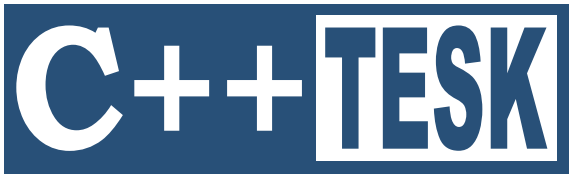

C++TESK Hardware Edition: **Быстрое знакомство**

Версия 1.0, 24/05/2011

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

Под *верификацией аппаратуры* обычно понимается процесс проверки ее *поведения* на соответствие ее *спецификации.* Такой процесс может быть сделан *формально,* с помощью, например, проверки моделей, автоматического доказательства теорем и др. Также верификация может быть проведена при *имитационном моделировании* модуля аппаратуры с помощью *симулятора.*

Учитывая сложность проверяемых моделей аппаратуры, обычно перед собственно верификацией необходимо решить задачу автоматизации. Чем больше процессов будут делаться автоматически, чем меньше потребуется ручного труда, тем эффективнее будет происходить проверка. Не затрагивая формальные методы верификации, в данном курсе мы остановимся на использовании только имитационного моделирования. Причем, далее речь в основном будет идти об одном из существующих инструментов верификации аппаратуры, созданном в Институте системного программирования РАН. Возможности инструмента позволяют говорить о нем, как о мощном и вполне современном решении. Итак, речь будет идти об использовании *C++TESK Testing ToolKit*.

C++TESK Testing ToolKit реализует подход к верификации на основе имитационного моделирования. Основным элементом инструмента является его *ядро* — библиотека, реализованная на языках программирования C и C++. Ядро основано на инструменте верификации *CTESK*, дополненного библиотеками *C++TESK, C++TESK-HE, NetFSM, Aspectrace*. Все компоненты ядра скомпонованы в один пакет и доступны по адресу в Интернете http://forge.ispras.ru/projects/cpptesk-toolkit/files. Текущая версия: 1.0.2 от 04.05.2011. Инструмент предназначен для создания *тестовых систем* с использованием C++ для различных моделей *синхронной аппаратуры* на различных *уровнях абстракции.* Тестовые системы создаются с использованием любых средств, предоставляемых C++, на основе подхода, макросов и классов, определенных C++TESK Testing ToolKit.

При создании тестовых систем для имитационного моделирования обычно решаются три основные задачи: *построения тестовой последовательности, оценки правильности поведения* и *оценки полноты тестирования.* C++TESK Testing ToolKit позволяет строить последовательности двух типов: *составляемые* на каждом такте моделирования *случайно* из заранее описанных стимулов и *получаемые* динамически *на* основе техник *обхода* неявно заданных *конечных автоматов.* Оценка правильности поведения определяется на каждом такте на основе *исполнимой эталонной модели*, созданной разработчиком тестовой системы, на том или ином уровне абстракции. Также может быть использована сторонняя модель, например, *симулятор системы*. *Полнота (законченность) тестирования* определяется либо по *количеству тактов тестирования* для случайно выбираемых воздействий, либо на основе *информации о завершенности обхода конечного автомата*.

Общая схема тестовой системы представлена на рис. 1.

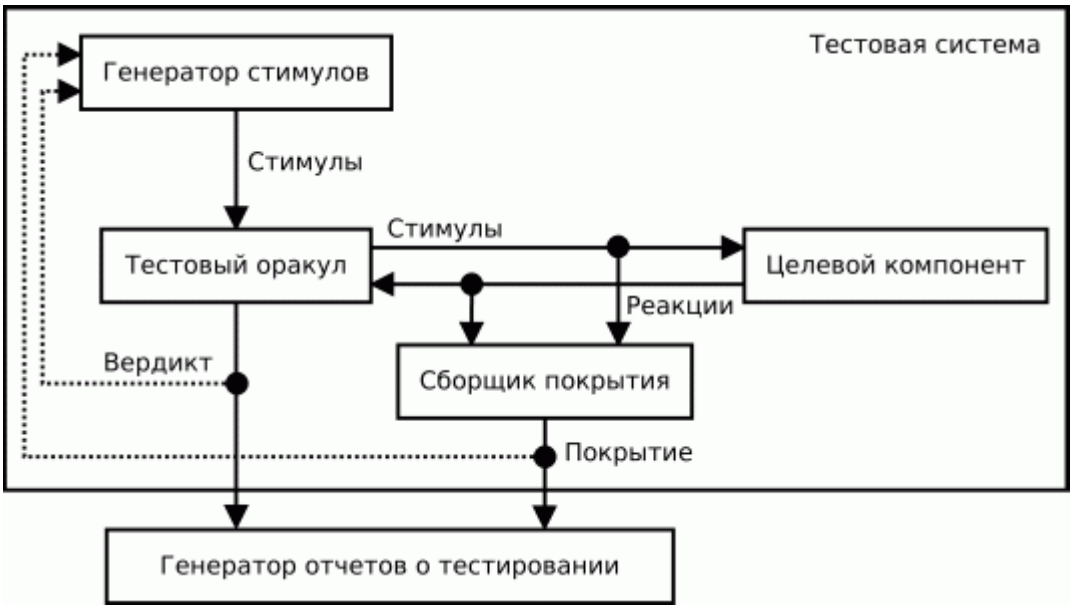

Рис. 1. Общая схема тестовой системы

Рассмотрим кратко элементы, представленные на данной общей схеме:

1. *Генератор стимулов* - компонент, создающий поток входных воздействий (стимулов); настраивается с помощью сценариев тестирования;

2. *Тестовый оракул* - компонент, принимающий потоки данных от генератора воздействий и от целевого компонента, передающий поток стимулов от генератора целевому компоненту, оценивающий корректность поведения целевого компонента;

3. *Целевой компонент* - модель аппаратуры, разработанная на одном из языков описания аппаратуры (в данном случае, Verilog), принимающая поток стимулов и отвечающая на него реакциями, которые необходимо проверить;

4. *Сборщик покрытия* - компонент, собирающий информацию о функциональном покрытии эталонной модели, которая в общем случае влияет на работу генератора стимулов с помощью информации об уже достигнутом покрытии,

5. *Генератор отчетов о тестировании* - компонент, создающий отчеты-трассы тестов с информацией о пройденных переходах, достигнутом покрытии, найденных ошибках и др.

Далее по шагам будут рассмотрены следующие вопросы:

- анализ документации применительно к разработке тестовых систем с помощью C++TESK Testing ToolKit;
- создание тестовых оракулов, включая эталонные модели и их медиаторы;
- определение модели функционального покрытия;
- настройка генераторов стимулов;
- собственно тестирование.

1. Анализ документации: выделение функциональности и интерфейсов

Под *верификацией* компонента мы пониманием процесс соотнесения его наблюдаемого поведения требованиям к нему. Иначе говоря, верификация – это установление соответствия между поведением тестируемой модели и ее спецификациями. Следовательно, помимо собственно модуля, в нашем распоряжении должна быть *спецификация* на модуль — документ, разрабатываемый в начале проектирования и немного модифицируемый в его процессе, содержащий сведения о функционировании модуля.

В реальной жизни возникают ситуации, когда спецификации очень скудные, либо их вообще нет, а разработчики ограничились списком сигналов *входных* и *выходных интерфейсов* тестируемого модуля. В таких случаях проводить верификацию затруднительно, так как сказать, что какое-то поведение тестируемого модуля ошибочно, можно только имея на руках информацию о том, каким оно должно быть. Тогда приходится опрашивать разработчиков модуля, самостоятельно составлять список требований, которым должно удовлетворять тестируемое устройство. Обладая списком требований (спецификацией), можно начинать разработку тестовой системы.

Практика подсказывает, что представлять спецификации, особенно потактовые, удобно следующим образом:

1. в виде *блок-схем* (см. рис.1).

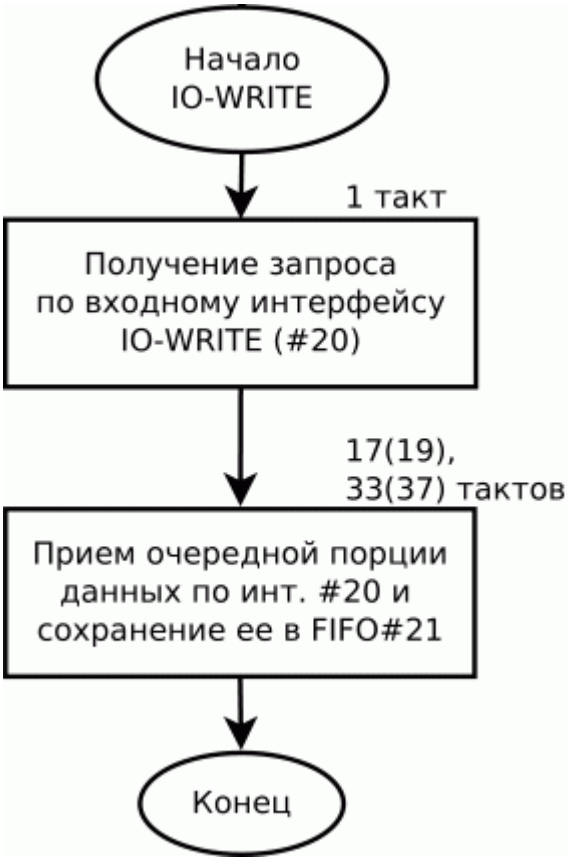

Рис. 1 Пример блок-схемы для отдельной операции

Блок-схема позволяет описать поведение эталонной модели с любым приближением к потактовой точности. На рис. 1 изображено абстрактное представление операции IO-WRITE (запись по каналу IO), которое начинается с однотактового запроса по интерфейсу 20. На следующем такте после запроса осуществляется прием данных, который длится 17, 19, 33 или 37 тактов в зависимости от длины данных и наличия маски среди передаваемых данных (данная блок-схема является недостаточной для построения эталонной модели, к ней прилагалась *таблица потактовой раскладки*, о которой будет рассказано далее). После получения всех данных и их сохранения, выполнение операции заканчивается.

В блок-схемах обычно присутствуют следующие обозначения:

- овальными символами обозначаются точки начала и конца операций; в точке начала пишется мнемоническое обозначение операции, которая будет исполняться в пределах этого пути контроля управления;
- прямоугольными символами обозначаются блоки, выполнение которых в процессе работы операции, занимает ровно один такт на данном уровне абстракции. Внутри блока пишется описание процесса функционирования эталонной модели. Рядом с блоком допускается описание количества реальных тактов выполнения стадии тестируемым устройством, другая вспомогательная информация, позволяющая сопоставить схему разработанной эталонной модели;
- ромбическими символами обозначаются пути ветвления потока управления. Внутри символа должно быть описано условие ветвления. Входящий путь должен быть один, исходящих путей должно быть ровно два;
- ветвление потока управления обозначается жирной точкой, из которой исходят два или более потока управления;
- соединение нескольких потоков управления осуществляется также в жирной точке, при этом потоки управления предполагаются синхронизированными, т.е. управление на выходной поток передастся только в том случае, когда выполнятся все входные потоки.

2. в виде *таблиц потактовой раскладки* (см. табл. 1).

Табл. 1. Пример таблицы потактовой раскладки для отдельной операции

Стимул	Ветвление 0	Микрооперация 0	Микрооперация 1	Ветвление 1	Ветвление 2	Микрооперация 2	Микрооперация 3
PRE: 1) единовременно может работать только одна операция данного типа 2) **val_wr_data_buff_ nreg_to_IO(o)=1**	**BRANCH:** если val_mask=1 (в стимуле), на следующем такте после подачи стимула начинает выполняться микрооперация 0. Иначе начинает выполняться микрооперация 1.	**PRE:** нет	**PRE:** нет	**BRANCH:** если wr64=0 (в стимуле), выполнение операции завершается. Иначе начинает выполняться ветвление 2.	**BRANCH:** если val_mask=1 (в стимуле), на следующем такте после выполнения микрооп. 1 начинает выполняться микрооп. 2. Иначе начинает выполняться микрооп. 3.	**PRE:** нет	**PRE:** нет
Устанавливаются параметры стимула: **val_wr_data_ from_IO (i)=1** (строб) **wr_data_ from_IO[15:0](i)={0 -7}** Стимул подается один такт.		По **wr_data_ from_IO[15:0] (i)** передается маска. Маска записывается в **буфер (21)** (глубина 4 32-х или 64-байтных слова). Данная микрооп.	По **wr_data_ from_IO[15:0] (i)** передается 2 байта данных. Данные записываются в **буфер (21).** Данная микрооперация повторяется 16			По **wr_data_ from_IO[15:0] (i)** передается маска для второй части 64-байтной передачи. Маска записывается в **буфер (21).** Данная микрооп. повторяется 2	По **wr_data_ from_IO[15:0] (i)** передается 2 байта данных второй части 64-байтной посылки. Данные записываются в **буфер (21).** Данная микрооп. повторяется 16 раз

		повторяется 2 раза подряд.	раз подряд.			раза подряд.	подряд.
		POST: на следующем такте после **val** сигнал **wr_data_buff_ nreg_to_IO (o)** (номер ячейки буфера (21), в которую происходит запись) меняется на номер следующей свободной ячейки; или на 8 в случае ее отсутствия	**POST:** на следующем такте после **val** сигнал **wr_data_buff_ nreg_to_IO (o)** (номер ячейки буфера (21), в которую происходит запись) меняется на номер следующей свободной ячейки; или на 8 в случае ее отсутствия			**POST:** нет	**POST:** нет
INPUT: **val_wr_data_from_IO**(i) - строб **wr_data_from_IO[15:0](i)** состоит из - [15:3] - резерв; - [2] **A5** - признак записи во вторую половину 64-байтовой ячейки; - [1] **wr64** - признак 64-байтовой посылки; - [0] **val_mask** - будет маска или нет.		**INPUT**: **wr_data_ from_IO[15:0]** - маска	**INPUT: wr_data_ from_IO[15:0]** - данные			**INPUT**: **wr_data_ from_IO[15:0]** - маска	**INPUT: wr_data_ from_IO[15:0]** - данные

Такая таблица отображает поведение тестируемого модуля с потактовой точностью, информация в ней соответствует блок-схеме, служит для дальнейшего проектирования потактовых эталонных моделей. Таблица создается для каждой отдельной операции, выполняемой тестируемым модулем. Первой колонкой является информация о стимуле, т.е. посылке в тестируемый модуль, начинающей операцию. Приводится информация о предусловии запуска операции (т.е. тех условиях, которые должны быть выполнены для начала работы операции), об устанавливаемых входных сигналах. Среди последующих колонок могут быть колонки четырех типов, соответствующих блок-схемам: микрооперация (один такт на данном уровне абстракции), ветвление, распараллеливание, соединение. Предполагается, что исполнение операции на потактовой раскладке происходит слева направо: первым запускается стимул, затем – то, что находится во второй колонке. Для каждой микрооперации пишутся сведения о предусловии ее запуска, выполняемых действиях эталонной модели, постусловии, которое должно быть проверено после выполнения микрооперации, используемые входные и выходные сигналы. Для каждого ветвления указывается условие ветвления и колонки, на которые будет происходить переход. Для каждого распараллеливания и соединения потоков управления обозначаются входные и выходные колонки.

Если для верификации не будет изготавливаться потактовая эталонная модель (например, она будет взята из системного симулятора) или она будет спроектирована абстрактной, обычно можно обойтись спецификациями в простом текстовом виде. Поэтому, если в таком виде они уже были написаны, создавать какие-либо дополнительные представления спецификаций не требуется.

Теперь перейдем непосредственно к анализу требований применительно к C++TESK Testing ToolKit.

Для создания C++TESK Testing ToolKit-компонентов «*эталонная модель*» и «*медиатор эталонной модели*» необходимо создавать *входные* и *выходные интерфейсы*. В каждом верифицируемом модуле присутствуют как входные, так и выходные сигналы, которые можно сгруппировать в *интерфейсы* по признаку их принадлежности к определенной активности: записи или чтению. Функционально законченная последовательность обращения

к интерфейсам, в результате которой модуль перейдет в некоторое стационарное состояние, в котором он сможет находиться бесконечно долгое время, будем называть *операцией*. Отметим, что сигналы типа CLK и RST обычно не относятся ни к одному интерфейсу. Каждая операция может использовать несколько входных и выходных интерфейсов. При анализе документации следует помнить, что входные интерфейсы содержат **только входные сигналы**, а выходные интерфейсы – **только выходные сигналы**.

Рассмотрим пример выделения информации об интерфейсах. Возьмем фрагмент спецификации на модуль Databox (DB) - коммутатор данных микропроцессора - ту ее часть, где осуществляется связь DB с внешним модулем Memory Access Unit (MAU) - компонентом доступа к памяти.

Данные из MAU.

*Арбитр **(13)**, работающий с круговым приоритетом, из 5 источников коротких запросов (запросы в общем случае приходят параллельно) выбирает один запрос, который будет направлен в MAU. Темп передачи 1 запрос за 2 такта в случае 32-х байтового или 1 запрос за 4 такта в случае 64-х байтового запроса. При этом сам короткий запрос запоминается в DB и в MAU не передается. Запрос в MAU имеет следующий интерфейс (см. сл. табл.).*

Табл. 2. Интерфейс запросов в MAU

Сигнал	***Тип***	***Назначение***
val_data_reqack_to_mau_03	***O***	*Значимость запроса. Запрос за младшей половиной 64 байтной кэш строки.*
val_data_reqack_to_mau_47	***O***	*Значимость запроса. Запрос за старшей половиной 64 байтной кэш строки.*
cop_data_reqack_to_mau[2:0]	***O***	*Код операции:* *«011» - Сбросить регистр LDB* *«100» - Выдать когерентный ответ.* *«101» - Сбросить регистр STB* *«110» - Выдать данные из STB* *«111» - Выдать данные и сбросить регистр STB*
source_reg_data_reqack_to_mau[4:0]	***O***	*Номер регистра LDB/STB*

Исходя из этого фрагмента документации, для построения абстрактной эталонной модели необходимо будет создать входные интерфейсы для всех пяти источников коротких запросов, прочитать об этих интерфейсах надо, видимо, в тексте продолжении фрагмента. Далее по тексту: «запрос передается в MAU», MAU – это устройство внешнее по отношению к тестируемому модулю, поэтому тестовой системе необходимо будет контролировать выходные значения, передаваемые туда.

Для контроля этих сигналов необходимо, во-первых, их считать, и, во-вторых, необходимо знать, какие именно сигналы будут правильными. Для решения второй задачи в эталонной

модели будет создан компонент "выходной интерфейс" соответствующий выходному интерфейсу тестируемой реализации. В медиаторе (компоненте-наследнике эталонной модели, об этом немного дальше) этот компонент перегружается, но, по-прежнему, называется выходным интерфейсом. Выходной интерфейс медиатора будет решать задачу считывания сигналов, исходящих от тестируемого компонента и передачи их на проверку. Этот выходной интерфейс поименуем, например, iface14. В него будут входить следующие сигналы: val_data_reqack_to_mau_03, val_data_reqack_to_mau_47, cop_data_reqack_to_mau, source_reg_data_reqack_to_mau.

Отметим, что выделение сигналов для интерфейса на данном этапе развития инструмента носит скорее логический характер. Только эти сигналы будут допущены для использования в функциях медиатора эталонной модели, связанных с этим интерфейсом, но проверок использования сигналов на предмет корректности в инструменте не предусмотрено.

После выделения интерфейсов, остальная документация упорядочивается, как было написано выше, в виде блок-схем или таблиц потактовой раскладки, если необходимо будет разрабатывать эталонную модель. Если эталонную модель создавать не требуется, т.к., например, она уже есть, можно переходить к следующей части проектирования.

2. Разработка эталонной модели

Мы переходим к поэтапному созданию компонента *тестовый оракул* - компонента проверки. В него поступают стимулы от генератора стимулов, затем он передает стимулы в целевой компонент и получает из него реакции для проверки в ответ на стимулы (см. рис. 2). Эта проверка требует наличия образца для сравнения: роль этого образца играет *эталонная модель*.

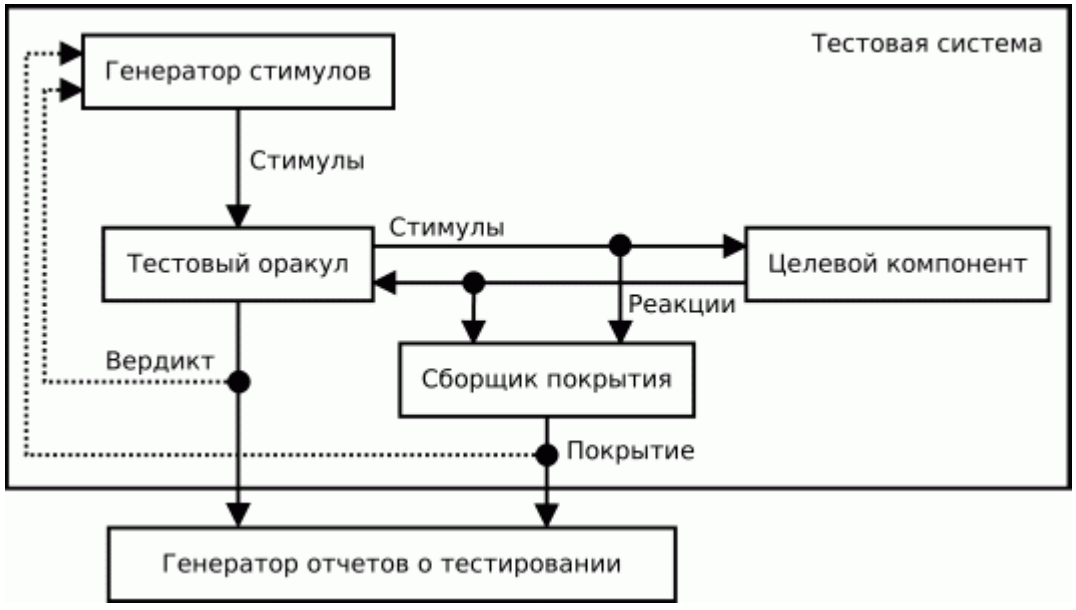

Рис. 2. Обобщенная структура тестовой системы

Можно сказать, что тестовый оракул – это своеобразная «обертка» над эталонной моделью, мы к нему еще вернемся позднее, остановившись в данном разделе на создании эталонной модели. Структура эталонной модели, принятой в C++TESK Testing ToolKit, представлена на рис. 3.

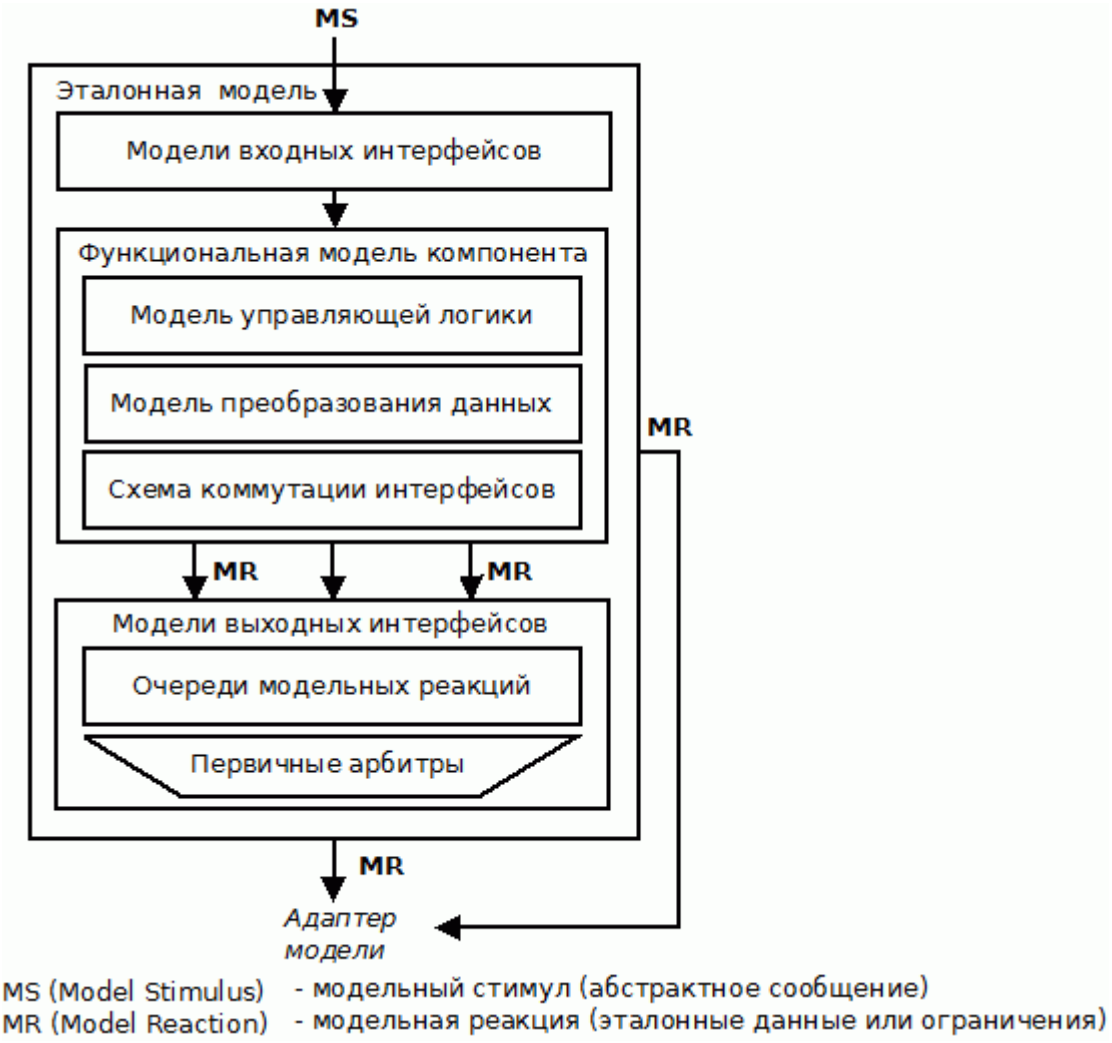

Рис. 3. Структура эталонной модели C++TESK Testing ToolKit

Основными составными частями модели являются:

- *модели входных интерфейсов*;
- *функциональная модель тестируемого компонента;*
- *модели выходных интерфейсов*.

Все данные внутри модели, а также в нее и из нее передаются с помощью *сообщений* – объектов классов-наследников класса *Message*. Это сделано для унификации интерфейсов между отдельными компонентами тестовой системы.

Модели входных и выходных интерфейсов являются объектами класса *Interface*.

Функциональная модель компонента является частью класса эталонной модели, хотя некоторая ее часть, допускающая в дальнейшем повторное использование, может быть реализована в отдельных классах. Функциональная модель состоит из *модели управляющей логики*, *модели преобразования данных*, *схемы коммутации интерфейсов*. Собственно, модели логики и преобразования данных и могут быть реализованы отдельно, а вот схема коммутации интерфейсов зависит от выделенных интерфейсов в классе эталонной модели, поэтому ее выделение в отдельный класс нецелесообразно.

Ниже приведены основные сведения о создании моделей сообщений (обычно сообщений требуется не менее двух: для входных и выходных интерфейсов), затем – о создании собственно эталонной модели.

2.1 Модель сообщений

Описание необходимых макросов и классов находится в файле **<hw/message.hpp>**.

Модель сообщения является классом, который создается с помощью макроса *MESSAGE(Имя класса) {}*. В скобках указывается имя класса сообщения. Внутри блока макроса объявляются стандартные конструктор и виртуальный деструктор, пока пустые. Затем включаем возможность автоматизированного копирования сообщения с помощью объявления макроса *SUPPORT_CLONE(Имя класса)*. Каждое сообщение включает в себя одно или несколько *полей* для данных, которые передает это сообщение. Эти поля не обязательно в точности соответствуют имеющимся сигнальным линиям тестируемого модуля, они должны быть удобны разработчику тестовой системы, чтобы наглядно представлять принимаемые от тестируемого устройства и передаваемые в тестируемое устройство данные. Можно сказать, что они соответствуют имеющимся сигнальным линиям на некотором абстрактном, логическом уровне, но не стоит доводить ситуацию до абсурда. Использовать только одно "глобальное" поле в сообщении, которое будет содержать данные, передаваемые тестируемым устройством по всем его входным и выходным сигнальным линиям, очень не удобно.

Для включения в сообщение полей, используется следующий макрос: *DECLARE_FIELD(имя, разрядность поля)*. Разрядность поля не должна превышать 64 бит. Для каждого поля автоматически создаются функции установки и считывания его значения, обращение к которым выглядит следующим образом: *имя_объекта_сообщения.get_имя_поля* и *имя_объекта_сообщения.set_имя_поля*. В конструкторе допускается ручная инициализация полей: они доступны по своим именам. Однако, у разработчика нет необходимости вручную присваивать включенным в сообщение полям случайные значения: он может просто вызвать в конструкторе макрос *RANDOMIZE_MESSAGE(указатель_на_класс_сообщения)*, где в качестве параметра передается **this*. Отметим, что все явно присвоенные значения полей будут потеряны после вызова этого макроса, поэтому его вызов надо делать перед ручной установкой. В реализации конструктора также необходимо использовать макрос регистрации поля *ADD_FIELD(имя_класса_сообщения::имя_поля_сообщения)*.

Таким образом, заголовочный файл описания модельных сообщений (*fifo_msg.h*) будет выглядеть, например, так:

```
#pragma once
#include <hw/message.hpp>
namespace cpptesk {
namespace fifo {

MESSAGE(InputData) {
public:
  InputData();
  virtual ~InputData();

  SUPPORT_CLONE(InputData);
  DECLARE_FIELD(data, 8);
};

MESSAGE(OutputData) {
public:
```

```
  OutputData();
  OutputData(uint8_t datum);
  virtual ~OutputData();

  SUPPORT_CLONE(OutputData);
  DECLARE_FIELD(data, 8);
};
}}
```

Обратите внимание, что директива #pragma once является распространенным, хотя и не совсем стандартным путем указания компилятору на необходимость наличия в откомпилированной библиотеке только одного включения данного заголовочного файла (другой путь - использование директив #ifndef, #define, и #endif).

Файл с реализацией методов классов сообщений (*fifo_msg.cpp*) может быть следующим:

```
#include <fifo_msg.h>
namespace cpptesk {
namespace fifo {

InputData::InputData(void) {
  ADD_FIELD(InputData::data);
  RANDOMIZE_MESSAGE(*this);
}

InputData::~InputData(void) {}

OutputData::OutputData(void) {
  ADD_FIELD(OutputData::data);
}

OutputData::OutputData(uint8_t output_data) {
  ADD_FIELD(OutputData::data);
  data = output_data;
}

OutputData::~OutputData(void) {}
}}
```

2.2 Эталонная модель

Описание необходимых макросов и классов находится в файле **<hw/model.hpp>**.

Класс модели создается с помощью макроса *MODEL(Имя_класса_модели) {}*. Необходимыми элементами являются конструктор и виртуальный деструктор. В модели определяются используемые интерфейсы с помощью макросов *DECLARE_INPUT(имя_интерфейса)* для входных в тестируемое устройство интерфейсов и *DECLARE_OUTPUT(имя_интерфейса)* для выходных из тестируемого устройства интерфейсов. К каждому входному интерфейсу необходимо определить метод, возвращающий состояние интерфейса: можно ли им пользоваться в данный момент времени или нет (реализация этого метода основана на стандартной возможности: функции *имя_интерфейса.isReady()*, возвращающей текущее состояние занятости интерфейса какой-либо операцией). Интерфейс может быть недоступен, если какая-либо операция уже заняла его. Условимся называть такие методы следующим образом: *virtual bool isИмяИнтерфейсаReady() const*.

Помимо интерфейсов, в модели определяются операции — последовательности обращений к зарегистрированным интерфейсам. Методы в классе модели, являющиеся операциями, задаются с помощью макросов *DECLARE_STIMULUS(имя_метода)* и *DECLARE_REACTION(имя_метода)*. В целом они не отличаются, но первые предполагается использовать для определения собственно операций, а вторые — для частей операций, требующих чтения выходных данных с выходных интерфейсов.

Описание класса эталонной модели (*fifo_model.h*) может выглядеть, например, так:

```
#include <hw/model.hpp>
#include <fifo_msg.h>
namespace cpptesk {
namespace fifo {

MODEL(FIFO) {
public:
  FIFO();
  virtual ~FIFO();
  DECLARE_INPUT(iface1);
  DECLARE_OUTPUT(iface2);
  virtual bool isIface1Ready() const;
  DECLARE_STIMULUS(push_msg); // Операция «Положить данные»
  DECLARE_STIMULUS(pop_msg); // Операция «Извлечь данные»
  void push_item(int data); // Функция эталонной модели «Положить данные»
  uint8_t pop_item(void); // Функция эталонной модели «Извлечь данные»
protected:
  std::vector<uint8_t> fifo;
};
}}
```

Теперь несколько слов о реализации методов класса эталонной модели. В конструкторе необходимо вызвать конструкторы интерфейсов, у которых единственный параметр - это текстовое описание данного интерфейса. Рекомендуется использовать осмысленные имена, так как потом они будут встречаться в диагностических сообщениях. Внутри конструктора также необходимо зарегистрировать входные интерфейсы с помощью макроса *ADD_INPUT(имя_интерфейса)*, а выходные интерфейсы - с помощью *ADD_OUTPUT(имя_интерфейса)*.

Переходим к описанию функций занятости интерфейсов: стандартно они содержат лишь использование метода isReady интерфейса и непосредственно возвращают его значение: *return имя_интерфейса.isReady()*.

Теперь опишем операции. Операции начинаются со строки копирования входного сообщения в локальную переменную с помощью макроса *CAST_MESSAGE(тип_входного_сообщения)*. Далее сообщение может быть передано на интерфейс, который был указан при запуске операции. Если оно посылается на входной интерфейс, мы говорим о запуске стимула (в обратном случае - о запуске реакции). Если мы хотим отправить сообщение на входной интерфейс со стандартными параметрами, т. е. без изменения сообщения, без изменения имени интерфейса (будет использован тот, который был указан при запуске операции), то запуск стимула делается с помощью макроса *START_STIMULUS(режим_запуска)*. Режимы запуска:

- *PARALLEL* — отправка сообщения и обработка результата этой отправки будет обработана в виде отдельного процесса, а этот процесс будет исполняться параллельно с командами, записанными в операции после START_STIMULUS;

- *SEQUENTIAL* — когда выполнение операции будет остановлено до исполнения работы стимула и всех процессов, которые он запускает.

Посылаемое сообщение будет *сериализовано* — то есть превращено в последовательность воздействий на тестируемый модуль через его сигнальные линии. Данную работу выполняют *сериализаторы* в *медиаторе* (см. след. этап). Если посылаемое сообщение или используемый интерфейс отличны от пришедших в эту операцию в качестве входных параметров (*что именно передается в операцию будет ясно немного дальше, на этапе создания сценариев*), необходимо использовать макрос *RECV_STIMULUS(режим_запуска, имя_интерфейса, имя_объекта_сообщения)*. При этом операция должна начинаться с *START_PROCESS()*, а не со START_STIMULUS(режим_запуска) и заканчиваться *STOP_PROCESS()* вместо *STOP_STIMULUS()* (*см. в конце абзаца*). В описании операции может содержаться макрос *CYCLE()*, который говорит тестовой системе, что необходимо сделать перерыв в один модельный такт в исполнении данной операции. Операция завершается с помощью макроса *STOP_STIMULUS()*.

Части операции, которые считывают данные с выходных интерфейсов, принято выделять в отдельные методы и определять их как «реакции» (*DEFINE_REACTION*). В методе реакции эталонная модель сообщает тестовой системе, что у нее есть определенное сообщение, которое теперь тестовая система пытается найти на выходном интерфейсе тестируемого модуля. Метод реакции сообщает о таком положении дел с помощью макроса *SEND_REACTION(режим_запуска, имя_интерфейса, имя_объекта_сообщения)*. Обратим внимание, что SEND_REACTION посылает сообщения только на выходные интерфейсы! Стандартные интерфейс и объект сообщения можно получить с помощью макросов *GET_IFACE()* и *GET_MESSAGE()* (под стандартными понимаются те, которые были переданы в метод в качестве входных параметров). При выделении методов реакций их необходимо вызывать из стимулов. Для этого делается вызов метода следующим образом: *имя_метода_реакции(process, имя_интерфейса, имя_объекта_сообщения)*, где process - это неявно заданный контекст выполнения операции, передавать его нужно для связывания всех частей операции в единое целое.

Разработка функциональной модели тестируемого компонента (скорее даже ее части: модели управляющей логики) не является специфической задачей создания тестовой системы с помощью C++TESK Testing ToolKit, она делается средствами C++, поэтому подробные комментарии по поводу ее создания здесь будут опущены.

В результате всех приведенных шагов получается, например, следующий файл реализации методов модели (*fifo_model.cpp*):

```
#include <fifo_model.h>
namespace cpptesk {
namespace fifo {

FIFO::FIFO() {
  ADD_INPUT(iface1);
  ADD_OUTPUT(iface2);
}

FIFO::~FIFO() {}

bool FIFO::isIface1Ready() const { return iface1.isReady(); }

DEFINE_STIMULUS(FIFO::push_msg) {
  InputData data = CAST_MESSAGE(InputData);
  push_item(data.get_data());
```

```
  START_STIMULUS(PARALLEL);
  CYCLE(); // вставлен для примера, необязателен
  STOP_STIMULUS();
}

DEFINE_STIMULUS(FIFO::pop_msg) {
  InputData data = CAST_MESSAGE(InputData);
  START_STIMULUS(PARALLEL);
  OutputData outdata = OutputData(pop_item());
  get_pop_msg(process, iface2, outdata);
  STOP_STIMULUS();
}
DEFINE_REACTION(FIFO::get_pop_msg) {
  SEND_REACTION(SEQUENTIAL, GET_IFACE(), GET_MESSAGE());
}

void FIFO::push_item(uint8_t data) {
  assert(fifo.size() < 16);
  fifo.push_back(data);
}

uint8_t FIFO::pop_item(void) {
  assert(fifo.size() > 0);
  uint8_t data = fifo[0];
  fifo.erase(fifo.begin());
  return data;
}

}}
```

3. Вспомогательный инструмент Veritool

Цифровые схемы разрабатываются на *языках проектирования аппаратуры* (*HDLs, Hardware Description Languages*), таких как *Verilog*, *VHDL* и т.д. В данном курсе мы будем использовать только язык Verilog (http://ru.wikipedia.org/wiki/Verilog), как один из наиболее часто используемых при разработке аппаратуры.

Для возможности создания тестовых систем на языке, отличном от Verilog, в стандарте Verilog предусмотрен специальный интерфейс *VPI* (*Verilog Procedural Interface*).

Для автоматического создания файлов *VPI-окружения* (под окружением понимается набор функций, реализованных в соответствии со стандартом языка Verilog, используемых для связи Verilog-симулятора и тестовых систем, созданных на C/C++) мы будем использовать инструмент *Veritool* (http://forge.ispras.ru/projects/veritool/files), распространяемый под лицензией GPL. Veritool использует синтаксический анализатор языка Verilog, который встроен в свободно распространяемый симулятор Icarus Veritog, поэтому необходима его предварительная установка (http://sourceforge.net/projects/iverilog/). Установка обоих приложений может осуществляться как вручную, так и автоматизированно с помощью скрипта, находящегося в поставке C++TESK Testing Toolkit.

3.1 Опции командной строки инструмента Veritool

Утилита Veritool имеет следующий формат запуска:

$ veritool [*опции*] *входные-файлы*

Ниже перечислены поддерживаемые опции командной строки:

1) --module=*module* – задает имя тестируемого Verilog-модуля, для которого генерируются компоненты тестовой системы. Если эта опция отсутствует, обрабатывается только первый модуль, определенный во входных файлах.

2) --clk=*signal* – задает имя сигнала синхроимпульса. Если эта опция отсутствует, считается, что таким сигналом является clk. Опция --clk имеет смысл, только если генерируется Verilog-окружение (указана опция --all или --testbench).

3) --rst=*signal* – задает имя сигнала сброса. Если эта опция отсутствует, считается, что таким сигналом является rst. Опция --rst имеет смысл, только если генерируется Verilog-окружение (указана опция --all или --testbench).

4) --rstpos – задает активный уровень сигнала сброса, равный логической единице. При отсутствии флага активным уровнем считается логический ноль.

5) --all – включает генерацию всех компонентов тестовой системы: Verilog-окружения, VPI-медиатора, VPI-функций и C-модели интерфейса. Установка этой опции эквивалентна одновременной установке --testbench, --vpi-media, --vpi-systf и --interface (без параметров) или отсутствию всех этих опций.

6) --testbench[=*file*] – включает генерацию Verilog-окружения и может задавать имя соответствующего файла. Если имя файла не указано, Verilog-окружение сохраняется в файле testbench.v.

7) --vpi-media[=*file*] – включает генерацию VPI-медиатора и может задавать имя соответствующего файла. Если имя файла не указано, VPI-медиатор сохраняется в файле vpi_media.c.

8) --vpi-media-header=*file* – задает имя заголовочного файла VPI-медиатора. Данная опция учитывается, только если установлена опция --vpi-media. Если опция --vpi-media-header отсутствует (при установленной --vpi-media), имя заголовочного файла получается из имени основного файла заменой расширения .h.

9) --vpi-systf[=*file*] – включает генерацию шаблонов VPI-функций и может задавать имя соответствующего файла. Если имя файла не указано, шаблоны VPI-функций сохраняются в файле vpi_systf.c.

10) --vpi-systf-header=*file* – задает имя заголовочного файла для шаблонов VPI-функций. Данная опция учитывается, только если установлена опция --vpi-systf. Если опция --vpi-systf-header отсутствует (при установленной --vpi-systf), имя заголовочного файла получается из имени основного файла заменой расширения на .h.

11) --interface[=*file*] – включает генерацию C-модели интерфейса и может задавать имя соответствующего файла. Если имя файла не указано, C-модель интерфейса сохраняется в файле interface.c.

12) --interface-header=*file* – задает имя заголовочного файла C-модели интерфейса. Данная опция учитывается, только если установлена опция --interface. Если опция --interface-header отсутствует (при установленной --interface), имя заголовочного файла получается из имени основного файла заменой расширения на .h.

13) --destination=*dir* – задает выходной каталог. Если эта опция не указана, файлы сохраняются в текущий каталог.

14) --version | -v – вывод информации о версии утилиты.

15) --help | -h – вывод информации о доступных опциях.

3.2 Запуск Veritool

Запуск инструмента осуществляется, например, следующим образом:

$(PATH_TO_VERITOOL)/bin/veritool --clk=clock --rst=reset --rstpos target/src/myfifo.v --vpi-systf=vpi_systf.cpp --vpi-media=vpi_media.cpp --interface=interface.cpp

Обратите внимание! Названия сигналов clk и rst надо задавать в точности такими, как они названы в тестируемом модуле. Инструмент рассчитан на то, что такие сигналы присутствуют в модуле в единственном экземпляре. Возможные названия: clock, CLOCK, clk, CLK, reset, RESET, rst, RST и т.д.

4. Разработка адаптера эталонной модели

Мы продолжаем разработку тестового оракула. На данном этапе нам необходимо создать *адаптер эталонной модели* (в данном случае слова *адаптер* и *медиатор* будут являться синонимами). Адаптер является классом-наследником эталонной модели и предоставляет средства:

- соединения с тестируемой реализацией для посылки стимулов и приема реакций;

- прослушивания выходных интерфейсов тестируемой реализации на предмет «лишних» реакций;

- сопоставления реакций, пришедших от тестируемой реализации, и реакций, пришедших от эталонной модели, с выводом подробной диагностики.

Структура адаптера эталонной модели представлена на рис. 4.

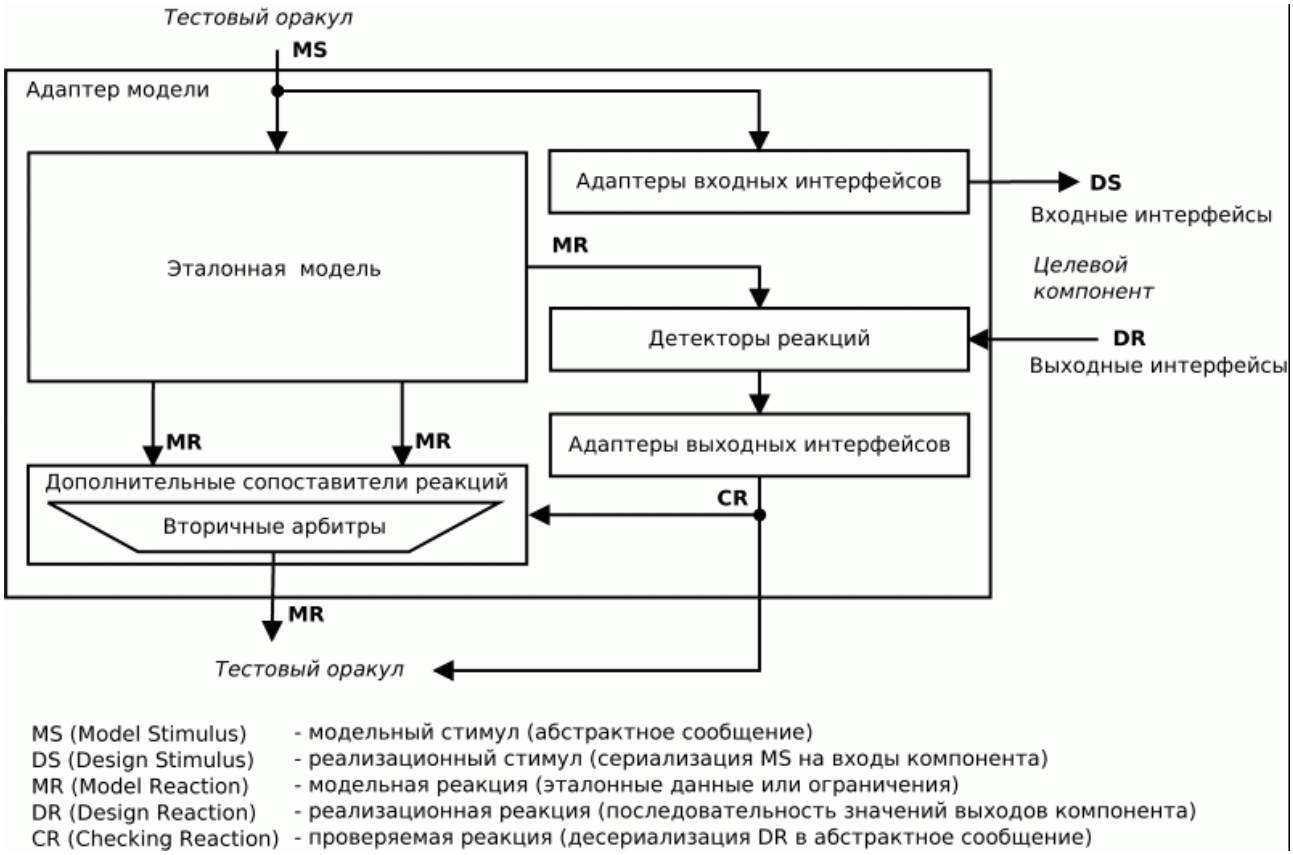

Рис. 4. Обобщенная структура медиатора эталонной модели

Основными элементами медиатора являются:

- унаследованная эталонная модель;

- *адаптеры входных интерфейсов* (или *сериализаторы*), осуществляющие преобразование модельных стимулов в реализационные стимулы, что выражается конвертированием абстрактных сообщений в последовательность воздействий сигнальных линий тестируемого компонента;

- *адаптеры выходных интерфейсов* (или *десериализаторы*), осуществляющие преобразование реализационных реакций в проверяемые реакции, что выражается конвертированием принимаемых последовательностей с выходных сигнальных линий тестируемого компонента в абстрактные сообщения;

- *детекторы реакций*, необходимые для отлова принимаемых реакций от тестируемого компонента; детекторы создаются отдельно для каждого выходного интерфейса и с помощью модельных реакций отлеживают пришедшие реализационные реакции;

- *дополнительные сопоставители реакций* (*вторичные арбитры*), осуществляющие окончательное утверждение соответствия между пришедшими реализационными и модельными реакциями (в случае, если не выполняется соотношение 1:1).

4.1 Создание класса адаптера

Все необходимые классы и макросы для создания медиатора находятся в файле **<hw/media.hpp>**.

Обычно для создания медиатора требуются вспомогательные структуры, описывающие VPI-интерфейс тестируемого модуля. Такие структуры ранее были созданы с помощью инструмента Veritool (файл interface.h).

Медиатор модели является наследником класса модели и определяется следующим макросом: *MEDIA(имя_класса_медиатора, имя_класса_модели)*. Синонимом слова "медиатор" в тестовой системе является "адаптер": макрос *ADAPTER* приводит к аналогичному результату, что и MEDIA. Внутри класса сначала обычно создаются стандартные конструктор и виртуальный деструктор, а также *максимальное время ожидания установленной реакции* (назовем его REACTION_TIMEOUT, это обычное целочисленное поле). Также в класс медиатора добавляются новые функции проверки занятости интерфейсов, перегружающие функции эталонной модели.

С помощью макроса *DECLARE_PROCESS(имя_сериализатора)* необходимо определить название метода, который будет преобразовывать сообщения, приходящие от эталонной модели на каждый входной интерфейс, определенный в модели, в последовательность тестовых воздействий. Данный метод называется *сериализатором*. Сериализаторы создаются для каждого входного интерфейса.

С помощью того же макроса *DECLARE_PROCESS(имя_десериализатора)* определяется название метода, который будет считывать сообщения, поступающие от тестируемого модуля на его выходные интерфейсы, по запросу от эталонной модели, которая будет генерировать сообщения, подаваемые на интерфейсы, определенные как выходные. Данный метод называется десериализатором. Десериализаторы создаются для каждого выходного интерфейса.

В ряде случаев на один выходной интерфейс может быть зарегистрировано несколько сообщений от эталонной модели и может получиться так, что медиатор не сможет определить, к какому именно эталонному сообщению относится данное реализационное. Тогда используется функция указания следования модельных сообщений (также называемый арбитр), который определяется следующим макросом: *DECLARE_ARBITER(тип_следования_модельных_сообщений, имя_функции_указания_следования)*. Среди типов функций указания следования можно выделить следующие:

- *FIFO* (все сообщения выстраиваются в очередь по мере их регистрации, поэтому предпочтение при выборе необходимого модельного сообщения будет отдано самому первому),

- *Priority* (аналогичен FIFO, но сообщения каждый раз перед выбором упорядочиваются по приоритету),

- *InterfaceOrdering* (аналогичен FIFO, но сообщения из разных интерфейсов находятся в разных очередях, а выбор происходит между первыми элементами всех очередей).

Также для выходных интерфейсов поддерживается возможность отображения *неожиданных реакций*. Реакция является неожиданной, если медиатору не удалось установить соответствие между реакцией, пришедшей от эталонной модели и реакцией, поступившей на выходной интерфейс тестируемого устройства. Это может произойти, если первой реакции не поступило, либо, если способ установления соответствия не дал положительного ответа, что пришедшие реакции являются идентичными. Отлов неожиданных реакций осуществляется с помощью детектора реакций, в большинстве случаев определяемого стандартно с помощью

макроса *DEFINE_BASIC_OUTPUT_LISTENER(имя_детектора, объект_типа_выходной_интерфейс, предикат_условие_установления_соответствия)*.

Помимо всех этих функций необходимо определить дополнительный набор функций, стандартных для тестирования Verilog-модулей:

- *virtual void initialize()* - функция инициализации тестового окружения, которая подготавливает тестируемую и тестовую систему к началу верификации;
- *virtual void finalize()* - функция завершения процесса тестирования, которая устанавливает определенные сигналы на тестируемую модель, чтобы остановить ее симулятор, а также может содержать завершающие команды для тестовой системы;
- *inputs_t inputs* — структура, поля которой соответствуют входным сигналам тестируемого устройства (сама структура генерируется автоматически);
- *virtual void setInputs()* - функция установки входных сигналов тестируемого устройства из структуры inputs;
- *outputs_t outputs* — структура, поля которой соответствуют входных сигналам тестируемого устройства (сама структура генерируется автоматически);
- *virtual void getOutputs()* - функция установки полей структуры outputs в соответствии с выходными сигналами тестируемого устройства на данном такте;
- *virtual void simulate()* - передача управления симулятору для имитационного моделирования одного такта.

Однако, если мы используем макрос *VERITOOL_ADAPTER* для определения адаптера эталонной модели, эти функции и поля будут созданы автоматически.

В результате данных действий получается, например, следующий заголовочный файл медиатора (*fifo_media.h*):

```
#pragma once
#include <hw/veritool/media.hpp>
#include <model.h>
#include <interface.h>
#include <string>

namespace cpptesk {
namespace fifo {

VERITOOL_ADAPTER(FIFOMediator, FIFO)
{
public:
  FIFOMediator();
  virtual ~FIFOMediator();

  const static int REACTION_TIMEOUT = 100;
  virtual bool isIface1Ready() const;
  DECLARE_PROCESS(serialize_iface1);
  DECLARE_PROCESS(deserialize_iface2);
  DEFINE_BASIC_OUTPUT_LISTENER(listen_iface2, iface2, outputs.DO_STROBE);
  DECLARE_ARBITER(Priority, matcher_iface2);
};
}}
```

Перейдем к реализации методов класса медиатора эталонной модели.

В реализации конструктора необходимо привязать имена входных и выходных интерфейсов к их *адаптерам* — сериализаторам и десериализаторам соответственно. Это делается с помощью:

- макроса *SET_INPUT_ADAPTER(имя_объекта_класса_интерфейса, имя_метода_адаптера_с_указанием_класса)* для входных интерфейсов;

- макроса *SET_OUTPUT_ADAPTER(имя_объекта_класса_интерфейса, имя_метода_адаптера_с_указанием_класса)* для выходных интерфейсов.

С помощью вызова функции *setReactionTimeout(максимальное_время)* необходимо установить максимальное время в тактах, которое зарегистрированная модельная реакция будет ждать на выходном интерфейсе. Здесь можно использовать ранее определенное поле *REACTION_TIMEOUT*. Если реакция тестируемого устройства не появилась за указанное количество тактов, фиксируется ошибка слишком большого времени ожидания.

В медиаторе необходимо инициализировать структуры inputs и outputs вызовом функций *clear_inputs(&inputs)* и *clear_outputs(&outputs)*, сгенерированные инструментом Veritool.

С помощью макроса *SET_ARBITER(имя_объекта_выходного_интерфейса, имя_функции_указания_следования_зарегистрированных_реакций_для_интерфейса)* необходимо привязать ранее определенную функцию указания следования модельных реакций к выходному интерфейсу.

С помощью макроса *CALL_OUTPUT_LISTENER(имя_объекта_детектора_с_указанием_класса, имя_выходного_интерфейса)* запускается детектор лишних реакций.

Для включения вывода информации о найденных ошибках на консоль необходимо вызвать стандартную функцию класса медиатора, *debug(уровень_вывода_информации_об_ошибках)*. На данном этапе допускается указывать только уровень *DEBUG_ERROR*.

Функции проверки занятости интерфейсов обычно перегружаются, но используют функции класса-родителя, то есть эталонную модель. К реализации этих функций можно добавить использование выходных сигналов тестируемой модели (outputs), если это необходимо.

Определение сериализаторов начинается с макроса *DEFINE_PROCESS(имя_сериализатора_с_именем_класса)*. В начале описания реализации сериализатора необходимо скопировать модельное сообщение в локальную переменную. Доступ к модельному сообщению осуществляется через макрос *CAST_MESSAGE(имя_класса_сообщения)*. После копирования сообщения начинается непосредственная реализация сериализатора: пишется макрос *START()*, с него начинается обработка стимула. С помощью вызова функции *iface.capture()* «захватывается» данный входной интерфейс. Этот захват является логическим и необходим для того, чтобы несколько различных операций не пытались одновременно использовать один и тот же входной интерфейс. При необходимости проверить состояние входного интерфейса перед отправкой туда стимула используются ранее определенные функции *isReady**. Далее происходят установки полей структуры, соответствующей входным сигналам тестируемого модуля (inputs) в соответствии с полями сообщения модельного стимула. После того, как все воздействия поданы, интерфейс необходимо освободить командой *iface.release()*, причем делать это надо за 1 такт до его реального освобождения для предотвращения появления пробелов в подаче стимулов. Сдвиг такта в стимуле осуществляется макросом CYCLE().

После iface.release() обычно пишется конструкция STOP(), обозначающая окончание обработки стимула. Необходимо, чтобы между iface.capture() и STOP() находился хотя бы один CYCLE().

Определение десериализаторов начинается с макроса *DEFINE_PROCESS(имя_десериализатора_с_именем_класса)*. В начале описания реализации десериализатора необходимо завести переменную-ссылку на пришедшее модельное сообщение. В это сообщение будет записана реакция тестируемого устройства. Макрос *START()* обозначает начало обработки реакции. После него пишется условие соответствия пришедшей реакции тестируемого устройства данному медиатору с помощью макроса *WAIT_REACTION(условие_соответствия)*. Часто в этой роли выступает просто сигнал валидности, но, поскольку вызов медиатора происходит по модельной реакции, здесь могут использоваться некоторые поля этой модельной реакции. Отметим, что, если несколько реакций попытаются считывать одни и те же выходные данные, то есть они захватят один и тот же выходной интерфейс, будет зафиксирована ошибка, это действие считается запрещенным. После того, как реакция нашлась, происходит считывание значений полей outputs и запись их в локальное сообщение. При необходимости между считываниями значений выходных сигналов вставляется макрос CYCLE(). После окончания считывания пишется макрос *NEXT_REACTION()*, который говорит тестовой системе, что необходимо перейти на следующую зарегистрированную модельную реакцию на данном интерфейсе, при условии, что она есть. Наконец, макрос STOP() обозначает окончание работы десериализатора.

В результате мы получаем, например, следующую реализацию медиатора эталонной модели (*fifo_media.cpp*):

```
#include <fifo_media.h>
#include <sync.h>
#include <interface.h>
#include <vpi_media.h>
#include <iostream>

namespace cpptesk {
namespace fifo {

FIFOMediator::FIFOMediator() {
  SET_INPUT_ADAPTER(iface1, FIFOMediator::serialize_iface1);
  SET_OUTPUT_ADAPTER(iface2, FIFOMediator::deserialize_iface2);

  setReactionTimeout(FIFOMediator::REACTION_TIMEOUT);

  SET_ARBITER(iface2, matcher_iface2);
  CALL_OUTPUT_LISTENER(FIFOMediator::listen_iface2, iface2);
  debug(DEBUG_ERROR);
}

FIFOMediator::~FIFOMediator() {}

bool FIFOMediator::isIface1Ready() const {
  return FIFO::isIface1Ready();
}

DEFINE_PROCESS(FIFOMediator::serialize_iface1) {
  InputData msg = CAST_MESSAGE(InputData);
  START();
  iface.capture();
  inputs.WR_STROBE = 1;
  inputs.DI = msg.get_data();
```

```
  iface.release();
  CYCLE();
  STOP();
}

DEFINE_PROCESS(FIFOMediator::deserialize_iface2)
{
  OutputData &msg = CAST_MESSAGE(OutputData);
  START();
  WAIT_REACTION(outputs.DO_STROBE);
  msg.set_data(outputs.DO);
  NEXT_REACTION();
  STOP();
}

}}
```

5. Разработка сценариев тестирования

Сценарии тестирования являются неотъемлемой частью тестовой системы. Основной их целью является *описание стимулов* и выбор *метода* их использования. Сценарии тестирования служат конфигурацией для объектов класса генератор стимулов из библиотеки инструмента C++TESK Testing ToolKit. На обобщенной структуре тестовой системы на рис.5 настроенный сценарием тестирования генератор изображен под названием «генератор стимулов».

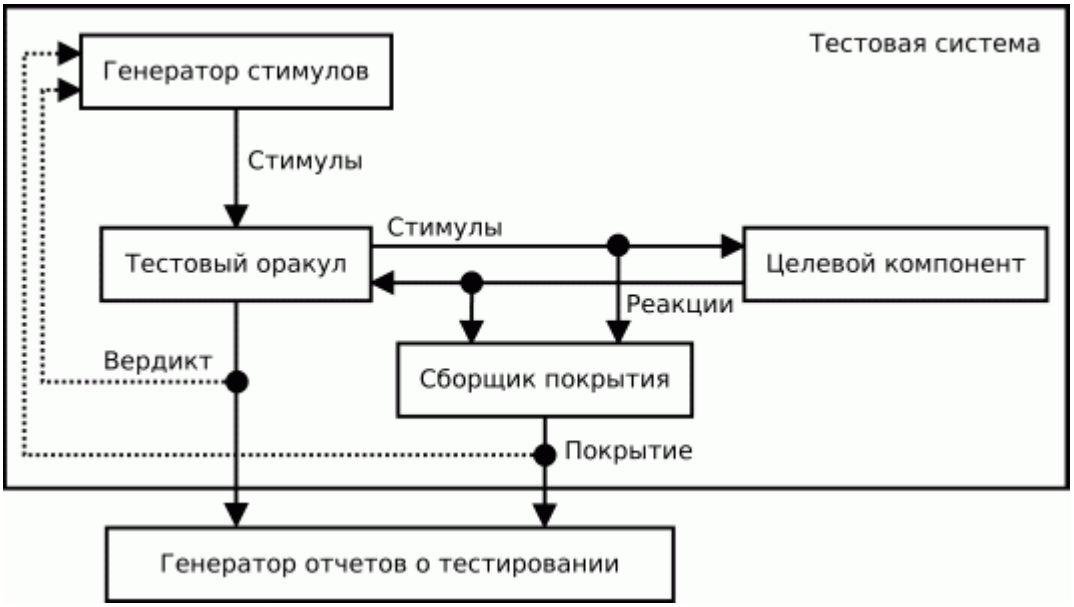

Рис. 5. Обобщенная структура тестовой системы

C++TESK Testing ToolKit поддерживает два основных способа генерации: на основе *обхода конечного автомата* (способ будет рассмотрен в одном из следующих разделов) и *случайный выбор стимула* из списка зарегистрированных стимулов. Оба способа имеют свои достоинства и недостатки. Первый предоставляет большую полноту тестирования, нацеленность на обход всех ситуаций, определенных конечным автоматом. Второй способ дает большую широту создаваемых воздействий при относительно небольшом времени, требуемом на разработку. При этом, первый способ, как правило, требует больших усилий при разработке и прогоне тестов. А, действуя вторым способом, едва ли можно охватить сложные ошибки, возникающие при специфических последовательностях подаваемых операций.

Вспомним, что предыдущими шагами по разработке тестовых систем были: эталонная модель, медиатор эталонной модели, сборщик покрытия. Для полноты набора компонентов, согласно рис. 5, необходимо, помимо генератора, иметь тестовый оракул. Он представляет собой расширенный с помощью *компонентов проверки* медиатор (или *адаптер*; наследник эталонной модели), см. рис.6. Этими компонентами являются:

- *Предусловия стимулов*;
- *Компараторы реакций*.

Предусловия стимулов являются предикатами, выполняющимися только в том случае, если текущее состояние модели позволяет запускать данный конкретный стимул. Таким образом, если предусловие выполнено стимул запускается на данном такте, если нет – он игнорируется. В зависимости от тестового сценария будет выбран другой стимул, либо сдвинут такт. Предусловия стимула можно размещать: внутри тестового сценария, либо внутри эталонной модели, используемой этим тестовым сценарием.

Компараторы реакций автоматически генерируются при создании медиатора эталонной модели.

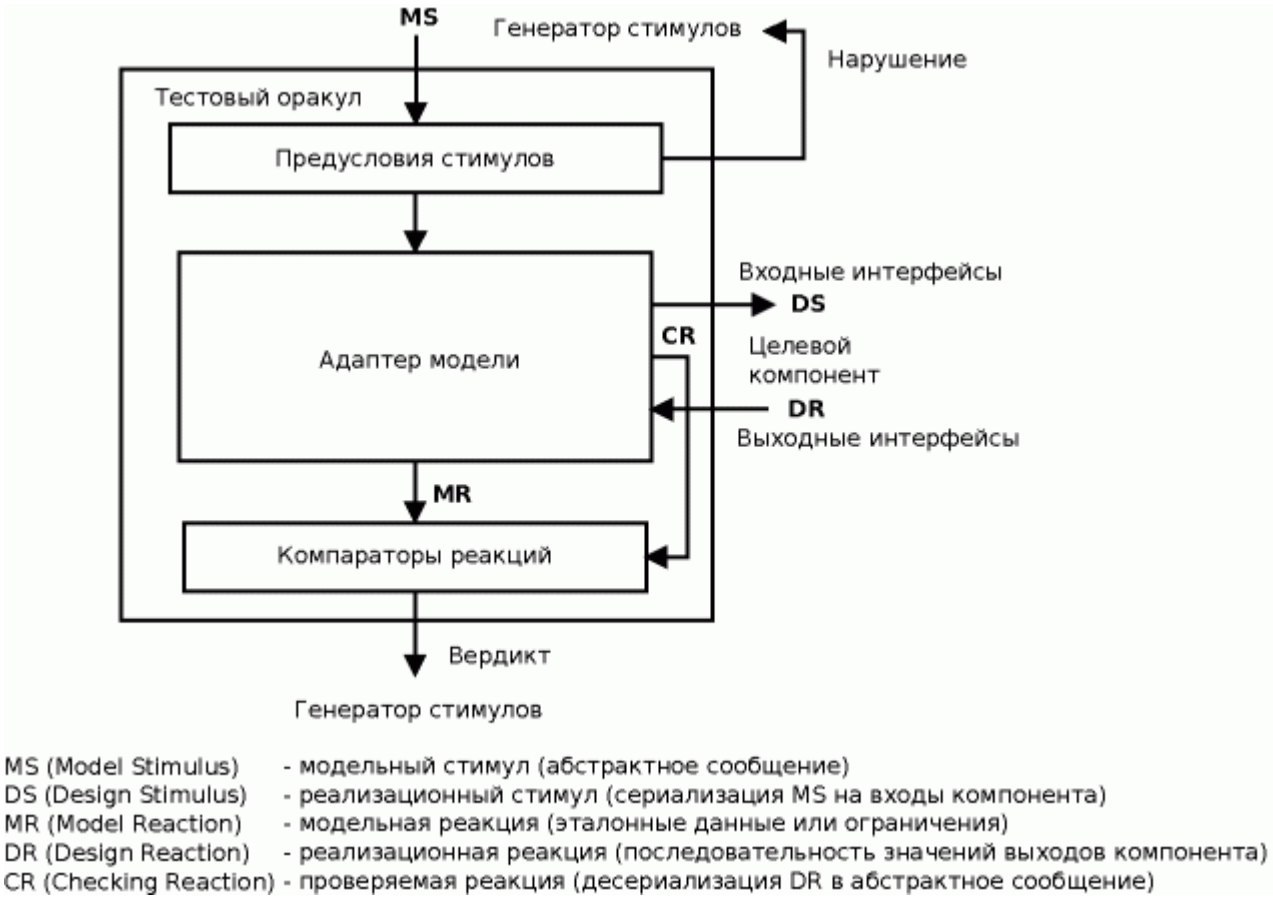

Рис. 6. Тестовый оракул

Таким образом, у нас есть все средства, чтобы перейти к разработке тестовых сценариев. Сценарии должны содержать:

- экземпляр эталонной модели;
- *сценарные функции*;
- (опционально) *функцию получения текущего состояния*.

Сценарные функции необходимы для создания модельных сообщений-стимулов и передачи их в эталонную модель, если выполнено предусловие (оно может проверяться и в эталонной модели).

Функция получения текущего состояния необходима в том случае, когда используется автоматный генератор стимулов. Возвращаясь к возможностям C++TESK Testing ToolKit, напомним, что генерируемые тестовые последовательности могут быть двух типов: *автоматные* и *случайные.* Первый тип будет рассмотрен позднее. Для генерации тестовых последовательностей на основе случайного выбора стимулов используются только сценарные функции, так как состояние эталонной модели тестируемого устройства не отслеживается. При использовании такого способа генерации необходимо придерживаться соглашения: при регистрации списка стимулов первым регистрируется стимул, *сдвигающий такт симуляции,* остальные стимулы этого делать не должны. При настройке генерации со случайным выбором стимулов можно указать:

- *максимальное количество тактов*, которое будет работать тест,
- *стратегию подачи стимулов (максимальная загрузка, динамическое изменение потока подаваемых стимулов*, а также их параметры).

Более подробно эти шаги будут освещены в последующих разделах.

Перейдем непосредственно к созданию сценариев. Все необходимые классы и макросы для создания сценариев определены в заголовочном файле **<ts/scenario.hpp>**. Используемые пространства имен: cpptesk::ts, cpptesk::hw, cpptesk::tracer.

Класс сценария создается с помощью макроса *SCENARIO(имя_класса_сценария) {}*. Следующие функции являются стандартными при определении сценария:

- конструктор и виртуальный деструктор;
- *virtual bool init(int argc, char ** argv)* — инициализационная функция;
- *virtual void finish()* — функция, запускаемая при окончании тестирования;
- функция получения текущего состояния *std::string get_state() (пока мы вместо нее будем использовать заглушку);*
- сценарные функции со следующей сигнатурой *bool имя_сценарной_функции(IntAbcCtx& ctx)*;
- экземпляр тестируемой эталонной модели с медиатором.

Получившийся класс является базовым для построения тестовых сценариев. От него можно наследоваться, получая различные дочерние тестовые сценарии. В этих сценариях регистрируются только необходимые сценарные функции, находящиеся в базовом классе, используется необходимая функция получения текущего состояния, определенная также в базовом классе. Никаких функций кроме конструктора и деструктора класс-наследник обычно не требует.

Декларация классов сценария может выглядеть, например, следующим образом:

```
#pragma once

#include <fifo_media.h>

#include <ts/scenario.hpp>

using namespace cpptesk::ts;
using namespace cpptesk::hw;
```

```
using namespace cpptesk::tracer;

namespace cpptesk {
namespace examples {

SCENARIO(ParentScenario) {
public:
  virtual bool init(int argc, char ** argv);
  virtual void finish();
  bool scen_nop(IntAbcCtx& ctx);
  bool scen_push(IntAbcCtx& ctx);
  bool scen_pop(IntAbcCtx& ctx);
  std::string get_current_state();

  ParentScenario(void);
  virtual ~ParentScenario(void);

protected:
  DUTMediator dut;
};

class ChildScenario : public ParentScenario {
public:
  ParentScenario();
  virtual ~ParentScenario();
};
}}
```

После разработки класса необходимо реализовать его методы. В конструкторе базового класса, если он напрямую не будет использоваться в качестве сценария, необходимо лишь указать вызов конструктора его базового класса *ScenarioBase()*. В конструкторе сценария-наследника определяется вызов конструктора его базового класса, а также вызывается следующая функция:

setup(строка_с_названием_сценария, &название_функции_инициализации_с_именем_класса, &название_функции_окончания_работы_с_именем_класса, &название_функции_получения_текущего_состояния_с_именем_класса).

Ограничением на использование функции setup на данный момент является требование нахождения всех используемых в ней функций в одном и том же классе. Сразу после функции setup в конструкторе можно регистрировать
сценарные функции с помощью макроса
ADD_SCENARIO_METHOD(название_сценарной_функции_с_именем_класса).

Сценарные функции разрабатываются отдельными методами класса базового сценария. В самом начале сценарной функции могут задаваться и обычные локальные переменные, причем заметим, что желательно их объявлять только в начале сценарной функции.

После определения всех переменных пишется макрос *IBEGIN*. После IBEGIN пишется макрос *IACTION{}*, внутри которого сообщение, которое будет подано в эталонную модель, создается, а затем и подается в эту модель. После создания сообщения можно написать простую реализацию предусловия подачи стимула: условный оператор if, проверяющий необходимые параметры эталонной модели на предмет возможности подачи именно этого созданного сообщения. Внутри условного оператора при удачном исходе проверки, происходит подача сообщения в модель с помощью следующей конструкции:
экземпляр_эталонной_модели.start(&имя_стимула_модели_с_именем_класса, название_интерфейса, сообщение, имя_класса_модели::START_REQUEST_IFACE).

Вне зависимости от возможности подачи стимула внутри IACTION, в самом его конце, необходимо включить написать *YIELD(экземпляр_эталонной_модели.verdict())*, который будет получать результат работы эталонной модели на данном такте и, при нахождении ошибки, ее фиксировать. В случае сценарной функции, сдвигающей такт моделирования, вместо конструкции экземпляр_эталонной_модели.start(…) используется вызов функции *экземпляр_эталонной_модели.cycle()*.

Возвращаясь к методам тестового сценария, отметим, что метод *init* обычно пуст и возвращает true, а метод *finish* включает остановку симуляции по вызову медиаторной функции *экземпляр_эталонной_модели.finalize()*. *Функция получения текущего состояния get_state* берет некоторые значения, возвращаемые функциями эталонной модели, создает на их основе строку и возвращает ее. Обычно правильное написание функции текущего состояния является непростой творческой работой.

Реализация методов класса тестового сценария может выглядеть примерно так:

```
#include <fifo_scen.h>

#include <tracer/tracer.hpp>

namespace cpptesk {
namespace examples {

bool ParentScenario::init(int argc, char ** argv) {
  return true;
}

void ParentScenario::finish() {
  dut.finalize();
}

bool ParentScenario::scen_nop(IntAbcCtx& ctx) {
  IBEGIN
  IACTION {
    dut.cycle();
    YIELD(dut.verdict());
  }
  IEND
}

bool ParentScenario::scen_push(IntAbcCtx& ctx) {
  IBEGIN
  IACTION {
    if(dut.isIface1Ready() && !dut.is_full()) {
      InputData msg = InputData();
      dut.start(&DUT::push_msg, dut.iface1, msg, DUT::START_REQUEST_IFACE);
    }
    YIELD(dut.verdict());
  }
  IEND
}

bool ParentScenario::scen_pop(cpptesk::ts::IntAbcCtx& ctx) {
  IBEGIN
  IACTION {
    if(dut.isIface2Ready() && !dut.is_empty()) {
      InputData msg = InputData();
      dut.start(&FIFO::pop_msg, dut.iface2, msg, FIFO::START_REQUEST_IFACE);
    }
    YIELD(dut.verdict());
```

```
  }
  IEND
}

std::string ParentScenario::get_state_fifo() {
  std::string state;
  std::stringstream out;
  out << "";
  state = out.str();
  return state;
}

ParentScenario::ParentScenario(void) : ScenarioBase() {}

ParentScenario::~ParentScenario(void) {}

ChildScenario::ChildScenario(void) : ParentScenario() {
  setup("Example scenario", &ParentScenario::init, &ParentScenario::finish,
      &ParentScenario::get_state_fifo);
  ADD_SCENARIO_METHOD(ParentScenario::scen_nop);
  ADD_SCENARIO_METHOD(ParentScenario::scen_push);
  ADD_SCENARIO_METHOD(ParentScenario::scen_pop);
}

ChildScenario::~ChildScenario(void) { };
}}
```

Теперь тестовый сценарий уже создан, но все еще необходимо связать генератор стимулов с данным сценарием. Это делается посредством *репозитория тестов*. Репозиторий содержит список экземпляров классов тестовых сценариев и сопоставляет каждому из них выбранный генератор стимулов и параметры запуска этого генератора. В каждой строке репозитория должны содержаться различные экземпляры классов тестовых сценариев: название объекта тестового сценария используется для идентификации текущего запускаемого сценария. При запуске теста оно передается как параметр +scen=*имя_сценария* (см. файл testbench.v, сгенерированный Veritool).

Для создания репозитория используются макросы, определенные в заголовочном файле *<utils/testreg.h>*. Репозиторий начинается с макроса *TEST_REGISTRY_BEGIN*, завершается макросом *TEST_REGISTRY_END*. Каждое сопоставление сценария и генератора осуществляется внутри репозитория с помощью макроса *REGISTER_TEST(экземпляр_класса_тестовый_сценарий, тип_генератора_стимулов, параметры_по_умолчанию)*. Параметр тип_генератора_стимулов может принимать только два значения: *engine::fsm* и *engine::rnd*. Первый обозначает генерацию на основе обхода конечного автомата, второй – с произвольным выбором стимулов. С помощью параметров_по_умолчанию производится дополнительная настройка генератора стимулов. Список параметров может включать, но ими не ограничен, следующие параметры:

- *-uerr=N, N>0* – выполнять тестирование до ошибки номер N;
- *-nt* – запрет использовать трассу *UTT*;
- *-nt2* – запрет использовать трассу *UTT2*;
- *--length N, N>0* – сценарий случайного тестирования будет продолжаться N тактов
- *--parallel* – включена возможность подавать стимулы параллельно (только для сценария случайного тестирования)
- *--maxload* – необходимо постоянно подавать все стимулы, у которых выполнены предусловия (только для сценария случайного тестирования).

Указываемые в репозитории параметры теста перекрываются пользовательскими параметрами, подаваемыми через ARGV (см. сгенерированный testbench.v с помощью Veritool).

Разработанный репозиторий (*fifo_tests.cpp*) может выглядеть, например, вот таким образом:

```
#include <utils/testreg.h>
#include <fifo_scen.h>

using namespace cpptesk::examples::fifo;

ChildScenario scen_rnd;

TEST_REGISTRY_BEGIN
REGISTER_TEST(scen_rnd, engine::rnd, "-uerr=1 --length 10000 --parallel")
TEST_REGISTRY_END
```

6. Разработка функционального покрытия

Дополнительной возможностью построения эталонной модели является построение и оценка *функционального покрытия*. Под покрытием обычно понимается некоторая величина, измеряемая в абсолютных или относительных величинах. Смысл сбора покрытия заключается в получении информации о полноте тестирования: чем больше абсолютная и относительная величина покрытия, тем более полно проверен тестируемый проект.

Существует несколько разных подходов к сбору покрытия, но в рамках знакомства с C++TESK Testing ToolKit мы будем говорить только о *функциональном покрытии*, покрытии, отвечающем на вопрос о полноте проверки выделенных инженером *функциональных свойств* тестируемой модели.

В текущей версии C++TESK Testing ToolKit функциональное покрытие задается вручную в виде набора ситуаций; далее возникновение этих ситуаций автоматически учитывается инструментом и выводится в виде отчета в конце тестирования.

Инженеру, занятому верификацией, необходимо проанализировать документацию на тестируемый компонент с целью выделения ситуаций, возникающих в функциональной модели, которые должны обязательно покрываться тестом. Для согласования извлеченных сведений с особенностями реализации тестируемого компонента полезно обратиться к его разработчику. Далее инженером для выделенных ситуаций создается структура функционального покрытия, которая в процессе тестирования будет динамически заполняться. Отметим, что работа по выявлению элементов покрытия является творческой, и ее результаты сильно зависят от квалификации занимающегося этим человека.

Необходимо отметить, что C++TESK Testing ToolKit на данном этапе его развития не позволяет описывать ситуации, в которых используются динамические характеристики: наступление события после какого-то другого события, через n тактов и т.д. Такие возможности планируется добавить по мере развития инструмента.

Все необходимые макросы и классы описаны в файлах **<ts/coverage.hpp>** и **<tracer/tracer.hpp>**.

Класс покрытия может являться частью эталонной модели, но для удобства разработки его предлагается создавать отдельным компонентом. Учитывая связь покрытия с эталонной моделью, внутри него необходимо хранить ссылку на эталонную модель.

Помимо обязательного конструктора и виртуального деструктора класс функционального покрытия должен включать объект класса CoverageTracker. Этот объект будет осуществлять хранение информации о зарегистрированных в нем структурах функционального покрытия, выводить структуру покрытия, а также информацию о достигнутом покрытии в файл в формате UTT2.

Задать покрытие можно с помощью макроса *DEFINE_ENUMERATED_COVERAGE(идентификатор_покрытия, строковое_представление_покрытия, ((метка_элемента_покрытия, строковое_представление_элемента_покрытия), (...,...), ...))*. При этом будет создан объект класса Coverage с именем *идентификатор_покрытия*. *Метка_элемента_покрытия* создается для уникальной идентификации всех компонентов определяемого покрытия. Все строковые представления используются для генерации отчетов. Инструмент позволяет также копировать покрытия, создавать покрытия-произведения, покрытия-произведения с вычитанием кортежей. Для этого используются макросы:
DEFINE_ALIAS_COVERAGE(идентификатор_покрытия, строковое_представление_покрытия, идентификатор_исходного_покрытия),
DEFINE_COMPOSED_COVERAGE(идентификатор_покрытия, строковое_представление_покрытия, идентификатор_исходного_покрытия_А, идентификатор_исходного_покрытия_Б),
DEFINE_COMPOSED_COVERAGE_EXCLUDING(идентификатор_покрытия, строковое_представление_покрытия, идентификатор_исходного_покрытия_А, идентификатор_исходного_покрытия_Б, {идентификатор_исключаемого_элемента_покрытия_А, идентификатор_исключаемого_элемента_покрытия_Б}, ...) .

Для учета покрытия необходимо сопоставить текущие параметры эталонной модели, а затем передать их в объект типа CoverageTracker посредством оператора <<. Это удобно делать с помощью двух функций. В декларации первой функции в качестве выходного значения должен стоять объект типа Coverage, а в реализации этой функции должен возвращаться элемент объекта типа Coverage, выбранный согласно используемому параметру эталонной модели. Вторая функция принимает возвращаемое значение первой функции и передает его в объект типа CoverageTracker с помощью оператора <<.

Заголовочный файл класса покрытия (*fifo_coverage.h*) может выглядеть, например, следующим образом:

```
#pragma once
#include <hw/model.hpp>
#include <ts/coverage.hpp>
#include <tracer/tracer.hpp>

#include <fifo_model.h>

using namespace cpptesk::ts;
using namespace cpptesk::hw;
using namespace cpptesk::tracer;
using namespace cpptesk::tracer::coverage;

namespace cpptesk {
namespace fifo {

class DUT;
class DUTCoverage {
public:
  DUTCoverage(DUT &dut): dut(dut) {}
```

```
  virtual ~DUTCoverage() {};
  CoverageTracker coverageTracker;
  DEFINE_ENUMERATED_COVERAGE(DUT_FULLNESS, "DUT fullness",
                                                ((I0, "0 (free)"), (I1, "1"),
(I2, "2"), (I3, "3"), (I4, "4 (full)")));
  DUT_FULLNESS select_coverage_element_DUT_FULLNESS(void) const;
  void update_coverageTracker_DUT_FULLNESS(void);
protected:
  FIFO &fifo;
};

}}
```

Реализация этого класса (*fifo_coverage.cpp*) может выглядеть, например, следующим образом:

```
#include <fifo_coverage.h>

namespace cpptesk {
namespace fifo {

DUTCoverage::DUT_FULLNESS
DUTCoverage::select_coverage_element_DUT_FULLNESS(void) const {
  switch(dut.dut_fullness()) {
    case 0:  return DUTCoverage::DUT_FULLNESS::I0;
    case 1:  return DUTCoverage::DUT_FULLNESS::I1;
    case 2:  return DUTCoverage::DUT_FULLNESS::I2;
    case 3:  return DUTCoverage::DUT_FULLNESS::I3;
    case 4:  return DUTCoverage::DUT_FULLNESS::I4;
    default: assert(false);
  }
}

void DUTCoverage::update_coverageTracker_DUT_FULLNESS(void) {
  coverageTracker << DUTCoverage::select_coverage_element_DUT_FULLNESS();
}

}}
```

Объект класса создаваемого покрытия должен быть создан в классе эталонной модели. Тогда в конструкторе эталонной модели появляется возможность связать используемую в классе покрытия ссылку на эталонную модель. Для обновления информации о покрытии необходимо вызывать функцию №2 из ранее созданных двух функций в классе покрытия в определенные моменты времени прогона тестовой системы. Такими моментами могут быть сдвиги такта, запуск каких-либо операций, в общем, это - события, когда происходят какие-либо действия, меняющие состояние модели, и в которых есть необходимость отследить функциональное покрытие.

Достигнутое покрытие выводится в трассу *utt2. Для анализа трассы используется специальный генератор отчетов, который можно вызвать командой *$CPPTESK_HOME/bin/reportgen.sh имя_трассы.utt2* из того каталога, где находится трасса (небольшое замечание: имя трассы не должно содержать пробелов). При этом будет создан каталог Reports с отчетами о тестировании, для просмотра которых необходимо использовать обычный браузер. Отметим, что возможности вывода покрытия в трассу доступны в C++TESK Testing ToolKit только начиная с версии 1.0.2.

7. Разработка сценариев тестирования на основе обхода конечных автоматов

Дополнение к главе 5.

Для генерации тестовых воздействий помимо рандомизированного генератора может использоваться обходчик конечных автоматов. Он подразумевает динамическое создание *конечного сильно-связного автомата* и его *обход* в процессе тестирования некоторым *неизбыточным алгоритмом.* Автомат создается с помощью функции получения текущего состояния и списка стимулов (в данном контексте сценарных функций), с описанием допустимости их запуска в текущем состоянии. Тест считается законченным, когда в каждом состоянии были поданы все допустимые в нем стимулы, но нигде не было обнаружено ошибок.

Для каждой сценарной функции определен *контекст итерации (см. сигнатуру сценарных функций - IntAbcCtx& ctx - это и есть контекст)*, который хранит значения *итераторов.* Итераторы — это конструкции для перебора входных параметров запускаемых операций эталонной модели таким образом, чтобы различные стимулы с различными параметрами считались разными стимулами. Это очень удобно в том случае, когда необходимо гарантированно перебрать все комбинации входных параметров всех стимулов.

Внутри сценарной функции сразу после ее начала пишется определение итерационной переменной: *int& имя_переменной = IVAR(a)*. Внутри макроса *IVAR()* должна стоять одна буква латинского алфавита, их всего 26 (от a до z), а так как для каждой переменной требуется своя отдельная буква, то всего переменных на данный момент поддерживается 26. Значения этих переменных будут храниться в вышеуказанном контексте итерации.

Напомним, что после определения всех переменных пишется пара макросов *IBEGIN-IEND.* Внутри пары IBEGIN-IEND могут идти вложенные друг в друга циклы по итерационным переменным, а внутри последнего из них должен быть вложен макрос *IACTION{}*. Цикл по итерационным переменным представляет собой обычный цикл *for*: *for(имя_переменной = 0; имя_переменной < 2; имя_переменной++)*. Использование итерационной переменной позволяет обозначить наличие целой серии стимулов, различающихся этим параметром *имя_переменной*. Отметим, что необходимость в итераторах возникает не во всех случаях, часто без них можно обойтись. В простых примерах типа FIFO необходимость в итерационных переменных отсутствует, но они могут быть добавлены искусственно, например, для увеличения размера автомата. Допустим, мы включим итерационную переменную в сценарную функцию nop, с тем чтобы увеличить количество дуг, исходящих из данного состояния.

```
bool ParentScenario::scen_nop(IntAbcCtx& ctx) {
  int &pseudo_iteration = IVAR(a);
  IBEGIN
  for(i = 0; i < 10; i++)
    IACTION {
      dut.cycle();
      YIELD(dut.verdict());
    }
  IEND
}
```

А функция получения текущего состояния может быть, например, такой:

```
std::string ParentScenario::get_state_fifo() {
  std::string state;
  std::stringstream out;
  out << dut.fullness() << ","<< dut.isIface1Ready() << dut.isIface2Ready();
  state = out.str();
  return state;
}
```

Для того чтобы запускать автоматный обход, необходимо указывать обходчик engine::fsm:

```
#include <utils/testreg.h>
#include <fifo_scen.h>

using namespace cpptesk::examples::fifo;

ChildScenario scen_fsm;

TEST_REGISTRY_BEGIN
REGISTER_TEST(scen_fsm, engine::fsm, "-uerr=1 -nt -nt2")
TEST_REGISTRY_END
```