с единственным значением, и для контейнерных типов (список, множество, отображение).

Предопределенными типами удобно пользоваться в *спецификациях* (например, для моделирования реализационного состояния), поскольку они предоставляют уже готовую, универсальную, гарантированно безошибочную функциональность.

Стандартные функции

Стандартные функции могут использоваться с любыми спецификационными ссылками. Результат выполнения функции будет зависеть от типа спецификационных ссылок. Например, если сравниваются две спецификационные ссылки разных типов, то результат всегда будет отрицательный, а если ссылки имеют одинаковый тип, то результат сравнения будет определяться функцией, заданной при описании этого типа.

Тип спецификационных ссылок определяется указателем на *дескриптор спецификационного типа*. Константа-дескриптор всегда имеет имя, состоящее из имени спецификационного типа с префиксом type_:

```
const Type type_имя_спецификационного_типа;
```

Функция создания ссылок

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

В качестве первого параметра функция получает указатель на *дескриптор спецификационного типа*. Остальные параметры являются параметрами инициализации типа и отличаются для разных типов. Для всех предопределенных типов параметры описаны в соответствующих разделах, а параметры инициализации пользовательских спецификационных типов задаются при описании этих типов.

Функция возвращает указатель на созданный объект.

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

В приведенном выше примере создается и инициализируется ссылка библиотечного спецификационного типа Integer — спецификационного аналога типа int.

Функция получения типа ссылки

const Type *type(Object* ref)

Функция возвращает указатель на константу-дескриптор спецификационного типа, на значение которого ссылается указатель ref.

```
Integer* ref = create(&type_Integer, 28);
if (type(ref) == &type_Integer) // истинно
```

Функции копирования значений по ссылкам

void copy(Object* src, Object* dst)

Функция копирует содержимое, находящееся по ссылке src, в содержимое, находящееся по ссылке dst. Ссылки должны быть ненулевыми и одного типа, то есть они должны

иметь одинаковые дескрипторы типов. Если эти условия не выполняются, то во время выполнения происходит завершение программы с сообщением об ошибке.

```
Integer* ref1 = create(&type_Integer, 28); // ref1 → 28
Integer* ref2 = create(&type_Integer, 47); // ref2 → 47
copy(ref1,ref2);                           // ref1 → 28, ref2 → 28
```

В приведенном выше примере ссылки ref1 и ref2 после инициализации ссылаются на разные значения спецификационного типа Integer. После вызова функции copy() значение по ссылке ref2 становится эквивалентным значению по ссылке ref1.

Object* clone(Object* ref)

Функция выделяет память для значения типа, на который ссылается ref, инициализирует выделенную память значением, эквивалентным значению по ссылке ref, и возвращает указатель на выделенную и инициализированную память.

```
Integer* ref1 = create(&type_Integer, 28); // ref1 → 28
String*  ref2 = clone(ref1);               // ref2 → 28
```

Значения по ссылкам ref1 и ref2 становятся эквивалентными после вызова clone().

Функции сравнения значений по ссылкам

int compare(Object* left, Object* right)

В случае эквивалентности значений по переданным ссылкам функция возвращает нулевое значение. Если значения не эквивалентны, функция возвращает ненулевое значение, которое может интерпретироваться в зависимости от типа сравниваемых значений. Например, для библиотечного типа String результат будет таким же, как у функции strcmp() для типа языка C char*. Если параметры имеют несравнимые типы, то есть типы ссылок неодинаковые, не являются подтипами одного типа, и тип одной ссылки не является подтипом другой (см. Инварианты типов), то функция возвращает ненулевое значение. Если одна из ссылок нулевая, а другая нет, то возвращается ненулевое значение. Если обе ссылки нулевые, то возвращается ноль.

```
if (!compare(ref1,ref2)) {/* значения эквивалентны */
  ...
}
else {/* значения не эквивалентны */
  ...
}
```

bool equals(Object* self, Object* ref)

Функция возвращает значение true, если значения по переданным ссылкам эквивалентны, и false в противном случае. Если параметры имеют различный тип, то функция возвращает false. Если одна из ссылок нулевая, а другая нет, то возвращается false. Если обе ссылки нулевые, то возвращается true.

```
if (equals(ref1,ref2)) {/* значения эквивалентны */
  ...
}
else {/* значения не эквивалентны */
  ...
}
```

Функция построения строкового представления значения по ссылке

`String* toString(Object* ref)`

Функция возвращает ссылку на значение типа String — спецификационное представление строкового типа.

```
Integer* ref = create(&type_Integer, 28);          // ref → 28
String*  str = toString(ref);                      // str → "28"
printf("*ref == '%s'\n", toCharArray_String(str));
```

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

Предопределенные спецификационные типы

Char

#include <atl/char.h>

Тип Char является спецификационным аналогом встроенного типа языка C char.

Char* create_Char(char d)

Создать спецификационную ссылку на значение типа Char.

Эта функция определена наряду со стандартной функций create():

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

Два показанных способа создания спецификационной ссылки эквивалентны.

char value_Char(Char* d)

Получить значение char, содержащееся в спецификационном типе.

К этому значению также можно обратиться с помощью разыменования спецификационной ссылки:

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

Функция value_Char() обеспечивает контроль типов во время исполнения и ей можно передавать указатель Object* без предварительного приведения:

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

Однако если нет уверенности в том, что переданная ссылка имеет нужный тип, может потребоваться явная проверка с помощью стандартной функции type():

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

Integer и UInteger

#include <atl/integer.h>

Типы Integer и UInteger являются спецификационными аналогами встроенных типов языка C int и unsigned int.

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

Создать спецификационную ссылку на значение типа Integer и UInteger.

Эта функция определена наряду со стандартной функций create():

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

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

Два показанных способа создания спецификационных ссылок эквивалентны.

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

Получить значение int и unsigned int, содержащиеся в спецификационных типах Integer и UInteger.

К этим значениям также можно обратиться с помощью разыменования спецификационной ссылки:

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

Функции value_Integer() и value_UInteger() обеспечивают контроль типов во время исполнения и им можно передавать указатель Object* без предварительного приведения:

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

Однако если нет уверенности в том, что переданная ссылка имеет нужный тип, может потребоваться явная проверка с помощью стандартной функции type():

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

Short и UShort

#include <atl/short.h>

Типы Short и UShort являются спецификационными аналогами встроенных типов языка C short int и unsigned short int.

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

Создать спецификационную ссылку на значение типа Short и UShort.

Эта функция определена наряду со стандартной функций create():

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

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

Два показанных способа создания спецификационных ссылок эквивалентны.

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

Получить значение short и unsigned short, содержащиеся в спецификационных типах Short и UShort.

К этим значениям также можно обратиться с помощью разыменования спецификационной ссылки:

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

Функции value_Short() и value_UShort() обеспечивают контроль типов во время исполнения и им можно передавать указатель Object* без предварительного приведения:

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

Однако если нет уверенности в том, что переданная ссылка имеет нужный тип, может потребоваться явная проверка с помощью стандартной функции type():

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

Long и ULong

#include <atl/long.h>

Типы Long и ULong являются спецификационными аналогами встроенных типов языка C long int и unsigned long int.

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

Создать спецификационную ссылку на значение типа Long и ULong.

Эта функция определена наряду со стандартной функций create():

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

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

Два показанных способа создания спецификационных ссылок эквивалентны.

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

Получить значение long и unsigned long, содержащиеся в спецификационных типах Long и ULong.

К этим значениям также можно обратиться с помощью разыменования спецификационной ссылки:

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

Функции value_Long() и value_ULong() обеспечивают контроль типов во время исполнения и им можно передавать указатель Object* без предварительного приведения:

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

Однако если нет уверенности в том, что переданная ссылка имеет нужный тип, может потребоваться явная проверка с помощью стандартной функции type():

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

Float

#include <atl/float.h>

Тип Float является спецификационным аналогом встроенного типа языка C float.

Float* create_Float(float d)

Создать спецификационную ссылку на значение типа Float.

Эта функция определена наряду со стандартной функций create():

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

Два показанных способа создания спецификационной ссылки эквивалентны.

float value_Float (Float* d)

Получить значение float, содержащееся в спецификационном типе.

К этому значению также можно обратиться с помощью разыменования спецификационной ссылки:

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

Функция value_Float () обеспечивает контроль типов во время исполнения и ей можно передавать указатель Object* без предварительного приведения:

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

Однако если нет уверенности в том, что переданная ссылка имеет нужный тип, может потребоваться явная проверка с помощью стандартной функции type():

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

Double

#include <atl/double.h>

Тип Double является спецификационным аналогом встроенного типа языка C double.

Double* create_Double (double d)

Создать спецификационную ссылку на значение типа Double.

Эта функция определена наряду со стандартной функций create():

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

Два показанных способа создания спецификационной ссылки эквивалентны.

double value_Double (Double* d)

Получить значение double, содержащееся в спецификационном типе.

К этому значению также можно обратиться с помощью разыменования спецификационной ссылки:

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

Функция value_Double() обеспечивает контроль типов во время исполнения и ей можно передавать указатель Object* без предварительного приведения:

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

Однако если нет уверенности в том, что переданная ссылка имеет нужный тип, может потребоваться явная проверка с помощью стандартной функции type():

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

VoidAst

#include <atl/void_ast.h>

Тип VoidAst является спецификационным аналогом типа языка C void*.

VoidAst* create_VoidAst (void *d)

Создать спецификационную ссылку на значение типа VoidAst.

Эта функция определена наряду со стандартной функций create():

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

Два показанных способа создания спецификационной ссылки эквивалентны.

void *value_VoidAst (VoidAst* d)

Получить значение void*, содержащееся в спецификационном типе.

К этому значению также можно обратиться с помощью разыменования спецификационной ссылки:

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

Функция value_VoidAst() обеспечивает контроль типов во время исполнения и ей можно передавать указатель Object* без предварительного приведения:

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

Однако если нет уверенности в том, что переданная ссылка имеет нужный тип, может потребоваться явная проверка с помощью стандартной функции type():

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

Unit

#include <atl/unit.h>

Тип Unit является типом с одним единственным значением.

Две (ненулевые) спецификационные ссылки типа Unit всегда считаются равными друг другу.

Unit* create_Unit ()

Создать спецификационную ссылку на значение типа Unit.

Эта функция определена наряду со стандартной функций create():

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

Два показанных способа создания спецификационной ссылки эквивалентны.

Complex

#include <atl/complex.h>

Тип Complex служит для представления комплексных чисел.

Для спецификационных ссылок на тип Complex действуют обычные правила сравнения ($(re_1, im_1) = (re_2, im_2) \leftrightarrow re_1 = re_2, im_1 = im_2$). Строковое представление имеет вид `( re + im*i )`.

Базовым типом для типа Complex служит структура:

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

Для получения действительной и мнимой частей не определено специальных функций; следует пользоваться разыменованием спецификационной ссылки:

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

Complex* create_Complex (double re, double im)

Создать спецификационную ссылку на значение типа Complex с действительной частью re и мнимой im.

Эта функция определена наряду со стандартной функций create():

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

Два показанных способа создания спецификационной ссылки эквивалентны.

String

#include <atl/string.h>

Тип String служит для представления строк.

Принятый в языке C способ представления строк в виде массива типа char не укладывается в рамки *допустимого типа*. Поэтому везде, где требуется *допустимый тип*, удобно представлять строки данным спецификационным типом.

Для спецификационных ссылок на тип String действуют обычные правила сравнения строк (как в функции strcmp). Позиции символов нумеруются от 0.

Со спецификационными строками можно работать как с обычными C-строками, принимая во внимание, что значение этой строки не должно изменяться. Для получения доступа к C-строке используется функция toCharArray_String(), возвращающая указатель на массив внутри спецификационного типа.

В то же время определено большое количество функций и для самих спецификационных строк. Во всех этих функциях спецификационные ссылки на тип String не должны быть нулевыми.

String* create_String (const char *cstr)

Создать спецификационную ссылку типа String и инициализировать ее C-строкой cstr.

Эта функция определена наряду со стандартной функций create():

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

Два показанных способа создания спецификационной ссылки эквивалентны.

char charAt_String(String* self, int index)

Возвращает символ в данной позиции index.

Номер позиции должен находиться в интервале от 0 до length_String(self)−1.

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

```
c
```

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

Возвращает конкатенацию строк self и str.

```
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)

Проверяет, заканчивается ли строка self строкой suffix.

Если строка suffix пуста, возвращает true. Если длина строки suffix больше, чем длина self, возвращает 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)

Возвращает позицию первого символа ch в строке. Если символ не найден, возвращает –1.

Для символа с кодом 0 всегда возвращает –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)

Возвращает позицию символа ch в строке, начиная с позиции fromIndex. Если символ не найден, возвращает –1.

Если позиция fromIndex превосходит длину строки self, т. е. fromIndex > length_String(self), возвращает –1. Если fromIndex < 0, то позиция принимается за 0. Для символа с кодом 0 всегда возвращает –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)

Возвращает позицию подстроки str в строке self. Если подстрока не найдена, возвращает –1.

Если подстрока пуста, она считается входящей в любую строку (в том числе в пустую) с позиции 0.

```
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)

Возвращает позицию подстроки str в строке self, начиная с позиции fromIndex. Если подстрока не найдена, возвращает –1.

Если позиция fromIndex превосходит длину строки self, т. е. fromIndex > length_String(self), возвращает –1. Если fromIndex < 0, то fromIndex полагается равным 0. Если подстрока пуста, она считается входящей в любую строку (в том числе в пустую) с позиции fromIndex.

```
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)

Возвращает позицию первого символа ch при просмотре строки справа налево. Если символ не найден, возвращает –1.

Для символа с кодом 0 всегда возвращает –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)

Возвращает позицию первого символа ch при просмотре строки справа налево, начиная с позиции fromIndex. Если символ не найден, возвращает –1.

Если fromIndex < 0, возвращает –1. Если fromIndex превосходит длину строки self, т. е. fromIndex > length_String(self), то позиция принимается за length_String(self). Для символа с кодом 0 всегда возвращает –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)

Возвращает позицию подстроки str при просмотре строки self справа налево. Если подстрока не найдена, возвращает –1.

Если подстрока пуста, она считается входящей в любую строку (в том числе в пустую) с позиции length_String(self).

```
String* s  = create_String("abracadabra");
String* s1 = create_String("abra");
String* s2 = create_String("cdbr");
printf("1) %d\n2) %d\n",
  indexOfString_String(s,s1),
  indexOfString_String(s,s2)
);
```

```
1) 7
2) -1
```

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

Возвращает позицию подстроки str при просмотре строки self справа налево, начиная с позиции fromIndex. Если подстрока не найдена, возвращает –1.

Если fromIndex < 0, возвращает –1. Если fromIndex превосходит длину строки self, т. е. fromIndex > length_String(self), то позиция принимается за length_String(self). Если подстрока пуста, она считается входящей в любую строку (в том числе в пустую) с позиции fromIndex.

```
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)

Возвращает длину строки.

```
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)

Проверяет совпадение фрагмента строки self (длиной len с позиции toffset) с фрагментом строки other (длиной len с позиции ooffset). Функция regionMatchesCase_String учитывает регистр символов, функция regionMatches_String имеет дополнительный параметр ignoreCase (если true, то регистр не учитывается, если false, то учитывается).

Длина фрагментов должна быть неотрицательной (len ≥ 0). Если фрагменты, определяемые позицией и длиной, выходят за границы строки, возвращает 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)

Возвращает строку, полученную из self заменой всех вхождения символа oldChar на символ newChar.

Символы не должны иметь нулевой код.

```
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)

Проверяет, начинается ли строка self строкой prefix.

Если строка prefix пуста, возвращает true. Если длина строки prefix больше, чем длина self, возвращает 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)

Проверяет, начинается ли строка self строкой prefix с позиции offset.

Если позиция отрицательна или превосходит длину строки self, возвращает false. Если строка prefix пуста, возвращает true. Если длина строки prefix больше, чем длина self, возвращает 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)

Возвращает фрагмент строки self с позиции beginIndex до конца.

Позиция beginIndex не должна выходить на пределы строки: 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)

Возвращает фрагмент строки self с позиции beginIndex до позиции endIndex–1 включительно.

Позиция beginIndex не должна быть отрицательной и не должна превосходить endIndex, а позиция endIndex не должна превосходить длину строки: 0 ≤ beginIndex ≤ endIndex ≤ length_String(self). Если начальная и конечная позиции совпадают, возвращает пустую строку.

```
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)

Возвращает C-строку, соответствующую данной спецификационной строке.

Функция возвращает указатель на массив внутри спецификационного объекта, поэтому к возвращенному указателю нельзя применять free() и нельзя использовать его после уничтожения объекта.

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

```
abracadabra
```

String* toLowerCase_String(String* self)

Возвращает строку, полученную из self приведением к нижнему регистру.

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

```
abracadabra
```

String* toUpperCase_String(String* self)

Возвращает строку, полученную из self приведением к верхнему регистру.

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

```
ABRACADABRA
```

String* trim_String(String* self)

Возвращает строку, полученную из self отбрасыванием пробельных символов в начале и конце строки.

Пробельными символами считаются пробелы, табуляции и переводы строк.

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

```
'abracadabra'
```

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

Возвращает спецификационную строку, соответствующую выводу функции printf() с теми же параметрами:

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

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

Два показанных способа создания строки эквивалентны.

String* valueOfBool_String(bool b)

Возвращает строковое представление значения типа bool.

```
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)

Возвращает строковое представление значения типа char.

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

```
a
```

String* valueOfShort_String(short i)

Возвращает строковое представление значения типа short.

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

```
-28
```

String* valueOfUShort_String(unsigned short i)

Возвращает строковое представление значения типа unsigned short.

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

```
47
```

String* valueOfInt_String(int i)

Возвращает строковое представление значения типа int.

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

```
-28
```

String* valueOfUInt_String(unsigned int i)

Возвращает строковое представление значения типа unsigned int.

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

```
47
```

String* valueOfLong_String(long i)

Возвращает строковое представление значения типа long.

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

```
-28
```

String* valueOfULong_String(unsigned long i)

Возвращает строковое представление значения типа unsigned long.

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

```
47
```

String* valueOfFloat_String(float f)

Возвращает строковое представление значения типа float.

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

```
3.140000
```

String* valueOfDouble_String(double d)

Возвращает строковое представление значения типа double.

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

```
3.140000
```

String* valueOfPtr_String(void *p)

Возвращает строковое представление значения типа void*.

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

```
0012FF6C
```

`String* valueOfObject_String( Object* ref )`

Возвращает строковое представление значения спецификационного типа.

Результат совпадает с результатом стандартной функции toString():

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

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

Возвращает строковое шестнадцатеричное представление массива байтов p длины l.

```
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>

Тип List является контейнерным типом, реализующим упорядоченный список элементов.

Элементами списка могут быть любые спецификационные ссылки. При создании списка можно ограничить тип его элементов. Такой список называется *типизированным*, а все функции, работающие с ним и имеющие параметр типа Object*, будут проверять, чтобы тип этого параметра совпадал с типом элементов списка.

Два списка считаются равными, если они имеют одинаковую длину и их элементы попарно равны друг другу. При этом не учитывается типизация списков, в частности, пустые списки всегда равны.

Элементы списка нумеруются с нуля.

List* create_List(const Type *elem_type)

Создает список и возвращает спецификационную ссылку типа List. Если параметр elem_type нулевой, то типы элементов списка не ограничены. В противном случае параметр должен являться указателем на константу-дескриптор типа элементов списка:

```
List* l1 = create_List(NULL);          // любые элементы
List* l2 = create_List(&type_Integer); // элементы только типа Integer
```

Эта функция определена наряду со стандартной функций create():

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

Два показанных способа создания списка эквивалентны.

Type *elemType_List(List* self)

Возвращает указатель на константу-дескриптор спецификационного типа, которым ограничены элементы данного списка.

Если список не типизирован, возвращает NULL.

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

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

Вставляет элемент ref в список на позицию index.

Если список типизирован, то тип элемента ref должен совпадать с типом элементов списка. Номер позиции должен находиться в интервале от 0 до длины списка включительно, т. е. 0 ≤ index ≤ size_List(self). Если номер позиции равен длине списка, то элемент вставляется в конец этого списка.

```
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)

Добавляет элемент ref в конец списка.

Если список типизирован, то тип элемента ref должен совпадать с типом элементов списка.

```
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)

Удаляет из списка все элементы.

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

Тот же результат можно получить, пересоздав список, но функция clear_List() более эффективна:

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

bool contains_List(List* self, Object* ref)

Проверяет, содержит ли список элемент, равный данному.

Если список типизирован, то тип элемента ref должен совпадать с типом элементов списка.

```
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)

Возвращает спецификационную ссылку на элемент в позиции index.

Номер позиции должен находиться в интервале от 0 до длины списка – 1, т. е. 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)

Возвращает номер позиции первого элемента в списке, равного ref.

Если список типизирован, то тип элемента ref должен совпадать с типом элементов списка. Если список не содержит указанного элемента, возвращает –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)

Проверяет список на пустоту.

```
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)

Возвращает номер позиции последнего элемента в списке, равного ref.

Если список типизирован, то тип элемента ref должен совпадать с типом элементов списка. Если список не содержит указанного элемента, возвращает –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)

Удаляет из списка элемент в позиции index.

Номер позиции должен находиться в интервале от 0 до длины списка, т. е. 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)

Заменяет в списке элемент в позиции index на элемент ref.

Если список типизирован, то тип элемента ref должен совпадать с типом элементов списка. Номер позиции должен находиться в интервале от 0 до длины списка включительно, т. е. 0 ≤ index ≤ size_List(self). Если номер позиции равен длине списка, то элемент вставляется в конец этого списка.

```
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)

Возвращает длину списка.

```
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 )`

Возвращает подсписок данного списка, содержащий элементы с позиции fromIndex до toIndex−1.

Позиция fromIndex не должна быть отрицательной и не должна превосходить toIndex, а позиция toIndex не должна превосходить длину списка: 0 ≤ fromIndex ≤ toIndex ≤ size_List(self). Если начальная и конечная позиции совпадают, возвращает пустой список.

```
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)`

Добавляет в список self все элементы списка other, вставляя их с позиции index.

Если список self типизирован, то типы всех элементов списка other должны совпадать с типом элементов списка self (при этом сам список other не обязан быть типизированным). Номер позиции должен находиться в интервале от 0 до длины списка self включительно, т. е. 0 ≤ index ≤ size_List(self). Если номер позиции равен длине списка self, то элементы вставляется в конец этого списка.

```
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)`

Добавляет к концу списка self все элементы списка other.

Если список self типизирован, то типы всех элементов списка other должны совпадать с типом элементов списка self (при этом сам список other не обязан быть типизированным).

```
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)`

Возвращает *множество*, состоящее из всех элементов данного списка.

Возвращаемое множество сохраняет типизацию списка: если элементы списка были ограничены каким-либо типом, то и элементы множества будут ограничены тем же типом.

```
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>

Тип Set является контейнерным типом, реализующим множество элементов.

Элементами множества могут быть любые спецификационные ссылки. При создании множества можно ограничить тип его элементов. Такое множество называется *типизированным*, а все функции, работающие с ним и имеющие параметр типа Object*, будут проверять, чтобы тип этого параметра совпадал с типом элементов множества.

Два множества считаются равными, если они имеют одинаковые элементы. При этом не учитывается типизация множеств, в частности, пустые множества всегда равны.

Set* create_Set(const Type* elem_type)

Создает множество и возвращает спецификационную ссылку типа Set. Если параметр elem_type нулевой, то типы элементов множества не ограничены. В противном случае параметр должен являться указателем на константу-дескриптор типа элементов множества:

```
Set* s1 = create_Set(NULL);          // любые элементы
Set* s2 = create_Set(&type_Integer); // элементы только типа Integer
```

Эта функция определена наряду со стандартной функций create():

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

Два показанных способа создания множества эквивалентны.

Type *elemType_Set(Set* self)

Возвращает указатель на константу-дескриптор спецификационного типа, которым ограничены элементы данного множества.

Если множество не типизировано, возвращает NULL.

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

bool add_Set(Set* self, Object* ref)

Добавляет элемент ref в множество.

Если множество типизировано, то тип элемента ref должен совпадать с типом элементов множества.

```
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)

Удаляет элемент из множества.

Если множество типизировано, то тип элемента ref должен совпадать с типом элементов множества.

```
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)

Удаляет из множества все элементы.

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

Тот же результат можно получить, пересоздав множество, но функция clear_Set() более эффективна:

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

bool contains_Set(Set* self, Object* ref)

Проверяет, содержит ли множество элемент, равный данному.

Если множество типизировано, то тип элемента ref должен совпадать с типом элементов множества.

```
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)

Проверяет множество на пустоту.

```
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)

Возвращает количество элементов в множестве.

```
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)

Возвращает спецификационную ссылку на элемент с номером index.

Эта функция служит для перебора всех элементов множества. Поскольку множество неупорядочено, элементы нумеруются некоторым произвольным образом. Номер позиции

должен находиться в интервале от 0 до размера множества – 1, т. е. 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)) ...               // истинно
```

bool containsAll_Set(Set* self, Set* set)

Проверяет, является ли множество set подмножеством self.

Иными словами, проверяет, входят ли все элементы множества set в множество self.

```
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)

Объединяет множество self с множеством set, возвращает true, если в результате работы множество self изменилось, и false — если нет.

Иными словами, добавляет все элементы множества set в множество self. Если множество self типизировано, то типы всех элементов множества set должны совпадать с типом элементов множества self (при этом само множество set не обязано быть типизированным).

```
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)

Пересекает множество self с множеством set, возвращает true, если в результате работы множество self изменилось, и false — если нет.

Иными словами, оставляет в множестве self только те элементы, которые есть в множестве 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)

Вычитает из множества self множество set, возвращает true, если в результате работы множество self изменилось, и false — если нет.

Иными словами, оставляет в множестве self только те элементы, которых нет в множестве 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)

Возвращает *список*, состоящий из всех элементов данного множества.

Порядок элементов в списке не определен. Возвращаемый список сохраняет типизацию множества: если элементы множества были ограничены каким-либо типом, то и элементы списка будут ограничены тем же типом.

```
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>

Тип Map является контейнерным типом, реализующим отображение элементов из *области определения* в *область значений*. Элемент из области определения называется *ключом*, а соответствующий элемент из области значений — просто *значением*. При этом одному ключу соответствует ровно одно значение.

Элементами отображения могут быть любые спецификационные ссылки. При создании отображения можно ограничить тип его элементов, причём отдельно для элементов области определения и для элементов области значений. Такое отображение называется *типизированным*, а все функции, работающие с ним и имеющие параметр типа Object*, будут проверять, чтобы тип этого параметра совпадал с типом элементов соответствующей области.

Два отображения считаются равными, если они имеют одинаковые множества ключей, и каждому ключу в обоих отображениях соответствуют одинаковые значения. При этом не учитывается типизация отображений, в частности, пустые отображения всегда равны.

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

Создает отображение и возвращает спецификационную ссылку типа Map. Если параметр key_type нулевой, то типы элементов области определения не ограничены. В противном случае параметр должен являться указателем на константу-дескриптор типа для ключей отображения. Аналогично, если параметр val_type нулевой, то типы элементов области значений не ограничены. Иначе параметр должен являться указателем на константу-дескриптор типа для значений отображения:

```
Map* m1 = create_Map(NULL, NULL);          // любые элементы
Map* m2 = create_Map(&type_Integer, NULL); // ключи только типа Integer
// отображение из Integer в String
Map* m3 = create_Map(&type_Integer, &type_String);
```

Эта функция определена наряду со стандартной функций create():

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

Два показанных способа создания отображения эквивалентны.

Type *keyType_Map(Map* self)

Возвращает указатель на константу-дескриптор спецификационного типа, которым ограничены ключи данного отображения.

Если область определения отображения не типизирована, возвращает NULL.

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

Type *valueType_Map(Map* self)

Возвращает указатель на константу-дескриптор спецификационного типа, которым ограничены значения данного отображения.

Если область значений отображения не типизирована, возвращает NULL.

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

void clear_Map(Map* self)

Удаляет из отображения все элементы.

```
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);                                     //
```

Тот же результат можно получить, пересоздав отображение, но функция clear_Map() более эффективна:

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

bool containsKey_Map(Map* self, Object* key)

Проверяет, содержит ли отображение ключ, равный данному.

Если область определения типизирована, то тип элемента key должен совпадать с типом ключей отображения.

```
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)

Проверяет, содержит ли отображение значение, равное данному.

Если область значений типизирована, то тип элемента value должен совпадать с типом значений отображения.

```
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)

Возвращает спецификационную ссылку на значение, соответствующее ключу.

Если область определения типизирована, то тип элемента key должен совпадать с типом ключей отображения. Если отображение не содержит данного ключа, возвращает 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)

Возвращает спецификационную ссылку на какой-нибудь ключ, которому соответствует данное значение.

Если область значений типизирована, то тип элемента value должен совпадать с типом значений отображения. Если отображение не содержит ни одного ключа, которому соответствует данной значение, возвращает NULL. В противном случае возвращает какой-нибудь подходящий ключ.

```
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);                          // или 28, или 47
if (equals( get_Map(m,key), val)) ...             // истинно
```

bool isEmpty_Map(Map* self)

Проверяет отображение на пустоту.

Возвращает true, если отображение не содержит ни одной пары «ключ–значение»; в противном случае возвращает 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)

Добавляет в отображение пару «ключ–значение».

Если область определения типизирована, то тип элемента key должен совпадать с типом ключей отображения. Если область значений типизирована, то тип элемента value должен совпадать с типом значений отображения.

Если отображение не содержит ключа key, то в отображение добавляется ключ key и соответствующее ему значение value; функция возвращает NULL. Если же отображение уже содержит ключ key, то функция возвращает соответствующее ему старое значение, а в отображении старое значение заменяется новым (value).

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

void putAll_Map(Map* self, Map* t)

Добавляет в отображение self все пары «ключ–значение» из отображения t.

Если область определения отображения self типизирована, то типы всех ключей отображения t должны совпадать с типом ключей отображения self. Аналогично, если область значений отображения self типизирована, то типы всех значений отображения t должны совпадать с типом значений отображения self. (При этом не требуется, чтобы области определения и значений отображения t были типизированы.)

Если отображение self уже содержит некоторый ключ из отображения t, то соответствующее ему значение заменяется на новое.

```
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);`

Функция remove_Map() удаляет пару ключ-значение для данного ключа, если таковая присутствует в отображении self. Возвращает значение, которое соответствовало ключу key в отображении self, или NULL, если такого значения не было.

```
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;
```

`int size_Map( Map* self )`

Возвращает размер отображения.

Размером считается количество пар «ключ–значение», содержащихся в отображении.

```
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 )`

Возвращает ключ отображения в позиции index.

Эта функция служит для перебора всех ключей отображения. Поскольку множество ключей неупорядочено, элементы нумеруются некоторым произвольным образом. Номер позиции должен находиться в интервале от 0 до размера отображения – 1, т. е. 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)) ...               // истинно
```

Грамматика SeC

```
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:
 *    se_storage_class_specifier has been added.
 */
storage_class_specifier ::=   "typedef"
                            | "extern"
                            | "static"
                            | "auto"
                            | "register"
                            | se_storage_class_specifier
                        ;

se_storage_class_specifier ::= "stable" ;

type_specifier ::=   "void"
                   | "char"
                   | "short"
                   | "int"
                   | "long"
                   | "float"
                   | "double"
                   | "signed"
                   | "unsigned"
                   | "_Bool"
                   | "_Complex"
                   | "_Imaginary"
                   | struct_or_union_specifier
                   | enum_specifier
                   | typedef_name
               ;

/*
 * 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" ;

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:
 *  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:
 *   GCC attributes optional list has been added.
 */
declarator ::= ( pointer )? 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:
 *   GCC attributes 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
 */

statement ::=   labeled_statement
              | compound_statement
              | expression_statement
              | selection_statement
              | iteration_statement
              | jump_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 ;


/*
 * 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__" ;
```