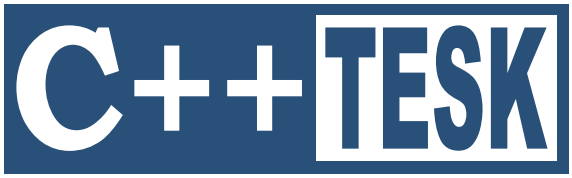

C++TESK Hardware Edition: **Краткий справочник**

Версия 1.0, 21/07/2011

Инструмент C++TESK Hardware Edition входит в состав набора инструментов C++TESK Testing ToolKit, который доступен для скачивания на странице http://forge.ispras.ru/projects/cpptesk-toolkit.

Вопросы по использованию C++TESK Hardware Edition и C++TESK Testing ToolKit в целом, а также описание обнаруженных проблем в инструментах и документации к ним отправляйте по адресу cpptesk-support@ispras.ru. Для этого также можно использовать форум http://hw-forum.ispras.ru.

Содержание

Введение 5
Поддержка устаревших возможностей 5
Соглашения по именованию 6
Создание эталонной модели 6
Класс сообщения 7
Рандомизатор входного сообщения 7
Компаратор выходного сообщения 7
Поля данных сообщения 8
Эталонная модель 8
Интерфейс 9
Процесс 9
Параметры процесса 10
Приоритет процесса 10
Моделирование задержек 11
Запуск процесса 11
Получение стимула 12
Посылка реакции 12
Операция 12
Функции обратного вызова 13
Функция onEveryCycle 13
Создание адаптера эталонной модели 13
Адаптер эталонной модели 13
Синхронизатор 14
Адаптер входного интерфейса 14
Адаптер выходного интерфейса 15
Прослушиватель выходного интерфейса 16
Арбитр реакций 16
Описание тестового покрытия 17
Класс тестового покрытия 17
Структура тестового покрытия 18
Перечислимое покрытие 18
Произведение покрытий 18
Произведение покрытий с исключением 19
Синоним покрытия 19

Функция вычисления тестовой ситуации 20
Функция трассировки тестовой ситуации 20
Создание тестового сценария 21
Сценарный класс 21
Метод инициализации тестового сценария 21
Метод завершения тестового сценария 21
Сценарный метод 22
Доступ к итерационным переменным 22
Блок тестового воздействия 23
Выход из сценарного метода 23
Временные задержки 23
Метод вычисления состояния 24
Запуск тестового сценария 24
Вспомогательные возможности 25
Утверждения 25
Отладочная печать 25
Макросы отладочной печати 25
Вывод стека вызовов процессов 26
Выделение отладочной печати цветом 26
Уровни отладочной печати 27
Настройка отладочной печати 27

Введение

Документ является кратким справочником по инструменту C++TESK Hardware Edition (далее C++TESK), входящему в состав набора инструментов C++TESK Testing ToolKit и предназначенному для автоматизированной разработки тестовых систем для HDL-моделей[1] аппаратуры. Инструмент построен в соответствии с общей концепцией UniTESK[2].

Под *тестовой системой* понимается специальная программа, которая путем подачи на *целевую систему* воздействий и анализа выдаваемых ей результатов оценивает ее функциональную корректность. Воздействия на тестируемую модель также называются *стимулами*, а результаты ее работы — *реакциями*. Типичная тестовая система состоит из трех основных компонентов: (1) *генератора стимулов*, создающего последовательности стимулов (*тестовые последовательности*), (2) *тестового оракула*, проверяющего правильность реакций, и (3) *подсистемы оценки полноты тестирования*.

Документ описывает возможности C++TESK, предназначенные для создания указанных выше компонентов тестовых систем, и включает четыре основных раздела:

- «*Создание эталонной модели*»
- «*Создание адаптера эталонной модели*»
- «*Описание тестового покрытия*»
- «*Создание тестового сценария*»

Первые два раздела описывают построение основополагающих частей тестового оракула: *эталонной модели*, задающей функциональность целевой системы, и *адаптера эталонной модели*, связывающего эталонную модель с целевой системой. Третий раздел посвящен оценке полноты тестирования на основе *тестового покрытия*. Последний раздел рассказывает о создании *тестового сценария* — основной части генератора стимулов.

Более подробное описание инструмента доступно в документе «*C++TESK Testing ToolKit: Руководство пользователя*».

Порядок установки C++TESK описан в отдельном документе «*C++TESK Testing ToolKit: Инструкция по установке*».

Поддержка устаревших возможностей

При развитии C++TESK обеспечивается совместимость новых версий инструмента с тестовыми системами, разработанными с помощью более старых версий. При внесении в инструмент изменения, несовместимого с предыдущими версиями, этому изменению сопоставляется номер сборки в формате `rгммдд`, например, `110415` — 15 апреля 2011 г. Новая правка делается доступной только при надлежащем определении макроса `CPPTESKHW_VERSION`[3]. Например, `-DCPPTESKHW_VERSION=110415`, включает поддержку несовместимых изменений, сделанных до 15 апреля 2011 г. включительно. Каждая сборка инструмента содержит полный список таких изменений.

В документации не описываются устаревшие возможности инструмента. Для тех средств, которые не совместимы с устаревшими возможностями, указывается номер сборки, начиная с которой они поддерживаются. Например, фраза «*средства поддерживаются, начиная со*

---

[1] HDL (Hardware Description Language) — класс языков, используемых для описания аппаратуры.

[2] http://www.unitesk.ru

[3] При использовании компиляторов gcc это делается опцией `-D`*`имя_макроса=значение`*.

сборки 110415» означает, что для возможности использования этих средств необходимо, чтобы были выполнены два условия: (1) сборка инструмента имеет номер 110415 или выше, (2) при компиляции тестовой системы указана опция `-DCPPTESKHW_VERSION=`*`сборка`*, где *`сборка`* не меньше 110415.

Если некоторые устаревшие возможности инструмента не используются или мешают развитию инструмента, разработчики могут отказаться от их поддержки. В этом случае при компиляции тестовой системы, использующей эти возможности, будет выдаваться сообщение, описывающее способ устранения проблемы.

При компиляции тестовых систем, разработанных с помощью C++TESK, **настоятельно рекомендуется** использовать опцию компилятора `-DCPPTESKHW_VERSION=`*`сборка`*.

Соглашения по именованию

Ядро инструмента реализовано в форме библиотеки на языке программирования C++. Доступные для использования средства сгруппированы по следующим пространствам имен:

- `cpptesk::hw` — средства создания эталонных моделей аппаратуры и их адаптеров;
- `cpptesk::ts` — базовые средства разработки тестовых систем;
- `cpptesk::ts::coverage` — средства описания тестового покрытия;
- `cpptesk::ts::engine` — библиотека обходчиков[4];
- `cpptesk::tracer` — средства трассировки;
- `cpptesk::tracer::coverage` — средства трассировки покрытия.

Многие средства C++TESK реализованы в форме макросов. Для избежания конфликтов имен названия всех макросов имеют префикс `CPPTESK_`, например, `CPPTESK_MODEL(`*`имя`*`)`. В общем случае для каждого макроса определены два дополнительных альтернативных имени: *сокращенное имя*, получаемое отбрасыванием префикса `CPPTESK_`, и *короткое имя*, получаемое дополнительным «сжатием» сокращенного имени. Например, для макроса `CPPTESK_ITERATION_BEGIN` доступны два альтернативных имени: `ITERATION_BEGIN` и `IBEGIN`. Чтобы иметь возможность использовать сокращенные и короткие имена, следует определить макросы `CPPTESK_SHORT_NAMES` и `CPPTESK_SHORT_SHORT_NAMES` соответственно.

Создание эталонной модели

Эталонной моделью называется определенным образом структурированный набор классов, описывающий функциональность целевой системы на некотором уровне абстракции[5]. Эталонная модель состоит из *классов сообщений*, описывающих формат входных и выходных данных (структуру стимулов и реакций), *основного класса* и набора вспомогательных классов. В дальнейшем под *эталонной моделью* понимается основной класс эталонной модели.

[4] Обходчиками называются библиотечные компоненты инструмента, используемые для построения тестовой последовательности. Входной информацией для обходчика является тестовый сценарий (см. раздел «*Создание тестового сценария*»).

[5] Инструмент позволяет разрабатывать как абстрактные функциональные модели, так и детальные модели, описывающие целевую систему с потактовой точностью.

Класс сообщения

Классы сообщений декларируются с помощью макроса `CPPTESK_MESSAGE(`*имя*`)`. Внутри декларации должна присутствовать строчка `CPPTESK_SUPPORT_CLONE(`*имя*`)`, определяющая метод копирования сообщения *имя*`* clone()`.

Пример:

```
#include <hw/message.hpp>

CPPTESK_MESSAGE(MyMessage) {
public:
    CPPTESK_SUPPORT_CLONE(MyMessage)
    ...
};
```

Замечание: наличие строки `CPPTESK_SUPPORT_CLONE(`*имя*`)` обязательно.

Рандомизатор входного сообщения

В классе входного сообщения должен перегружаться виртуальный метод `randomize()` — *рандомизатор* сообщения. Для этого можно использовать специальные макросы `CPPTESK_{DECLARE|DEFINE}_RANDOMIZER`.

Пример:

```
CPPTESK_MESSAGE(MyMessage) {
    ...
    CPPTESK_DECLARE_RANDOMIZER();
private:
    int data;
};

CPPTESK_DEFINE_RANDOMIZER(MyMessage) {
    data = CPPTESK_RANDOM(int);
}
```

Замечание: рандомизатор можно не определять, если поля данных сообщения описаны с помощью специальных макросов (см. раздел «*Поля данных сообщения*»).

Компаратор выходного сообщения

В классе выходного сообщения должен перегружаться виртуальный метод `compare()` — *компаратор* сообщения. Для этого можно использовать специальные макросы (сборка не ниже 110428) `CPPTESK_{DECLARE|DEFINE}_COMPARATOR`.

Пример:

```
CPPTESK_MESSAGE(MyMessage) {
    ...
    CPPTESK_DECLARE_COMPARATOR();
private:
    int data;
};

CPPTESK_DEFINE_COMPARATOR(MyMessage) {
    const MyMessage &rhs = CPPTESK_CONST_CAST_MESSAGE(MyMessage);
    // в случае несовпадения сообщений возвращается непустая строка
    if(data != rhs.data)
        { return "incorrect data"; }
    // пустая строка интерпретируется как отсутствие ошибок
    return "";
}
```

Замечание: компаратор можно не определять, если поля данных сообщения описаны с помощью специальных макросов (см. раздел «*Поля данных сообщения*»).

Поля данных сообщения

Для декларации целочисленных *полей данных* сообщения предназначены следующие макросы:

- `CPPTESK_DECLARE_FIELD(`*`имя, разрядность`*`)`
  декларирует целочисленное поле с заданным именем и разрядностью.
- `CPPTESK_DECLARE_MASKED_FIELD(`*`имя, разрядность, маска`*`)`
  декларирует целочисленное поле с заданным именем, разрядностью и маской.
- `CPPTESK_DECLARE_BIT(`*`имя`*`)`
  декларирует однобитовое поле с заданным именем.

Пример:

```
#include <hw/message.hpp>

CPPTESK_MESSAGE(MyMessage) {
public:
    CPPTESK_DECLARE_MASKED_FIELD(addr, 32, 0xffffFFF0);
    CPPTESK_DECLARE_FIELD(data, 32);
    CPPTESK_DECLARE_BIT(flag);
    ...
};
```

Замечание: разрядность полей данных не должна превышать 64 бита.

Все задекларированные поля должны быть зарегистрированы в конструкторе класса сообщения с помощью макроса `CPPTESK_ADD_FIELD(`*`полное_имя`*`)`.

Пример:

```
MyMessage::MyMessage() {
    CPPTESK_ADD_FIELD(MyMessage::addr);
    CPPTESK_ADD_FIELD(MyMessage::data);
    CPPTESK_ADD_FIELD(MyMessage::flag);
    ...
}
```

Замечание: при регистрации полей указывается их полное имя, включающее имя класса сообщения.

Эталонная модель

Эталонная модель (*основной класс эталонной модели*) декларируется с помощью макроса `CPPTESK_MODEL(`*`имя`*`)`.

Пример:

```
#include <hw/model.hpp>

CPPTESK_MODEL(MyModel) {
    ...
};
```

В эталонной модели декларируются *входные* и *выходные интерфейсы*, *операции*, а также вспомогательные *процессы* и данные, необходимые для описания операций.

Интерфейс

Входные и выходные интерфейсы эталонной модели декларируются с помощью макросов `CPPTESK_DECLARE_{INPUT|OUTPUT}(`*имя*`)` соответственно.

Пример:

```
#include <hw/model.hpp>

CPPTESK_MODEL(MyModel) {
public:
    CPPTESK_DECLARE_INPUT(input_iface);
    CPPTESK_DECLARE_OUTPUT(output_iface);
    ...
};
```

Все задекларированные интерфейсы должны быть зарегистрированы в конструкторе эталонной модели с помощью макросов `CPPTESK_ADD_{INPUT|OUTPUT}(`*имя*`)`.

Пример:

```
MyModel::MyModel() {
    CPPTESK_ADD_INPUT(input_iface);
    CPPTESK_ADD_OUTPUT(output_iface);
    ...
}
```

Процесс

Процессы являются основным средством функциональной спецификации аппаратуры. Процессы подразделяются на *операции* (см. раздел «*Операция*») и *внутренние процессы*. Операции описывают логику работы целевой системы, связанную с обработкой стимулов определенного типа. Внутренние процессы используются для определения других, более сложных процессов, в том числе операций.

Декларация и определение процессов эталонной модели осуществляется с помощью макросов `CPPTESK_{DECLARE|DEFINE}_PROCESS(`*имя*`)`. Определение процесса должно начинаться вызовом макроса `CPPTESK_START_PROCESS()`, а заканчиваться вызовом макроса `CPPTESK_STOP_PROCESS()`.

Пример:

```
#include <hw/model.hpp>

CPPTESK_MODEL(MyModel) {
public:
    ...
    CPPTESK_DECLARE_PROCESS(internal_process);
    ...
};

CPPTESK_DEFINE_PROCESS(MyModel::internal_process) {
    CPPTESK_START_PROCESS();
    ...
    CPPTESK_STOP_PROCESS();
}
```

Замечание: при определении процесса допускается только одно использование макроса `CPPTESK_START_PROCESS()`, которое обычно предшествует описанию логики процесса. Семантика `CPPTESK_STOP_PROCESS()` схожа с семантикой оператора `return` — при выполнении этого макроса осуществляется выход из метода процесса.

Замечание: запрещается называть процесс именем `process`, поскольку это имя зарезервировано.

Параметры процесса

Процесс имеет не менее трех *параметров*: (1) *контекст* выполнения процесса, (2) *интерфейс*, ассоциированный с процессом, и (3) *сообщение*. Для доступа к параметрам процесса используются следующие макросы:

- `CPPTESK_GET_PROCESS()` — получение контекста;
- `CPPTESK_GET_IFACE()` — получение интерфейса;
- `CPPTESK_GET_MESSAGE()` — получение сообщения.

Приведение параметра-сообщения к нужному типу осуществляется с помощью макросов:

- `CPPTESK_CAST_MESSAGE(`*`класс_сообщения`*`);`
- `CPPTESK_CONT_CAST_MESSAGE(`*`класс_сообщения`*`).`

Пример:

```
#include <hw/model.hpp>

CPPTESK_DEFINE_PROCESS(MyModel::internal_process) {
    // копирование сообщения в локальную переменную
    MyMessage msg = CPPTESK_CAST_MESSAGE(MyMessage);
    // получение ссылки на сообщение
    MyMessage &msg_ref = CPPTESK_CAST_MESSAGE(MyMessage);
    // получение константной ссылки на сообщение
    const MyMessage &const_msg_ref = CPPTESK_CONST_CAST_MESSAGE(MyMessage);
    ...
}
```

Приоритет процесса

При выполнении каждому процессу эталонной модели сопоставляется *приоритет* — целое беззнаковое число, принимающее значения из отрезка [1, 255] (значение 0 зарезервировано). Приоритет влияет на очередность выполнения процессов внутри одного такта (сначала выполняются процессы с бо́льшим приоритетом, затем — с меньшим). Приоритеты могут использоваться при сопоставлении реакций целевой системы с эталонными реакциями (см. раздел «*Арбитр реакций*»). При запуске все процессы получают одинаковый приоритет (`NORMAL_PRIORITY`). Для работы с приоритетами используются следующие макросы:

- `CPPTESK_GET_PRIORITY()` — получение приоритета процесса.
- `CPPTESK_SET_PRIORITY(`*`приоритет`*`)` — установка приоритета процесса.

Некоторые константы для значений приоритета определены в перечислимом типе `priority_t` (пространство имен `cpptesk::hw`). Основными из них являются:

- `NORMAL_PRIORITY` — нормальный приоритет;
- `LOWEST_PRIORITY` — минимальный приоритет;
- `HIGHEST_PRIORITY` — максимальный приоритет.

Пример:

```
#include <hw/model.hpp>
#include <iostream>

CPPTESK_DEFINE_PROCESS(MyModel::internal_process) {
    ...
    std::cout << "process priority is " << std::dec
              << CPPTESK_GET_PRIORITY() << std::end;
    ...
```

```
    CPPTESK_SET_PRIORITY(cpptesk::hw::HIGHEST_PRIORITY);
    ...
}
```

Моделирование задержек

Для моделирования *временных задержек* в описании процессов можно использовать следующие макросы:

- `CPPTESK_CYCLE()`
  задержка в один такт.
- `CPPTESK_DELAY(`*`число_тактов`*`)`
  задержка в несколько тактов.
- `CPPTESK_WAIT(`*`условие`*`)`
  задержка до выполнения условия.
- `CPPTESK_WAIT_TIMEOUT(`*`условие`*`, `*`таймаут`*`)`
  ограниченная по времени задержка до выполнения условия.

Пример:

```
#include <hw/model.hpp>
#include <iostream>

CPPTESK_DEFINE_PROCESS(MyModel::internal_process) {
    ...
    std::cout << "cycle: " << std::dec << time() << std::end;
    // задержка в один такт
    CPPTESK_CYCLE();
    std::cout << "cycle: " << std::dec << time() << std::end;
    ...
    // ожидание выполнения условия outputs.ready,
    // но не более 100 тактов
    CPPTESK_WAIT_TIMEOUT(outputs.ready, 100);
    ...
}
```

Запуск процесса

Запуск процесса эталонной модели из другого процесса осуществляется с помощью макроса `CPPTESK_CALL_PROCESS(`*`режим`*`, `*`имя_процесса`*`, `*`интерфейс`*`, `*`сообщение`*`)`, где *`режим`* имеет одно из двух значений: `PARALLEL` или `SEQUENTIAL`. В первом случае создается отдельный процесс, выполняемый параллельно родительскому процессу. Во втором случае осуществляется последовательное выполнение: сначала выполняется запускаемый процесс, а после его завершения возобновляется работа родительского процесса.

Пример:

```
#include <hw/model.hpp>

CPPTESK_DEFINE_PROCESS(MyModel::some_process) {
    ...
    // запуск параллельного процесса
    CPPTESK_CALL_PROCESS(PARALLEL, internal_process,
        CPPTESK_GET_IFACE(), CPPTESK_GET_MESSAGE());
    ...
    // запуск процесса и ожидание его завершения
    CPPTESK_CALL_PROCESS(SEQUENTIAL, internal_process,
        CPPTESK_GET_IFACE(), CPPTESK_GET_MESSAGE());
    ...
}
```

Замечание: создавать *новые* процессы можно из любых методов эталонной модели, а не только из методов, описывающих процессы. Для этого следует использовать специальный макрос `CPPTESK_CALL_PARALLEL(`*`имя_процесса`*`, `*`интерфейс`*`, `*`сообщение`*`)`. Запуск процесса в режиме `SEQUENTIAL` из метода, не являющегося процессом, запрещен.

Получение стимула

Внутри процесса можно моделировать *получение стимула* по одному из входных интерфейсов. Для этого используется макрос `CPPTESK_RECV_STIMULUS(`*`режим`*`, `*`интерфейс`*`, `*`сообщение`*`)`. При его выполнении тестовая система подает стимул на целевую систему, запуская адаптер соответствующего входного интерфейса (см. раздел «*Адаптер входного интерфейса*»). Семантика параметра *`режим`* описана в разделе «*Запуск процесса*».

Пример:

```
#include <hw/model.hpp>

CPPTESK_DEFINE_PROCESS(MyModel::some_process) {
    // моделирование получения стимула
    CPPTESK_RECV_STIMULUS(PARALLEL, input_iface, input_msg);
    ...
}
```

Посылка реакции

Моделирование *посылки реакции* процессом осуществляется с помощью макроса `CPPTESK_SEND_REACTION(`*`режим`*`, `*`интерфейс`*`, `*`сообщение`*`)`. При его выполнении тестовая система запускает адаптер соответствующего выходного интерфейса, который запускает цикл ожидания реакции целевой системы, а после получения реакции и преобразует ее в форму объекта соответствующего класса сообщения (см. раздел «*Адаптер выходного интерфейса*»). После этого тестовая система сравнивает эталонное значение сообщения с полученным, используя компаратор (см. раздел «*Компаратор выходного сообщения*»). Семантика параметра *`режим`* описана в разделе «*Запуск процесса*».

Пример:

```
#include <hw/model.hpp>

CPPTESK_DEFINE_PROCESS(MyModel::some_process) {
    // моделирование посылки реакции
    CPPTESK_SEND_REACTION(SEQUENTIAL, output_iface, output_msg);
    ...
}
```

Операция

Декларация и определение *интерфейсных операций* эталонной модели осуществляется с помощью макросов `CPPTESK_{DECLARE|DEFINE}_STIMULUS(`*`имя`*`)`. Определение должно начинаться с макроса `CPPTESK_START_STIMULUS(`*`режим`*`)`, а заканчиваться `CPPTESK_STOP_STIMULUS()`.

Пример:

```
#include <hw/model.hpp>

CPPTESK_MODEL(MyModel) {
public:
    CPPTESK_DECLARE_STIMULUS(operation);
    ...
};
```

```
CPPTESK_DEFINE_STIMULUS(MyModel::operation) {
    CPPTESK_START_STIMULUS(PARALLEL);
    ...
    CPPTESK_STOP_STIMULUS();
}
```

Замечание: операции являются частным случаем процессов, поэтому в них можно использовать все конструкции, описанные в разделе «*Процесс*».

Замечание: вызов макроса `CPPTESK_START_STIMULUS(`*`режим`*`)` равносилен вызову макроса `CPPTESK_RECV_STIMULUS(`*`режим`*`, ...)`, которому в качестве интерфейса и сообщения передаются соответствующие параметры операции.

Функции обратного вызова

В базовом классе эталонной модели определены несколько *функций обратного вызова* (*callbacks*), которые можно перегружать в эталонной модели. Основной функцией такого типа является `onEveryCycle()`.

Функция onEveryCycle

Функция `onEveryCycle()` вызывается в начале каждого такта работы эталонной модели.

Пример:

```
#include <hw/model.hpp>

CPPTESK_MODEL(MyModel) {
public:
    virtual void onEveryCycle();
    ...
};

void MyModel::onEveryCycle() {
    std::cout << "onEveryCycle: time=" << std::dec << time() << std::endl;
}
```

Создание адаптера эталонной модели

Адаптером эталонной модели (*медиатором*) называется компонент тестовой системы, осуществляющий связывание эталонной модели с целевой системой. Адаптер эталонной модели осуществляет *сериализацию* объектов входных сообщений в последовательность значений входных сигналов, *десериализацию* последовательности значений выходных сигналов в объекты выходных сообщений и *сопоставление* полученных от целевой системы реакций с эталонными значениями.

Адаптер эталонной модели

Адаптер эталонной модели представляет собой подкласс класса эталонной модели. Для его декларации используется макрос `CPPTESK_ADAPTER(`*`имя_адаптера, имя_модели`*`)`.

Пример:

```
#include <hw/media.hpp>

CPPTESK_ADAPTER(MyAdapter, MyModel) {
    ...
};
```

В адаптере эталонной модели декларируются *методы синхронизации*, *адаптеры входных* и *выходных интерфейсов*, *прослушиватели выходных интерфейсов* и *арбитры реакций*.

Синхронизатор

Синхронизатором называется низкоуровневая часть адаптера эталонной модели, которая отвечает за синхронизацию тестовой системы с тестируемой HDL-моделью. Реализация синхронизатора заключается в перегрузке пяти методов адаптера:

- `void initialize()` — инициализация тестовой системы;
- `void finialize()` — завершение тестовой системы;
- `void setInputs()` — синхронизация входов;
- `void getOutputs()` — синхронизация выходов;
- `void simulate()`— синхронизация времени.

При верификации Verilog-моделей аппаратуры данные методы могут быть реализованы с использованием интерфейса VPI (Verilog Procedural Interface). Для автоматизации построения синхронизатора может быть также использована утилита VeriTool[6]. В этом случае для упрощения декларации адаптера эталонной модели можно использовать макрос `CPPTESK_VERITOOL_ADAPTER(`*`имя_адаптера`*`, `*`имя_модели`*`)`.

Пример:

```
#include <hw/veritool/media.hpp>
// файл, генерируемый утилитой VeriTool
#include <interface.h>

CPPTESK_VERITOOL_ADAPTER(MyAdapter, MyModel) {
    ...
};
```

Функции и структуры данных (поля `inputs` и `outputs`), используемые для определения методов синхронизации, генерируются автоматически утилитой VeriTool на основе анализа интерфейса Verilog-модели.

Замечание: при использовании макроса `CPPTESK_VERITOOL_ADAPTER` декларировать поля `inputs` и `outputs` и перегружать методы синхронизатора и декларировать в классе поля **не требуется**.

Адаптер входного интерфейса

Адаптером входного интерфейса называется процесс, определенный в адаптере эталонной модели и сопоставленный одному из входных интерфейсов. Адаптер входного интерфейса запускается при вызове макроса `CPPTESK_START_STIMULUS(`*`режим`*`)` (см. раздел «*Операция*») или `CPPTESK_RECV_STIMULUS(`*`режим`*`, `*`интерфейс`*`, `*`сообщение`*`)` (см. раздел «*Получение стимула*»). Декларация и определение адаптера входного интерфейса осуществляется обычным для процесса образом.

Следует обратить внимание на то, что в адаптере входного интерфейса непосредственно перед началом сериализации необходимо *захватить* интерфейс. Это делается с помощью макроса `CPPTESK_CAPTURE_IFACE()`. Соответственно в конце сериализации интерфейс должен быть освобожден. Для этого используется парный макрос `CPPTESK_RELEASE_IFACE()`.

Пример:

```
#include <hw/media.hpp>
```

[6] http://forge.ispras.ru/projects/veritool

```
CPPTESK_ADAPTER(MyAdapter, MyModel) {
    CPPTESK_DECLARE_PROCESS(serialize_input);
    ...
};

CPPTESK_DEFINE_PROCESS(MyAdapter::serialize_input) {
    MyMessage msg = CPPTESK_CAST_MESSAGE(MyMessage);
    // начало процесса сериализации
    CPPTESK_START_PROCESS();
    // захват входного интерфейса
    CPPTESK_CAPTURE_IFACE();
    // установка строба запуска операции
    inputs.start = 1;
    // установка информационных сигналов
    inputs.addr  = msg.get_addr();
    inputs.data  = msg.get_data();
    // задержка в один такт
    CPPTESK_CYCLE();
    // сброс строба запуска операции
    inputs.start = 0;
    // освобождение входного интерфейса
    CPPTESK_RELEASE_IFACE();
    // завершение процесса сериализации
    CPPTESK_STOP_PROCESS();
}
```

Сопоставление адаптера с интерфейсом осуществляется в конструкторе с помощью макроса CPPTESK_SET_INPUT_ADAPTER(*имя_интерфейса*, *полное_имя_адаптера*).

Пример:

```
MyAdapter::MyAdapter{
    CPPTESK_SET_INPUT_ADAPTER(input_iface, MyAdater::serialize_input);
    ...
};
```

Замечание: при регистрации адаптера входного интерфейса указывается его полное имя, включающее имя класса адаптера эталонной модели.

Адаптер выходного интерфейса

Адаптером выходного интерфейса называется процесс, определенный в адаптере эталонной модели и сопоставленный одному из выходных интерфейсов. Адаптер выходного интерфейса запускается при вызове макроса CPPTESK_SEND_REACTION(*режим*, *интерфейс*, *сообщение*) (см. раздел «*Посылка реакции*»). При определении адаптера выходного интерфейса можно использовать следующие макросы:

- CPPTESK_WAIT_REACTION(*условие*)
  ожидание реакции и разрешения доступа к ней от арбитра реакций;
- CPPTESK_NEXT_REACTION()
  освобождение арбитра реакций (см. раздел «*Арбитр реакций*»).

Пример:

```
#include <hw/media.hpp>

CPPTESK_ADAPTER(MyAdapter, MyModel) {
    CPPTESK_DECLARE_PROCESS(deserialize_output);
    ...
};

CPPTESK_DEFINE_PROCESS(MyAdapter::deserialize_input) {
    // получение ссылки на объект сообщения
```

```
        MyMessage &msg = CPPTESK_CAST_MESSAGE(MyMessage);
        // начало процесса десериализации
        CPPTESK_START_PROCESS();
        // ожидание выставления строба результата
        CPPTESK_WAIT_REACTION(outputs.result);
        // считывание информационных сигналов
        msg.set_data(outputs.data);
        // освобождение арбитра реакций
        CPPTESK_NEXT_REACTION();
        // завершение процесса десериализации
        CPPTESK_STOP_PROCESS();
    }
```

Максимальное время ожидания реакции на выходах целевой системы задается с помощью макроса CPPTESK_SET_REACTION_TIMEOUT(*таймаут*), который также как и макрос CPPTESK_SET_OUTPUT_ADAPTER(*имя_интерфейса*, *полное_имя_адаптера*) вызывается в конструкторе адаптера эталонной модели.

Пример:

```
MyAdapter::MyAdapter{
    CPPTESK_SET_OUTPUT_ADAPTER(output_iface, MyAdater::deserialize_output);
    CPPTESK_SET_REACTION_TIMEOUT(100);
    ...
};
```

Замечание: при регистрации адаптера интерфейса указывается его полное имя, включающее имя класса адаптера эталонной модели.

Прослушиватель выходного интерфейса

Прослушивателем выходного интерфейса называется специальный процесс, который в цикле ожидает возникновения реакций на соответствующем интерфейсе целевой системы и, в случае если возникает неожиданная реакция, регистрирует ошибку. Определение прослушивателя выходного интерфейса осуществляется с помощью макроса CPPTESK_DEFINE_BASIC_OUTPUT_LISTENER(*имя*, *имя_интерфейса*, *условие*).

Пример:

```
#include <hw/media.hpp>

CPPTESK_ADAPTER(MyAdapter, MyModel) {
    CPPTESK_DEFINE_BASIC_OUTPUT_LISTENER(output_listener,
        output_iface, outputs.result);
    ...
};
```

Запуск прослушивателя выходного интерфейса осуществляется в конструкторе с помощью макроса CPPTESK_CALL_OUTPUT_LISTENER(*полное_имя*, *имя_интерфейса*).

Пример:

```
MyAdapter::MyAdapter{
    ...
    CPPTESK_CALL_OUTPUT_LISTENER(MyAdapter::output_listener, output_iface);
    ...
};
```

Арбитр реакций

Арбитр реакций (*арбитр выходного интерфейса*) предназначен для сопоставления реакций, полученных от целевой системы, с реакциями, вычисленными эталонной моделью (*эталонными реакциями*). После сопоставления пары реакций вызывается компаратор,

сравнивающий их и в случае несовпадения фиксирующий ошибку (см. раздел «*Компаратор выходного сообщения*»).

Основными типами арбитров реакций являются:

- `CPPTESK_FIFO_ARBITER` — при получении реакции от целевой системы предпочтение отдается той эталонной реакции, для которой эталонная модель раньше вызвала макрос `CPPTESK_SEND_REACTION()` (см. раздел «*Посылка реакции*»).
- `CPPTESK_PRIORITY_ARBITER` — предпочтение отдается эталонной реакции, для которой `CPPTESK_SEND_REACTION()` вызвал процесс с максимальным приоритетом (см. раздел «*Приоритет процесса*»).

Для декларации арбитра реакций в классе адаптера эталонной модели используется макрос `CPPTESK_DECLARE_ARBITER(`*`тип`*`, `*`имя`*`)`, а для привязки арбитра к выходному интерфейсу — `CPPTESK_SET_ARBITER(`*`интерфейс`*`, `*`арбитр`*`)`, который следует вызывать в конструкторе адаптера.

Пример:

```
#include <hw/media.hpp>

CPPTESK_ADAPTER(MyAdapter, MyModel) {
    CPPTESK_DECLARE_ARBITER(CPPTESK_FIFO_ARBITER, output_iface_arbiter);
    ...
};

MyAdapter::MyAdapter() {
    CPPTESK_SET_ARBITER(output_iface, output_iface_arbiter);
    ...
}
```

Описание тестового покрытия

Тестовое покрытие предназначено для оценки степени завершенности тестирования. Как правило, структура тестового покрытия описывается явным перечислением возможных при тестировании ситуаций (*тестовых ситуаций*). При описании сложных ситуаций используется композиция тестовых покрытий: тестовая ситуация представляется в форме совокупности нескольких более простых ситуаций.

Тестовое покрытие можно описывать в основном классе эталонной модели, а можно выносить его описание в отдельный класс (*класс тестового покрытия*). Во втором случае, класс с описанием тестового покрытия должен иметь ссылку на эталонную модель (см. раздел «*Класс тестового покрытия*»).

Класс тестового покрытия

Классом тестового покрытия называется специальный класс, содержащий *описание структуры тестового покрытия* и *функции вычисления тестовых ситуаций*. Поскольку тестовое покрытие определяется в терминах эталонной модели, класс тестового покрытия должен иметь ссылку на основной класс эталонной модели. Для трассировки тестовых ситуаций класс тестового покрытия содержит *трассировщик тестовых ситуаций* — объект класса `CoverageTracker` (пространство имен `cpptesk::tracer::coverage`) и *функции трассировки тестовых ситуаций*.

Пример:

```
#include <ts/coverage.hpp>
#include <tracer/tracer.hpp>

class MyModel;
```

```
// Декларация класса тестового покрытия
class MyCoverage {
public:
    MyCoverage(MyModel &model): model(model) {}

    // Трассировщик тестовых ситуаций
    CoverageTracker tracker;

    // Описание структуры тестового покрытия
    CPPTESK_DEFINE_ENUMERATED_COVERAGE(MY_COVERAGE, "My coverage", (
         (SITUATION_1, "Situation 1"),
         ...
         (SITUATION_N, "Situation N")
    ));

    // Функция вычисления тестовой ситуации: сигнатура функции
    // включает все необходимые для этого параметры
    MY_COVERAGE cov_MY_COVERAGE(...) const;

    // Функция трассировки тестовой ситуации: сигнатура функции
    // совпадает с сигнатурой функции вычисления тестовой ситуации
    void trace_MY_COVERAGE(...);
    ...
private:
    // ссылка на эталонную модель
    MyModel &model;
};
```

Структура тестового покрытия

Структура тестового покрытия описывается с помощью средств определения *перечислимых покрытий*, *произведения покрытий*, *произведения покрытий с исключением* и задания *синонимов покрытий*.

Перечислимое покрытие

Перечислимое покрытие, как следует из названия, определяется путем явного перечисления всех возможных тестовых ситуаций. Для этой цели предназначен специальный макрос `CPPTESK_DEFINE_ENUMERATED_COVERAGE(`*`покрытие`*`, `*`описание`*`, `*`ситуации`*`)`, где *`покрытие`* — это идентификатор, задающий тип покрытия, *`описание`* — строковый литерал, а *`ситуации`* — список тестовых ситуаций вида `((`*`идентификатор`*`, `*`описание`*`), ...)`.

Пример:

```
#include <ts/coverage.hpp>

CPPTESK_DEFINE_ENUMERATED_COVERAGE(FIFO_FULLNESS, "FIFO fullness", (
    (FIFO_EMPTY, "Empty"),
     ...
    (FIFO_FULL, "Full")
));
```

Произведение покрытий

Произведение позволяет на основе двух тестовых покрытий построить одно, множество тестовых ситуаций которого состоит из всевозможных пар ситуаций этих двух покрытий. Для этого предназначен макрос `CPPTESK_DEFINE_COMPOSED_COVERAGE(`*`тип`*`, `*`описание`*`, `*`покрытие_1`*`, `*`покрытие_2`*`)`. Описание тестовых ситуаций строится по шаблону `"%s,%s"`.

Пример:

```
#include <ts/coverage.hpp>

CPPTESK_DEFINE_ENUMERATED_COVERAGE(COVERAGE_A, "Coverage A", (
    (A1, "A one"),
    (A2, "A two")
));

CPPTESK_DEFINE_ENUMERATED_COVERAGE(COVERAGE_B, "Coverage B", (
    (B1, "B one"),
    (B2, "B two")
));

// Произведение тестовых покрытий A и B задает следующие ситуации:
// (COVERAGE_AxB::Id(A1, B1), "A one, B one")
// (COVERAGE_AxB::Id(A1, B2), "A one, B two")
// (COVERAGE_AxB::Id(A2, B1), "A two, B one")
// (COVERAGE_AxB::Id(A2, B2), "A two, B two")
CPPTESK_DEFINE_COMPOSED_COVERAGE(COVERAGE_AxB, "Coverage AxB",
    COVERAGE_A, COVERAGE_B);
```

Произведение покрытий с исключением

Для того чтобы построить произведение тестовых покрытий и *исключить* недостижимые тестовые ситуации, нужно вместо макроса CPPTESK_DEFINE_COMPOSED_COVERAGE использовать CPPTESK_DEFINE_COMPOSED_COVERAGE_EXCLUDING(*тип*, *описание*, *покрытие_1*, *покрытие_2*, *исключения*), где *исключения* — это список вида ({*покрытие_1*::*идентификатор*, *покрытие_2*::*идентификатор*}, ...). Вместо конкретного идентификатора тестовой ситуации можно использовать запись ANY(). Для произведений покрытий, которые сами являются произведениями, необходимо вместо пар использовать кортежи идентификаторов соответствующего размера.

Пример:

```
#include <ts/coverage.hpp>

CPPTESK_DEFINE_ENUMERATED_COVERAGE(COVERAGE_A, "Coverage A", (
    (A1, "A one"),
    (A2, "A two")
));

CPPTESK_DEFINE_ENUMERATED_COVERAGE(COVERAGE_B, "Coverage B", (
    (B1, "B one"),
    (B2, "B two")
));

// Определяемое ниже тестовое покрытие задает следующие ситуации:
// (COVERAGE_AxB::Id(A1, B2), "A one, B two")
// (COVERAGE_AxB::Id(A2, B1), "A two, B one")
// (COVERAGE_AxB::Id(A2, B2), "A two, B two")
CPPTESK_DEFINE_COMPOSED_COVERAGE_EXCLUDING(COVERAGE_AxB, "Coverage AxB",
    COVERAGE_A, COVERAGE_B, ({COVERAGE_A::A1, COVERAGE_B::B1}));
```

Синоним покрытия

Для создания *синонима тестового покрытия*, то есть покрытия с другим именем, но состоящего из тех же самых тестовых ситуаций, нужно использовать макрос CPPTESK_DEFINE_ALIAS_COVERAGE(*синоним*, *описание*, *покрытие*).

Пример:

```
#include <ts/coverage.hpp>
```

```
CPPTESK_DEFINE_ENUMERATED_COVERAGE(COVERAGE_A, "Coverage A", (
    (A1, "A one"),
    (A2, "A two")
));

// COVERAGE_B – синоним COVERAGE_A
CPPTESK_DEFINE_ALIAS_COVERAGE(COVERAGE_B, "Coverage B", COVERAGE_A);
```

Функция вычисления тестовой ситуации

Функцией вычисления тестовой ситуации называется функция, которая на основе анализа состояния эталонной модели и, возможно, входных параметров операции возвращает идентификатор текущей тестовой ситуации (см. раздел «*Структура тестового покрытия*»). Для перечислимых покрытий идентификатор тестовой ситуации имеет вид *покрытие*::*идентификатор*; а при использовании за пределами области видимости класса тестового покрытия — *класс*::*покрытие*::*идентификатор*. Функция вычисления тестовой ситуации для произведения покрытий получается путем вызова функций для составляющих его покрытий и «произведения» их результатов (оператор * перегружен должным образом).

Пример:

```
#include <ts/coverage.hpp>

class MyCoverage {
    // определение перечислимого покрытия COVERAGE_A
    CPPTESK_DEFINE_ENUMERATED_COVERAGE(COVERAGE_A, "Coverage A", (
        (A1, "A one"),
        (A2, "A two")
    ));
    // функция вычисления тестовой ситуации покрытия COVERAGE_A
    COVERAGE_A cov_COVERAGE_A(int a) const {
        switch(a) {
        case 1: return COVERAGE_A::A1;
        case 2: return COVERAGE_A::A2;
        }
        assert(false);
    }

    // определение COVERAGE_B – синонима COVERAGE_A
    CPPTESK_DEFINE_ALIAS_COVERAGE(COVERAGE_B, "Coverage B", COVERAGE_A);
    // функция вычисления тестовой ситуации покрытия COVERAGE_B
    COVERAGE_B cov_COVERAGE_B(int b) const {
        return cov_COVERAGE_A(b);
    }

    // определения COVERAGE_AxB – произведения COVERAGE_A и COVERAGE_B
    CPPTESK_DEFINE_COMPOSED_COVERAGE(COVERAGE_AxB, "Coverage AxB",
        COVERAGE_A, COVERAGE_B);s
    // функция вычисления тестовой ситуации покрытия COVERAGE_AxB
    COVERAGE_AxB cov_COVERAGE_AxB(int a, int b) const {
        return cov_COVERAGE_A(a) * cov_COVERAGE_B(b);
    }
    ...
};
```

Функция трассировки тестовой ситуации

Функция трассировки тестовой ситуации определяется для каждого покрытия верхнего уровня. Поскольку из функции трассировки вызывается функция вычисления тестовой ситуации, параметры этих функций обычно совпадают. Для реализации этой функции

используется трассировщик тестовых ситуаций — объект класса `CoverageTracker` (пространство имен `cpptesk::tracer::coverage`).

Пример:

```
#include <ts/coverage.hpp>
#include <tracer/tracer.hpp>

CoverageTracker tracer;

void trace_COVERAGE_A(int a) {
    tracer << cov_COVERAGE_A(a);
}
```

Создание тестового сценария

Тестовым сценарием называется высокоуровневая спецификация теста, которая путем ее интерпретации *обходчиком* (см. раздел «*Запуск тестового сценария*») используется тестовой системой для генерации тестовой последовательности. Тестовый сценарий оформляется в виде специального класса, который называется *сценарным классом*.

Сценарный класс

Сценарный класс декларируется с помощью макроса `CPPTESK_SCENARIO(`*имя*`)`.

Пример:

```
#include <ts/scenario.hpp>

CPPTESK_SCENARIO(MyScenario) {
    ...
private:
    // тестирование осуществляется через адаптер эталонной модели
    MyAdapter dut;
};
```

В сценарном классе декларируются *методы инициализации* и *завершения тестового сценария*, *сценарные методы* и *метод вычисления состояния*.

Метод инициализации тестового сценария

Метод инициализации тестового сценария объединяет в себе действия, которые нужно выполнить непосредственно перед началом тестирования. Он определяется путем перегрузки виртуального метода `bool init(int argc, char **argv)` базового класса. Метод возвращает `true` в случае успешной инициализации и `false` в случае ошибки.

Пример:

```
#include <ts/scenario.hpp>

CPPTESK_SCENARIO(MyScenario) {
public:
    virtual bool init(int argc, char **argv) {
        dut.initialize();
        std::cout << "Test has started..." << std::endl;
    }
    ...
};
```

Метод завершения тестового сценария

Метод завершения тестового сценария содержит действия, которые нужно выполнить сразу после окончания тестирования. Он определяется путем перегрузки виртуального метода `void finish()` базового класса.

Пример:

```
#include <ts/scenario.hpp>

CPPTESK_SCENARIO(MyScenario) {
public:
    virtual void finish() {
        dut.finalize();
        std::cout << "Test has finished..." << std::endl;
    }
    ...
};
```

Сценарный метод

Сценарные методы итерируют параметры входных сообщений и осуществляют запуск операций, используя адаптер эталонной модели. В одном сценарном классе может декларироваться несколько сценарных методов. Типом возвращаемого значения сценарного метода является `bool`: возвращаемое значение интерпретируется как признак возникновения ошибки. Единственным параметром сценарного метода является *контекст итерации* — объект, хранящий значения переменных, по которым осуществляется итерация (*итерационных переменных*). Определение сценарного метода начинается с вызова макроса `CPPTESK_ITERATION_BEGIN`, а заканчивается вызовом `CPPTESK_ITERATION_END`.

Пример:

```
#include <ts/scenario.hpp>

CPPTESK_SCENARIO(MyScenario) {
public:
    bool scenario(cpptesk::ts::IntCtx &ctx);
    ...
};

bool MyScenario::scenario(cpptesk::ts::IntCtx &ctx) {
    CPPTESK_ITERATION_BEGIN
    ...
    CPPTESK_ITERATION_END
}
```

Сценарные методы регистрируются в конструкторе сценарного класса с помощью макроса `CPPTESK_ADD_SCENARIO_METHOD(`*`полное_имя`*`)`.

Пример:

```
MyScenario::MyScenario() {
    CPPTESK_ADD_SCENARIO_METHOD(MyScenario::scenario);
    ...
}
```

Доступ к итерационным переменным

Итерационные переменные — это поля контекста итерации, передаваемого в качестве параметра сценарного метода. Для доступа к итерационным переменным предназначен макрос `CPPTESK_ITERATION_VARIABLE(`*`имя`*`)`, где *имя* — это название одного из полей контекста итерации `ctx`, передаваемого в качестве параметра.

Пример:

```
#include <ts/scenario.hpp>

bool MyScenario::scenario(cpptesk::ts::IntCtx &ctx) {
    // получение ссылки на итерационную переменную
    int &i = CPPTESK_ITERATION_VARIABLE(i);
    CPPTESK_ITERATION_BEGIN
```

```
        for(i = 0; i < 10; i++) {
            ...
        }
        CPPTESK_ITERATION_END
    }
```

Блок тестового воздействия

Тестовое воздействие (подготовка входного сообщения и запуск операции) осуществляется в специальном блоке `CPPTESK_ITERATION_ACTION{ ... }` сценарного метода.

Пример:

```
#include <ts/scenario.hpp>

...
CPPTESK_ITERATION_BEGIN
for(i = 0; i < 10; i++) {
     ...
     // блок тестового воздействия
     CPPTESK_ITERATION_ACTION {
         // рандомизация входного сообщения
         CPPTESK_RANDOMIZE_MESSAGE(input_msg);
         input_msg.set_addr(i);
         // запуск операции
         CPPTESK_CALL_STIMULUS_OF(dut, MyModel::operation,
             dut.input_iface, input_msg);
         ...
     }
}
CPPTESK_ITERATION_END
```

Выход из сценарного метода

Каждая итерация сценарного метода должна заканчиваться вызовом макроса `CPPTESK_ITERATION_YIELD(`*`вердикт`*`)`, который осуществляет выход из сценарного метода. При следующем обращении к сценарному методу его выполнение продолжится со следующей итерации.

Пример:

```
#include <ts/scenario.hpp>

...
CPPTESK_ITERATION_BEGIN
for(i = 0; i < 10; i++) {
     ...
     // блок тестового воздействия
     CPPTESK_ITERATION_ACTION {
         ...
         // выход из сценарного метода с указанием вердикта
         CPPTESK_ITERATION_YIELD(dut.verdict());
     }
}
CPPTESK_ITERATION_END
```

Временные задержки

Для создания в тесте *временных задержек* необходимо хотя бы в одном сценарном методе осуществлять вызов метода `cycle()` адаптера эталонной модели. Распространенным подходом к разработке тестовых сценариев для HDL-моделей аппаратуры, допускающей параллельную подачу стимулов, является создание специального сценарного метода

`nop()`[7], в котором создается временн*ая* задержка путем вызова метода `cycle()`. Во всех остальных сценарных методах `cycle()` не вызывается.

Пример:

```
#include <ts/scenario.hpp>

bool MyScenario::nop(cpptesk::ts::IntCtx& ctx) {
    CPPTESK_ITERATION_BEGIN
    CPPTESK_ITERATION_ACTION {
        dut.cycle();
        CPPTESK_ITERATION_YIELD(dut.verdict());
    }
    CPPTESK_ITERATION_END
}
```

Замечание: сценарный метод `nop()` должен регистрироваться перед всеми остальными сценарными методами.

Метод вычисления состояния

Метод вычисления состояния предназначен для обходчиков, использующих для построения тестовой последовательности обход графа состояний целевой системы. Возвращаемое методом значение интерпретируется как состояние системы. Тип возвращаемого значения и имя метода могут быть произвольными, параметры отсутствуют.

Пример:

```
#include <ts/scenario.hpp>

CPPTESK_SCENARIO(MyScenario) {
public:
    ...
    int get_model_state() {
        return dut.buffer.size();
    }
};
```

Установка метода вычисления состояния осуществляется в конструкторе сценарного класса с помощью метода `void setup(...)`.

Пример:

```
#include <ts/scenario.hpp>

MyScenario::MyScenario() {
    setup("My scenario",
    UseVirtual::init,
    UseVirtual::finish,
    &MyScenario::get_model_state);
    ...
}
```

Запуск тестового сценария

Запуск тестового сценария на одном компьютере осуществляется с помощью вызова функции `localmain(`*`обходчик`*`, `*`сценарий`*`.getTestScenario(), `*`argc`*`, `*`argv`*`)`(пространство имен `cpptesk::ts`).

Доступные обходчики (пространство имен `cpptesk::ts::engine`):

[7] Название этого метода может быть произвольным.

- `fsm` — генератор тестовой последовательности на основе обхода графа состояний;
- `rnd` — генератор случайной тестовой последовательности.

Пример:

```
#include <netfsm/engines.hpp>

using namespace cpptesk::ts;
using namespace cpptesk::ts::engine;
...
MyScenario scenario;
localmain(fsm, scenario.getTestScenario(), argc, argv);
```

Вспомогательные возможности

К вспомогательным возможностям инструмента C++TESK относятся *утверждения* (*assertions*) и *отладочная печать*. Эти возможности можно использовать как в эталонных моделях, так и в любых других компонентах тестовой системы (адаптерах, тестовых сценариях и др.). В первую очередь они предназначены для *отладки* тестовой системы.

Утверждения

Утверждениями (*assertions*) называются предикаты (логические выражения), используемые для описания свойств программ и, как правило, проверяемые во время выполнения. При нарушении утверждения (ложном значении предиката) фиксируется ошибка, и выполнение программы прекращается. Задание утверждений осуществляется с помощью макроса `CPPTESK_ASSERTION(`*`предикат`*`, `*`описание`*`)`, где *`предикат`* — это проверяемое свойство, а *`описание`* — строка, описывающая ошибку, связанную с нарушением этого свойства.

Пример:

```
#include <hw/assertion.hpp>

...
CPPTESK_ASSERTION(pointer, "pointer is null");
```

Отладочная печать

Отладочная печать, как видно из названия, предназначена для отладки тестовой системы. В отличие от обычного вывода с помощью, например, потоков STL отладочную печать удобнее настраивать (включать/выключать, изменять цвет вывода и т.п.).

Макросы отладочной печати

Отладочная печать осуществляется с помощью двух основных макросов: `CPPTESK_DEBUG_PRINT(`*`уровень`*`, `*`сообщение`*`)`, где *`уровень`* — это уровень отладочной печати (см. раздел «*Уровни отладочной печати*»), а *`сообщение`* — выводимое на печать *отладочное сообщение*, и `CPPTESK_DEBUG_PRINTF(`*`уровень`*`, `*`формат`*`, `*`параметры`*`)`, где (*`формат`*`, `*`параметры`*`)` — это форматная строка и значения используемых в ней параметров в том виде, в каком они используются в библиотечной функции `printf()` языка C.

Пример:

```
#include <hw/debug.hpp>

using namespace cpptesk::hw;
...
CPPTESK_DEBUG_PRINT(DEBUG_USER, "The input message is "
    << CPPTESK_GET_MESSAGE());
```

```
...
CPPTESK_DEBUG_PRINTF(DEBUG_USER, "counter=%d", counter);
```

Замечание: в качестве отладочного сообщения в макросе `CPPTESK_DEBUG_PRINT()` можно использовать любое «*потоковое выражение*», допустимое при выводе с помощью потоков STL (стандартной библиотеки языка C++).

Для того чтобы к сообщению добавлялась информация о положении макроса отладочной печати в исходном коде (имя файла и номер строки), можно использовать макросы `CPPTESK_DEBUG_PRINT_FILE_LINE()` и `CPPTESK_DEBUG_PRINTF_FILE_LINE()`, параметры которых аналогичны параметрам макросов, описанных выше.

Вывод стека вызовов процессов

Для вывода *стека вызова процессов* эталонной модели предназначен макрос `CPPTESK_CALL_STACK()`, который можно использовать внутри или вместо отладочного сообщения в макросе `CPPTESK_DEBUG_PRINT()`.

Пример:

```
#include <hw/model.hpp>

CPPTESK_DEFINE_PROCESS(MyModel::some_process) {
    CPPTESK_START_PROCESS();

    CPPTESK_DEBUG_PRINT(DEBUG_USER, "Call stack is "
        << CPPTESK_CALL_STACK());
    ...
    CPPTESK_STOP_PROCESS();
}
```

Замечание: макрос `CPPTESK_CALL_STACK()` можно использовать только внутри эталонной модели.

Выделение отладочной печати цветом

Для ускорения визуального поиска отладочных сообщений определенного вида в общем потоке отладочной печати можно использовать макросы *цветной отладочной печати*: `CPPTESK_COLORED_DEBUG_PRINT(`*`уровень`*`, `*`цвет`*`, `*`цвет_фона`*`, `*`сообщение`*`)` и `CPPTESK_COLORED_DEBUG_PRINTF(`*`уровень`*`, `*`цвет`*`, `*`цвет_фона`*`, `*`формат`*`, `*`параметры`*`)`, а также макросы `CPPTESK_COLORED_DEBUG_PRINT_FILE_LINE()` и `CPPTESK_COLORED_DEBUG_PRINTF_FILE_LINE()`.

Определены следующие цветовые константы (пространство имен `cpptesk::hw`):

- `BLACK` — черный;
- `RED` — красный;
- `GREEN` — зеленый;
- `YELLOW` — желтый;
- `BLUE` — синий;
- `MAGENTA` — пурпурный;
- `CYAN` — голубой;
- `WHITE` — белый.

Пример:

```
#include <hw/debug.hpp>

using namespace cpptesk::hw;
```

```
...
CPPTESK_COLORED_DEBUG_PRINT_FILE_LINE(DEBUG_USER, RED, BLACK,
    "The input message is " << CPPTESK_GET_MESSAGE());
...
CPPTESK_COLORED_DEBUG_PRINTF(DEBUG_USER, WHITE, BLACK,
    "counter=%d", counter);
```

Уровни отладочной печати

При отладочной печати указывается *уровень* выводимого сообщения. Уровень характеризует степень важности сообщения. Обычно для отладочных сообщений разных уровней используется различное выделение цветом. Определены следующие значения уровней отладочной печати (пространство имен `cpptesk::hw`):

- `DEBUG_MORE` — подробные отладочные сообщения, выдаваемые инструментом;
- `DEBUG_INFO` — базовые отладочные сообщения, выдаваемые инструментом;
- `DEBUG_USER` — пользовательские отладочные сообщения;
- `DEBUG_WARN` — предупреждения (как правило, выдаются инструментом);
- `DEBUG_FAIL` — сообщения об ошибках (как правило, выдаются инструментом).

Самым «*важным*» уровнем является `DEBUG_FAIL`, затем — `DEBUG_WARN` и т.д. Для пользовательских сообщений определен один уровень отладочной печати — `DEBUG_USER`.

Настройка отладочной печати

Для настройки объема вывода можно указать *уровень отладочной печати* — печатаются только сообщения с уровнем не меньшим заданного. Это осуществляется с помощью макроса `CPPTESK_SET_DEBUG_LEVEL(`*`уровень_печати`*`, `*`цветной_вывод`*`)`. Макрос имеет еще один булев параметр *`цветной_вывод`*, включающий или отключающий цветовое выделение. По умолчанию установлен уровень печати `DEBUG_INFO`. Для отключения отладочной печати следует использовать специальный уровень `DEBUG_NONE`.

Для каждого уровня отладочной печати можно задать *цвет вывода* сообщений этого уровня. Для этого предназначен макрос `CPPTESK_SET_DEBUG_STYLE(`*`уровень`*`, `*`цвет_тэга`*`, `*`цвет_фона_тэга`*`, `*`цвет`*`, `*`цвет_фона`*`)`.

Пример:

```
#include <hw/model.hpp>

using namespace cpptesk::hw;
...
// конструктор эталонной модели
MyModel::MyModel() {
    // печатать только сообщения об ошибках,
    // включить цветовое выделение
    CPPTESK_SET_DEBUG_LEVEL(DEBUG_FAIL, true);
    // [FAIL] Error message style
    CPPTESK_SET_DEBUG_STYLE(DEBUG_FAIL, BLACK, RED, RED, BLACK);
}
```