Продолжение. Начало в № 3 `2008

Проектирование в условиях временных ограничений: верификация проектов

«Доверяй, но проверяй!»
Народная мудрость

Ростислав ГРУШВИЦКИЙ
RIGrushvitsky@mail.eltech.ru
Максим МИХАЙЛОВ
yamaksya@yandex.ru

Авторы продолжают рассмотрение вопросов верификации проектируемой аппаратуры. Данный выпуск посвящен описанию языка PSL (Property Specification Language).

В языках верификации аппаратуры (**HVL, Hardware Verification Languages**) он занимает нижний уровень иерархии. Целесообразность обсуждения нижнего уровня иерархии обусловлена тем, что средства последующих уровней либо непосредственно опираются на **PSL**, либо, изменяя синтаксис, наращивают функциональные возможности верификационных средств с сохранением и развитием основных концепций языка **PSL**. Язык **PSL** помимо своего прямого назначения (как средства перечисления характерных особенностей аппаратуры) идеально подходит для формальной спецификации аппаратных средств. В этой ситуации **PSL**-описание можно использовать в качестве исходных данных для функциональной верификации. С его помощью можно задать свойства проекта, которые выбираются для контроля при верификации. С одной стороны, спецификация на языке **PSL** легко читаема разработчиком, с другой — математически точна. Поэтому **PSL**-спецификацию иногда называют *исполняемой верификационной документацией* проекта. В большинстве случаев время, дополнительно требуемое на проверки, не очень значительно. И затраты на отладку при методологии, основанной на проверке базовых свойств аппаратуры, более чем компенсируются эффективностью ее применения.

Совсем немного об истории языка

Идея создания **PSL** принадлежит техническому комитету функциональной верификации фирмы Accellera. За основу был выбран язык Sugar, разработанный внутри IBM, где при верификации моделей для описания свойств аппаратуры использовалась формальная система CTL (Computation Tree Logic). Для нотации CTL была свойственна краткость изложения, однако она сложно воспринималась неспециалистами. Основная функция Sugar (дословно — сахар) как раз и заключалась в том, чтобы «подсластить» синтаксис CTL. Другими словами, изначально язык Sugar представлял собой дружественную к пользователю синтаксическую надстройку. Позднее в Sugar появилось расширение регулярных выражений для временной области и добавилась поддержка динамической проверки свойств при моделировании. Заключительная стадия изменений была связана с заменой семантического основания языка с CTL на LTL (Linear-time Temporal Logic, логика линейного времени), поскольку последнее оказалось более пригодным для моделирования и доступным широкой общественности. На данный момент развитием и адаптацией **PSL** занимается организация PSL/Sugar Consortium. В настоящее время в Интернете доступно справочное руководство к стандарту PSL v1.1 LRM (например, www.pslsugar.org).

Исходно в его реализацию закладывались концепции, позволяющие ему легко эволюционировать и опираться на операторы, лежащие в основе наиболее распространенных языков описания аппаратуры, способы описания булевских выражений, принятые в этих языках, и т. д. Однако там, где это было необходимо, вводили собственные семантику и синтаксис. В частности, это касается правил построения сложных временных взаимоотношений между булевскими выражениями.

Необходимость введения подобного языка диктовалась двумя основными соображениями. С одной стороны, он должен был отражать специфику задач верификации, а с другой — поддерживаться теми средствами, при помощи которых разработчик собирается контролировать все шаги своего проектного процесса.

Существенным при этом является, что **PSL** поддерживает передовые идеи методологии верификационной техники, такие как формальная верификация и функциональное покрытие. При этом наличие верификационных фрагментов в исходных материалах проекта не воспринимается компилирующими средствами и не влияет на окончательные технические характеристики создаваемого оборудования.

Язык реализован в различных вариантах в зависимости от выбора базового языка описания проекта или его свойств. В настоящий момент **PSL** находит основное применение в проектах, созданных с помощью **VHDL** или **Verilog**. Кроме того, язык органически встроен в состав **SystemVerilog**. Дальнейшее расширение **PSL** предполагает также возможность использования выражений в нейтральной форме на языке **GDL (General Description Language)**. Описанные на **PSL** свойства могут быть встроены в основной **HDL**-код как комментарии либо размещены в отдельном файле.

Полнота и точность проверки определяется исключительно опытом разработчика. Свойства кладутся в основу утверждений об их выполнимости при проведении той или иной верификационной процедуры. Именно поэтому методы верификации, базирующиеся на формальном определении свойств проекта, носят название методов верификации, основанных на утверждениях (**ABV, Assertion Based Verification**). Число утверждений, которое целесообразно добавить к определенному проекту, различно. Проект может требовать как единицы, так и сотни утверждений в зависимости от сложности проекта и сложности свойств, которые планируется проверять. Включение проектировщиком верификационных утверждений позволяет достаточно просто контролировать (руководителю проекта, тестировщику или

самому проектировщику) проверяемые свойства. Тем самым легко обеспечить разнообразные схемы взаимодействия различных участников проекта.

Упрощения написания можно достичь, ориентируясь на мониторы утверждения. Мониторы утверждений в определенной степени похожи на подпрограммы (стандартные и пользовательские) в классических языках программирования. Написание мониторов для заданного проекта с требованием получения максимальной эффективности требует тщательного анализа и планирования. Необходимо отметить, что рост числа утверждений не обязательно приводит к увеличению степени покрытия проекта. При включении проверок следует также учитывать определенное увеличение времени компиляции за счет добавления верификации.

Особенности языков верификации аппаратуры

При создании языка верификации аппаратуры разработчики должны были, прежде всего, учесть основные отличия языков описания аппаратуры от классических языков программирования. Продолжая сравнение операторов типа **assert** с программными ловушками для обнаружения дефектов разработок SW, необходимо иметь в виду, что в основе языков описания аппаратуры лежит принцип параллельного исполнения операторов. Поэтому в этих языках расстановка ловушек в последовательности расположения параллельных операторов (за исключением последовательных операторов) не соответствует задачам фиксаций событий, вызываемых выполнением предшествующих, а тем более — окружающих операторов.

Каждый оператор языка (или их неразрывная последовательность) должны отвечать ряду решаемых задач:

- содержать признак начала и конца принадлежности к определяемому языку;
- отражать целевое назначение оператора (обнаружение логического отклонения, степень полноты покрытия верификационными средствами, обнаружение отклонений от ограничений проекта и т. д.);
- определять логические условия возникновения верифицируемого свойства;
- задавать временной аспект контроля свойств (абсолютный или относительный);
- определять действия, планируемые при выполнении логических и временных заданий.

Язык необходимо создавать в форме надстройки над соответствующим базовым языком HDL. При этом он должен поддерживать синтаксические конструкции базового языка.

Всем этим требованиям отвечают концепции языка описания свойств **PSL**. Операторы и фрагменты языка **PSL** могут быть реализованы в зависимости от пожеланий разработчика либо в форме комментариев, встроенных в исходный код HDL-программы, либо оформлены в виде внешнего файла, на который ссылается исходный файл. Язык **PSL** вводит обильный набор конструкций для построения сложных конструкций из выбранных свойств проекта. Сложность таких конструкций закладывается в их поведенческую природу. Конструкции имеют не только произвольную логическую сложность, но и сложность временных привязок. Основным отличием языка от других языков описания аппаратуры является то, что его используют для описания только тех свойств проекта (и только в те моменты времени), которые выбраны для контроля при верификации состояния именно этих свойств. Полное описание языка содержится в стандарте [1]. Подробное описание приведено в приложении работы [3]. Различные аспекты языка рассматриваются в работах [4], [5], [6]. В существующей литературе примеры обычно приводятся с ориентацией на язык Verilog, в данной работе примеры даются на языке VHDL (учитывая его распространенность в России).

Property Specification Language

Следуя основной идее статьи, перейдем от обсуждения общих вопросов языков верификации аппаратуры (HVL, Hardware Verification Languages) к подробному рассмотрению языка описания свойств. Знакомство с **PSL** удобнее всего начать с анализа структуры типового выражения.

Структура PSL-предложения

В качестве примера остановимся на структуре **PSL**-предложения для языка VHDL, утверждающего, что только один из сигналов проекта, en1 или en2, равен логической '1' на восходящем фронте сигнала clk. Предложение будет иметь вид:

```
-- psl property enable is always (en1 xor en2) @ rising_edge (clk);
```

В этой структуре можно выделить следующие синтаксические элементы:

- - **psl:** комментарий и ключевое слово. **PSL**-утверждение может быть вставлено в код VHDL-программы и может занимать несколько линий подряд. Для этого они должны начинаться с символа комментария;
- **property:** определение свойства;
- enable: имя свойства;
- **is:** разделительный символ для отделения имени property;
- **always:** оператор сохранности свойства. Чаще всего, как в данном примере, используется ключевое слово **always** для истинного или условного утверждения, для всегда ложных утверждений пользуются словом **never**. Реже употребляются слова **eventually** и **next**. Первое соответствует необходимости контроля в неопределенном сейчас будущем времени, второе — в определенном будущем;
- (en1 xor en2): булевское выражение, заключенное в круглые скобки и определяющее условие, проверяемое утверждением. В качестве булевского выражения можно использовать любой вид условного оператора языка VHDL, включая операторы IF, подпрограммы, преобразование типа и т. д. **PSL** интерпретирует std_logic'1' как истинно, '0' соответственно как ложно. Выражения могут содержать объекты только типа std_logic, поэтому оператор эквивалентности (=) необходимо опускать;
- @ rising_edge (clk): временное условие, которое определяет временные рамки проверки. Если свойство тактируемое, то оно должно начинаться с тактирующего оператора @. Любое тактирующее выражение уровня регистровых передач (RTL) можно использовать с любым временным атрибутом любого VHDL-сигнала. Обычно форма утверждения упрощается, если используется не системный такт, а какой-либо стробирующий сигнал;
- завершает предложение символ точки с запятой.

Для начала проверок условий в режиме моделирования свойство enable должно быть активировано предложением вида:

```
-- psl assert enable;
```

Здесь выделяются синтаксические конструкции:

- - **psl:** знак комментария и ключевое слово **psl**;
- **assert:** тип верификационной директивы;
- enable: имя активизируемого свойства;
- завершает предложение символ точки с запятой.

Необходимо отметить, что ключевые слова в языке **PSL** зависят от регистра. Приведем еще несколько примеров выражений на языке **PSL**. Одно из простейших свойств, которое можно представить, принимает форму комбинационного булевского условия, всегда оцениваемого как истина. С помощью **PSL** такое утверждение может быть описано следующим образом:

```
assert always CONDITION;
```

Однако на практике в такой форме свойство оказывается не очень полезным, так как оно зависит от состязания логических элементов. Более распространенным вариантом при проектировании синхронных схем является введение тактирования (проверка истинности условия в строго определенные моменты времени):

```
assert (always CONDITION) @(posedge clk);
```

Также в **PSL** есть возможность задания тактового сигнала по умолчанию. Таким образом, можно избавиться от необходимости явного использования оператора @ в каждом отдельном утверждении (для Verilog):

```
default clock = (posedge clk);
assert (always CONDITION);
```

Еще одним распространенным приемом при описании свойств является представление в форме импликации, когда некоторое предусловие должно быть выполнено до того, как начнется проверка основного условия:

```
assert always PRECONDITION -> CONDITION;
```

Данное утверждение означает, что всякий раз, когда выполняется предусловие PRECONDITION, условие CONDITION должно удовлетворяться в том же цикле. В качестве условий могут выступать временные последовательности, заключенные в фигурные скобки:

```
assert always {a;b} |-> {c;d};
```

Последовательность **{a;b}** выполняется в том случае, если условие **a** удовлетворяется в текущем цикле, а условие **b** — в следующем. Символ «|->», помещенный между двумя последовательностями, обозначает суффиксную импликацию, то есть если первая последовательность выполняется, то вторая также должна выполняться, причем в том же цикле.

Кроме всего прочего, в утверждение можно добавить условие остановки (например, по сигналу reset), при активации которого прекратится процесс проверки временной последовательности:

```
assert always ({a;b} |-> {c;d} abort reset);
```

Данное выражение утверждает, что при появлении сигнала reset снимаются все обязательства по выполнению условия, независимо от того, насколько свойство соответствует действительности.

Определения и структура языка

В силу относительной новизны формальной верификации как области тестирования (особенно это касается ПЛИС) существует неразбериха, обусловленная множеством различных направлений и идей. Особенно данная ситуация отражается на терминологии. Говорить об устоявшейся номенклатуре этого раздела в отечественных изданиях, к сожалению, не приходится. Скорее можно констатировать ее отсутствие по причине слабой освещенности рассматриваемой темы в России. Но и иностранные источники не отличаются единством во мнениях и зачастую выдают противоположные по смыслу определения. А поскольку вопросы терминологии являются базовыми при любом обсуждении, то авторы сочли необходимым посвятить часть статьи пояснению некоторых фундаментальных понятий. В соответствии с названием языка ключевым в **PSL** является понятие свойства.

Свойство (property) можно определить как факт, имеющий отношение к проверяемому устройству, то есть утверждение, которое может быть непосредственно подтверждено или опровергнуто. Если же утверждение не может быть признано верным (или неверным) на базе доказательств, то оно называется предположением (assumption). На основе свойств строятся утверждения (assertions). Утверждение, по сути, является командой средству верификации доказать истинность заданного свойства. В качестве такого средства может выступать как моделирующая система (динамическая верификация), так и программа проверки модели, выстраивающая математическое доказательство свойства (статическая формальная верификация).

Свойства могут играть несколько различных ролей при верификации:

- Как мониторы, выполняющие динамическую проверку состояния проекта в процессе моделирования.
- Как ограничения, которые определяют допустимые последовательности входных воздействий при моделировании.
- Как метрики функционального покрытия, с помощью которых можно оценить степень завершенности моделирования.
- Как утверждения, требующие доказательства средствами статической формальной верификации.
- Как предположения, выдвигаемые при доказательстве утверждений.

Как уже говорилось выше, свойства в **PSL** состоят из булевских выражений, написанных на базовом языке (VHDL или Verilog), а также временных операторов и последовательностей, присущих **PSL**. Булевские выражения позволяют производить выборку состояния HDL-проекта в конкретные моменты, а временные операторы и последовательности, в свою очередь, описывают отношения между этими состояниями во времени. Возможность ясного и точного описания временных отношений — ключевое средство, которое **PSL** привносит в базовый язык.

Структурно язык **PSL** разбит на 4 четко выраженных уровня:

- булевский уровень (boolean layer);
- временной уровень (temporal layer);
- уровень верификации (verification layer);
- уровень моделирования (modeling layer).

Уровни временных отношений и верификации имеют свои собственные синтаксисы, тогда как уровни булевских выражений и моделирования заимствуют синтаксис базовых HDL-языков. Поэтому, ведя речь о языке **PSL**, обычно уточняют какой «привкус» (flavour) он имеет: VHDL, Verilog, GDL и т. д. Этот «привкус» кроме того, что явно определяет синтаксис уровней булевских выражений и моделирования, оказывает и некоторое влияние на синтаксис временного уровня.

Булевский уровень содержит выражения базового языка. Формально в качестве выражения булевского уровня может выступать любое выражение, разрешенное для применения в качестве условия в операторе IF базового языка. Далее приведен пример равноценных выражений булевского уровня:

```
(a & (a — 1)) == 0 // Verilog
(a and (a — 1)) = 0 -- VHDL
```

Временной уровень составляет основу языка **PSL**. Наряду с выражениями булевского уровня, здесь допустимо использование временных операторов и так называемых последовательных расширенных регулярных выражений (SERE, Sequential Extended Regular Expression). Обычно выборка выражений временного уровня производится под управлением некоторого тактового сигнала, так как язык **PSL** предназначен в основном для описания синхронных систем:

```
(always req -> next (ack until grant)) @clk
```

Уровень верификации содержит верифицирующие директивы, а также синтаксические конструкции для группирования операторов **PSL** и их привязки к HDL-модулям. По сути, верифицирующая директива — это команда инструменту верификации, поясняющая действия, которые необходимо выполнять со свойством. Основными верифицирующими директивами являются **assert** (требование проверить свойство), **assume** (предположение об истинности заданного свойства) и **cover** (оценка частоты выполнения заданного свойства при моделировании):

```
assert (always req -> ack) @clk;
```

Уровень моделирования предназначен для расширения возможностей **PSL** при помощи базового языка. Например, уровень моделирования может быть использован для вычисления ожидаемого значения выходного сигнала. В этот уровень включены некоторые дополнительные языковые конструкции и ряд полезных функций. Например, функция **prev**() возвращает значение выражения на предыдущем цикле. Таким образом, уровень моделирования предоставляет средства описания поведения моделей проектных входов и выходов, включая достаточно сложную логику (не являющуюся собственно частью проекта, но необходимую для фиксации состояний модели, задаваемых на достаточно высоком уровне).

Предлагаемое обычно четырехуровневое представление свойства является достаточно простым и естественным путем толкования концепции и не должно создавать впечатления сложности и громоздкости синтаксической конструкции языка.

Булевский уровень

Булевскому уровню property соответствуют булевские выражения, объединяющие переменные проектной модели. Примером на языке Verilog является описание условия нахождения в активно высоком состоянии только одного из двух сигналов проекта (en1 и en2):

```
!(en1 & en2) // enable are mutually exclusive.
```

Заметим, что данное выражение не ассоциируется с какими-либо временными соотношениями и в этом смысле считается неопределенным.

Временной уровень

Реальная мощь свойств связана с понятием временного уровня. Этот уровень определяет отношение булевских выражений к каждому моменту времени. Целям привязки служат временные операторы (temporal operators), которые позволяют совершенно точно определять, когда соответствующее выражение булевского уровня должно соответствовать предписанным значениям. Понятие временного уровня позволяет нам однозначно выразить поведение проектного свойства как серию булевских выражений, соотнесенных с множеством тактовых циклов.

Среди прочих временных операторов, пожалуй, стоит обратить внимание на следующие: **always**, **never**, **next**, **until** и **before**. Назначение этих операторов интуитивно понятно, однако существуют некоторые тонкости их использования.

Оператор **always** выполняется в том случае, если все его операнды выполняются в каждом отдельном цикле. В противоположность ему, оператор **never** выполняется тогда, когда все его операнды не выполняются в каждом отдельном цикле. Оператор **next** выполняется, если его операнды выполняются в следующем за текущим циклом. Например, утверждение:

```
assert always req -> next grant;
```

означает, что всякий раз, когда сигнал **req** принимает значение «истина», сигнал **grant** должен также принимать значение «истина» в следующем цикле. Размер цикла определяется либо заданием синхросигнала по умолчанию (default clock), либо с помощью оператора @ внутри свойства. Необходимо отметить, что данное утверждение никак не ограничивает сигнал **grant** в циклах, отличных от следующего за тем, в котором сигнал **req** имеет истинное значение. Также приведенное выражение не определяет поведение сигнала **grant**, если значение **req** — ложь.

Оператор next в качестве аргумента может принимать количество циклов:

```
assert always req -> next[2] (grant);
```

Данное утверждение означает, что всякий раз, когда сигнал **req** принимает значение «истина», сигнал **grant** должен принимать значение «истина» через два цикла. Особенность этой языковой конструкции в том, что в случае если сигнал **req** имеет значение «истина» на протяжении, например, трех последовательных циклов, то сигнал **grant** также должен быть истиной в течение трех циклов, но с задержкой в два цикла.

Назначение оператора **until** более «тонкое»:

```
assert always req -> next (ack until grant);
```

Данное утверждение означает, что если в текущем цикле сигнал **req** принимает значение «истина», то в следующем цикле значение «истина» должно быть у сигнала **ack**, и оно должно оставаться таковым до тех пор, пора в каком-нибудь из последующих циклов сигнал **grant** не станет истиной. Значение сигнала **ack** в цикле, где **grant='1'**, уже не играет роли. Если сигнал **req='1'**, а в следующем цикле истинным становится сигнал **grant**, то значение сигнала **ack** также оказывается не важным. С другой стороны, данное утверждение не обязывает сигнал **grant** когда-либо становиться истиной. В этом случае сигнал **ack** должен оставаться истинным бесконечно.

И, наконец, оператор **before**:

```
assert always req -> next (ack before grant);
```

В этом примере утверждается, что если в текущем цикле сигнал **req** принимает значение «истина», то сигнал **ack** должен иметь значение «истина», по крайней мере, один раз за период, начиная со следующего цикла и до цикла, после которого сигнал **grant** примет значение «истина». Опять же, данное утверждение не накладывает ограничений на появление сигнала **grant**. Другими словами, возможна ситуация, когда сигналы **ack** и **grant** остаются ложными бесконечно долго после появления сигнала **req**.

Верификационный уровень

Булевский и временной уровни определяют общее поведение «контролирующего оборудования», но они не детализируют, как описанное свойство необходимо использовать во время верификации. Возможны различные варианты подобного использования:

- свойство является утверждением (asserted) и должно проверяться (checked);
- свойство необходимо рассматривать как определенное ограничение;
- свойство необходимо проверять и в случае его истинности использовать далее в качестве события (event), запускающего определенные заданными функциональными границами действия.

По своей сути свойства являются просто декларациями, и их работой должны управлять верификационные средства, порядок вызова которых определяют содержимое и последовательность директив. Язык **PSL** поддерживает следующие директивы:

- **assert**;
- **assume**;
- **assume_guarantee**;
- **cover**;
- **restrict**;
- **restrict_guarantee**;
- **fairness**;
- **string_fairness**.

Наиболее часто используются во время моделирования с верификацией, основанной на утверждениях, директивы **assert** и **cover**. Другие по большей части предназначены для формальной верификации.

Директива **assert** дает команду средству верификации на проверку выполнимости свойства. Дополнительно директива assert может определять сообщение, которое будет выводиться, если свойство не выполняется:

```
assert (always !(en1 & en2)) report «message»;
```

Директива **cover** предписывает верификационному средству проверять покрытие определенного пути при моделировании на основе тестового набора, что позволяет судить о полноте тестирования. Так же, как и для директивы **assert**, допустим строковый параметр:

```
cover {start_trans;!end_trans[*];start_trans & end_trans}
report «Transactions overlapping by one cycle covered»;
```

Данный пример демонстрирует указание верификационному средству проверить, есть ли, по крайней мере, один случай перекрытия транзакций (следующая транзакция начинается в том же цикле, когда завершается предыдущая).

При помощи директив **assume** в процесс верификации вводятся ограничения. Предположения (assumptions) чаще всего используются для задания условий работы определенного свойства проекта путем ограничения правил поведения входов системы. Другими словами, свойство должно выполняться только на тех путях прохода алгоритма системы, которые удовлетворяют сделанному предположению.

Описанные на языке **PSL** свойства необходимо объединять с самим устройством (его RTL-моделью), чтобы иметь возможность проверки проектных требований. Один из

вариантов подключения к проекту предполагает встраивание верификационных директив в основной HDL-код как комментарии. В качестве альтернативы, директивы можно группировать в верификационные модули (vunit, verification unit) и размещать в отдельном файле. Фактически существует три типа верификационных модулей: наиболее общий тип **vunit**; тип **vprop**, не содержащий ничего, кроме утверждений; тип **vmode**, который содержит все, что угодно, но только не утверждения. Рассмотрим примеры.

Для Verilog:

```
vunit v1 {
   property pkt-xfer = {pkt_sop; pkt_xfer_in_progress[*1:100];
                                          pkt_abort} @posedge clk;
   al : assert pkt_xfer;
} // vunit v1
```

Для VHDL:

```
vunit v2 {
   property pkt-xfer IS {pkt_sop; pkt_xfer_in_progress[*1:100];
                                          pkt_abort} @posedge clk;
   al : assert pkt_xfer;
} // vunit v2
```

Верификационный модуль может быть явно ограничен определенным HDL-модулем. Например:

```
vunit my_properties(myVerilog.instance.name) {
   assert (always req -> ack) @ clk;
   assume (never req && reset)@ clk;
}
```

Уровень моделирования

Уровень моделирования предоставляет возможность задать поведение входов верифицируемого проекта. Чаще всего такая необходимость возникает при ориентации на такие средства формальной верификации, в которых не предусмотрены какие-либо другие способы спецификации поведения. Кроме того, средства уровня позволяют декларировать и дать описание внутренних вспомогательных сигналов и переменных.

Моделирующий уровень вводит в употребление четыре разновидности описаний — соответственно для языков SystemVerilog, Verilog, VHDL и GDL. Для каждой разновидности, в любом месте, где может появляться HDL-выражение, моделирующий слой может расширяться, предоставляя возможность вставить любую форму HDL- или **PSL**-выражения, большинство из которых включено в описание булевского уровня. Таким образом, HDL-выражения, **PSL**-выражения, встроенные функции, точки завершения (endpoints) и выражения-модули (union expression) можно использовать в качестве выражения внутри моделирующего слоя.

Каждая разновидность описаний моделирующего слоя вводит и поддерживает свои трактовки конструкций, соответствующих языковому описанию аппаратуры. Для языка VHDL поддерживаются следующие встроенные функции:

- prev (<любой тип>[,<число>]); — возвращает значение выражения на предыдущем цикле (если есть число, то значение, предшествующее на это число циклов);
- next (<любой тип>); — возвращает значение выражения на следующем цикле;
- stable (<любой тип>); — возвращает значение «истина», если значение выражения совпадает для текущего и предыдущего цикла;
- rose (<бит>); — возвращает значение «истина», если аргумент битового типа принимает значение 1 на текущем цикле и 0 на предыдущем (передний фронт);
- fell (<бит>); — возвращает значение «истина», если аргумент битового типа принимает значение 0 на текущем цикле и 1 на предыдущем (задний фронт);
- isunknown (<битовый вектор>); — возвращает значение «истина», если хотя бы один из битов вектора имеет неопределенное значение (не 0 и не 1);
- countones (<битовый вектор>); — возвращает число битов, имеющих значение 1;
- onehot (<битовый вектор>); — возвращает значение «истина», если только одна 1 в битовом векторе;
- onehot0 (<битовый вектор>); — возвращает значение «истина», если не более одной 1 в векторе.

Последовательности (Sequences)

Завершив необходимое краткое рассмотрение всех уровней, можно вернуться к ключевом средству, которое язык **PSL** привносит в базовый язык — к возможности описания сложных временных отношений между логическими выражениями. Большинство стандартных алгоритмических языков вводит целый ряд понятий при краткости формальных правил их записи, существенно расширяя при этом функциональные возможности описаний. Так и язык **PSL** помимо обычных (для языков VHDL или Verilog) способов описания логики поведения аппаратуры вводит новые синтаксические конструкции.

Одним из основных требований к языку утверждений является возможность лаконично отобразить поведение определенных свойств проекта на множестве тактов. Чтобы ответить этим требованиям, в язык **PSL** введено понятие *последовательностных выражений* SERE. Эти выражения дают разработчикам простой и эффективный инструмент для описания последовательного во времени поведения проекта и его фрагментов. По своей сути последовательности в **PSL** являются синтаксическими эквивалентами временным диаграммам. При этом выполнение указания о движении времени вперед соответствует чтению текста последовательности слева направо. Для обозначения перемещения вперед на один цикл внутри последовательности выбран оператор «точка с запятой». Фигурные скобки отмечают начало и конец последовательности.

Поэтому фрагмент кода:

```
{sig_a; sig_b}
```

должен трактоваться следующим образом:

- как только случится, что в текущем цикле «**sig_a**» имеет значение «истина»;
- дождаться следующего цикла;
- проверить, что «**sig_b**» является истиной.

Выражения SERE позволяют записывать длинные последовательности и группы связанных последовательностей в компактном виде. Например:

- выражение {a[*2]} есть то же самое, что и {a;a};
- {a[+]} эквивалентно записи {a;a;…;a}, где a повторяется один или более раз;
- {a[*]} эквивалентно записи {a;a;…;a}, где a повторяется ноль или более раз;
- выражение {[*]} обозначает любую последовательность;
- {a[=2]} раскрывается как {[*];a;[*];a;[*]};
- {a[*1 to 3]} обозначает либо последовательность {a}, либо {a;a}, либо {a;a;a}.

Также существуют операторы, которые работают с последовательностями в целом:

- запись {seq1} | {seq2} означает, что одна или другая последовательность должна выполняться;
- запись {seq1} & {seq2} означает, что обе последовательности должны выполняться;
- запись {seq1} && {seq2} означает, что обе последовательности должны выполняться и должны иметь одинаковую длину.

В реальных условиях между двумя отмеченными событиями («sig_a» и «sig_b») может проходить более чем один цикл. Количество циклов может лежать в определенном диапазоне. Также допустимо задание различных условий (например, события не должны происходить в смежных тактах и т. д.). Для удовлетворения подобных требований в языке **PSL** существуют операторы повторения:

- упорядоченный (consecutive) оператор позволяет строить повторяющуюся последовательную цепочку событий, где событие — это либо булевское выражение, либо последовательность (обозначается [*]);
- неупорядоченный (non-consecutive) оператор позволяет строить повторяющуюся (возможно, непоследовательную) цепочку из булевского выражения (обозначается [=]);
- оператор goto строит цепочку из булевского выражения таким образом, что это выражение должно быть истинным в последнем цикле цепочки — переход к последнему повтору в цепочке (обозначается ([->])).

Рассмотрим несколько примеров применения операторов повторения. Пояснение будет строиться на основе следующего протокола записи:

- появление сигнала **wr_started** означает начало рассматриваемой последовательности;

- в течение нескольких (специфично для каждого примера) циклов контролируется сигнал занятости **wr_channel_busy**;
- по истечению заданного числа циклов ожидается появление сигнала **wr_done**.

Для упорядоченных операторов повторения возможны такие вариации:

- {wr_started; wr_channel_busy[*2]; wr_done} — сигнал должен быть активным в течение двух циклов;
- {wr_started; wr_channel_busy[*0:100]; wr_done} — сигнал должен быть активным в диапазоне от 0 до 100 циклов;
- {wr_started; wr_channel_busy[*2: inf]; wr_done} — сигнал должен быть активным как минимум 2 цикла;
- {wr_started; wr_channel_busy[+]; wr_done} — сигнал должен быть активным один или более циклов.

Для неупорядоченных операторов повторения:

- {wr_started; wr_channel_busy[=2]; wr_done} — сигнал должен появиться 2 раза (не обязательно в последовательных циклах);
- {wr_started; wr_channel_busy[=0:100]; wr_done} — сигнал должен появиться 100 раз или меньше на протяжении всей цепочки (также нет ограничения на смежность циклов);
- {wr_started; wr_channel_busy[=2: inf]; wr_done} — сигнал должен появиться как минимум 2 раза (при этом справедливо замечание, свойственное всем неупорядоченным операторам).

Для операторов goto:

- {wr_started; wr_channel_busy[->2]; wr_done} — цепочка повторов должна завершаться на цикле, в котором сигнал занятости становится активным второй раз;
- {wr_started; wr_channel_busy[->1:100]; wr_done} — количество повторов должно быть от 1 до 100, последний повтор должен совпадать с окончанием цепочки.

Заключение

Естественно, в рамках журнальной статьи было невозможно дать полное описание языка. Поэтому за границами рассмотрения остались многие тонкости. Для дальнейшего изучения будут полезны информационные источники, указанные в разделе «Литература». Кроме того, авторы со своей стороны всегда готовы по мере сил ответить на любые вопросы, касающиеся данной тематики.

Очевидно, наилучший способ овладения — это постоянное приведение примеров использования описываемых синтаксических конструкций. Более доступным (в сравнении с **PSL**) для практических применений являются средства мониторов утверждений. Мониторы базируются на конструкциях **PSL**, поэтому изложение последующих примеров оказалось бы затруднительным без хотя бы краткого знакомства с языком. В следующей статье цикла авторы планируют остановиться на мониторах и тогда уже в основу рассмотрения положить типовые ситуации, возникающие при разработке реальных проектов.

Литература

1. Property Specification Language. Reference Manual v.1.1.
2. Материалы фирмы Doulos. http://www.doulos.com/knowhow/psl/
3. Foster H., Krolnic A., Lacey D. Assertion-Based Design, 2nd ed. Boston: Kluwer Academic Publishers, 2003.
4. Advanced Formal Verification Edited by R. Drechsler. Kluwer Academic Publishers, 2004
5. Долинский М. Assertion Based Verification — верификация, основанная на утверждениях // Компоненты и технологии. 2004. № 9.
6. Максфилд К. Проектирование на ПЛИС. Архитектура, средства и методы. Курс молодого бойца. М.: Додека-XXI, 2007.
7. Cohen B., Venkataramanan S., Kumari A. Using PSL/Sugar for Formal and Dynamic Verification. 2nd ed. VHDLCOHEN Publishing, 2004.