Основы методологии квазислучайного тестирования

В методологии используются следующие принципы.

 случайные воздействия с ограничениями

 Функциональные покрытия

 Построение слоистых testbench с использованием транзакторов

Общие testbench для всех тестов

 специфический тест-код для хранится отдельно от testbench

Random constraned simulation

1,8 Случайные стимулы с ограничениями-

Хотя вы хотите принудить симулятор создавать стимулы, вы не хотите чтобы они принимали

совершенно случайные значения. Если использовать SystemVerilog то задается Формат стимула ("адрес 32-бит, код операции X, Y, Z -байт, длина пересылки <32 байт "), а симулятор берет только значения, которые удовлетворяют ограничениям Эти значения передаются в модель устройства , а также в высоко уровневую модель, которая предсказывает

какой должен быть результат. Фактический выход устройства сравнивается с предсказанным.

Рисунок 1-4 показывает покрытие RCT- по сравнению с общей областью устройства. Во-первых, обратите внимание, что случайный тест часто охватывает более широкое пространство, чем направленный. Это дополнительное покрытие может перекрывать другие тесты, или может исследовать новые области, которые вы не ожидали. Если в этих новых областях обнаружили ошибку,- это удача! Если новая область является недопустимой, вам нужно написать несколько дополнительных ограничений и исключить ее. Наконец, вы, возможно, придется написать несколько направленных тестов, чтобы найти случай, не

покрытые каким-либо из испытаний с RCT

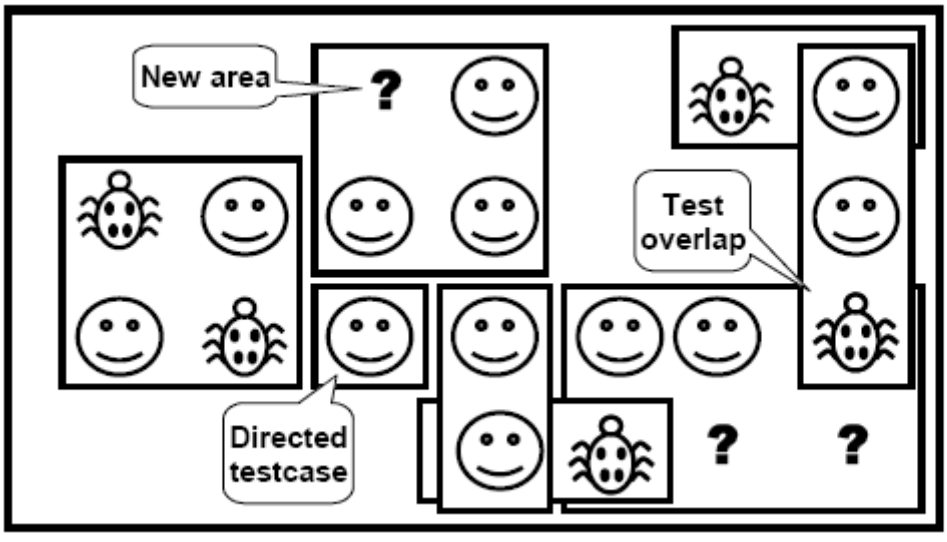


Рисунок 1-4 покрытия случайного теста с ограничениями-

Figure 1-3 Constrained-random test progress

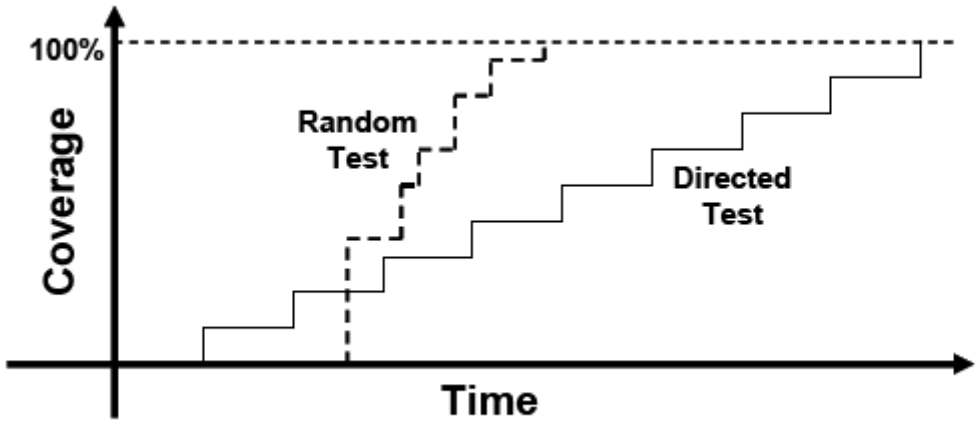
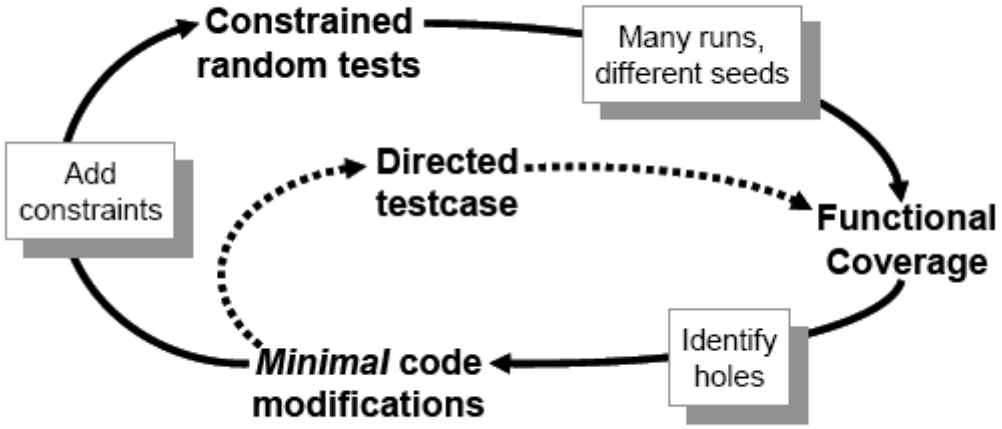


Рисунок показывает пути к достижению полного охвата.
Начнем с верхнего левого угла и используем начальный RCT . Запускайте его с различными ядрами. Просматривая отчеты о функциональном покрытии следует найти дыры, где Существуют пробелы. Теперь делаем минимальные изменениями кода, возможно, с новыми ограничениями, вносимыми ошибками или задержками DUT . Проводите большую часть работы в этом внешний цикле, направленные тесты пишите только для нескольких функций, которые вряд ли будут достигнуты путем случайных испытаний.

Figure 1-5 Coverage convergence



Когда вы думаете о случайном воздействии на тестируемое устройство, первое, что надо определить зто способ представления поля данных. Это проще всего сделать используя системную функцию $ random. Проблема в том, что такой подход имеет очень низкую окупаемость с точки зрения найденных ошибок: вы только найти ошибки потока данных, возможно ошибки битового уровня.

Следует учитывать следующие моменты

- Конфигурация устройства
- конфигурация окружающей среды
- ввод данных
- инкапсуляция данных
- исключения протокола
- Задержки
- Переходные Состояния
- Ошибки и возмущения

6.2.1 Конфигурация устройства
Что является наиболее распространенной причиной пропуска ошибок во время испытаний

устройства на RTL уровне? Недостаточное количество различных конфигураций было опробовано! Большинство тестов просто используют сброс устройства , или применяют фиксированный набор инициализируюших векторов для перевода его в известное состояние. Это подобно тестированию операционной системы ПК сразу после ее установки, и без каких-либо приложений (прикладных задач) Конечно исполнение прекрасно, и нет аварий.
Между тем, в реальных условиях, конфигурации DUT становятся все более и более случайными. Например, пользователь Synopsys использует мультиплексор - переключатель имеющий 600 входных каналов и 12 выходных каналов с разделением по времени,. Когда устройство было установлено в системе клиента, оказалось что каналы выделялись и освобождались снова и снова, так что в любой момент времени не было бы почти никакой связи между соседними каналами. Другими словами, поведение выглядит случайным. Для проверки этого устройства, инженер-тестировщик должен был написать несколько десятков строк Tcl кода для настройки каждого канала. В результате, он так и не смог бы
просмотреть конфигурации с включением более чем некоторое ограниченное число каналов. Используя CRT Методологию, оказывается достаточно написать testbench с рандомизированными параметрами для одного канала, а затем поместить это в цикл для настройки всего устройства. Тогда есть надежда, что испытания будут выявить ошибки, которые ранее были бы упущены.

6.2.2 конфигурации Окружающей среды
Устройство, которое вы разрабатываете работает в среде, содержащей
другие устройства. Когда вы проверяете тестируемое устройства, нужен testbench, который имитирует среду. Необходимо рандомизировать всю среды, в том числе количество объектов, и их настройки. Пусть создается чип переключателя ввода / вывода, который связывает несколько PCI шин к внутренней шине памяти. В начале моделирования пользователь использует рандомизации для выбора количество шин PCI (1-4), числа устройств на каждой шине (1-8), и параметры для каждого устройства (мастер или раб, контрольную сумму адреса и т.д.). И Хотя возможно много комбинаций разработчики могут быть уверены, что все было покрыто.

6.2.3 Ввод первичных данных
Это , вероятно, то о чем думают прежде всего, приступая к квазислучайному
тестированию: возьмите к примеру транзакции на шине или ATM ячейке и заполните их некоторыми случайными значениями. Насколько это трудно ? На самом деле это довольно просто если вы тщательно подготовили свои классы транзакций. Вы должны
предусмотреть возможное использование многослойных протоколов и **error injection .**

6.2.4 И**нкапсуляция** входных данных
Многие устройства обрабатывают нескольких слоев воздействий . Например, устройство может создать TCP-трафик, который затем кодируется в IP-протоколе, а затем посылается внутри Ethernet-пакетов. Каждый уровень имеет свои собственные поля управления, которое можно рандомизировать, чтобы попробовать новые комбинации. Таким образом, вы должны рандомизировать и данные и **слои,** которые из окружают. Нужно написать ограничения, которые задают существенные
поля управления, но которые также допускают **инъекционные ошибки**.

6.2.5 Исключения Протокола, ошибки и возмущения
Все, что может пойти не так, в конечном счете пойдет не так,. Наиболее сложной частью разработки и верификаци, является выбор способа обработки ошибки . следует предусмотреть все ситуации , в которых процесс может пойти неправильно, моделировать их и убедиться, что устройство обрабатывает их корректно, без блокировки или перехода в запрещенное состояние. Хороший инженер – тестировщик подводит устройство к границе функциональной спецификации, а иногда даже за нее.
Что произойдет когда два устройства взаимодействуют, если передача прерывается до полного завершения? Может ли ваш testbench имитировать эти разрывы? Если Есть поля обнаружение и исправление ошибки, вы должны убедиться, что все комбинации проверены. Случайной составляющей этих ошибок является то, что ваш testbench должен иметь возможность посылать функционально правильные стимулы, а затем, при изменении конфигурационного бита, начать выдавать случайные типы ошибок через случайные промежутки времени.

6.2.6 Задержки

Во многих случаях паспортные данные на компоненты задают диапазон задержек. Стоит ли в таком случае применять RCT? На логическом уровне это маловероятно. К сожалению, законы распределения времени задержек в большинстве случаев неизвестны. Кроме того, такой метод оказывается очень трудоемким – надо исследовать большое число компонентов с разными сочетаниями задержек. RCT может дать существенные отличия лишь при моделировании устройств с относительно длинными последовательными цепочками комбинационных схем, а такие структуры в принципе нежелательно применять при проектировании высокопроизводительных устройств. По возможности следует перестраивать подобные устройства, приводя их к параллельной или конвейерной форме. Поэтому при моделировании задержек на логическом уровне лучше применять метод моделирования с нарастающей неопределенностью, описанный выше.

Тем не менее, на структурном и системном уровне некоторые временные соотношения процессов обработки можно и нужно проверять с использованием RCT. Например, ответ ready может появиться на шине задержкой от 0 до большого числа тактов, причем при правильной организации работы любая задержка допустима. Трактовка такой ситуации как неопределенной некорректна – фактически весь цикл моделируется как неопределенность и ложно признается ошибкой. Здесь лучше использовать случайные задержки из их допустимого диапазона во время каждого теста, чтобы попытаться обнаружить, что существует комбинация, которая порождает ошибку.

И последнее замечание: вы никогда не сможете доказать, остались ли еще ошибки, так что нужно постоянно придумывать новые тактики проверки

Многие протоколы связи указывают диапазоны задержек. Например ready может появиться с задержкой от 0 до большого числа тактов . Многие прямые тесты, оптимизированные для
быстрого моделирования, и используют наименьшие задержки, за исключением, единственного теста, который пытается исследовать различные задержки. Ваши testbench всегда должны использовать случайные задержки из их допустимого диапазона во время каждого теста, чтобы попытаться обнаружить, что существует комбинация, которая порождает ошибку.
некоторые проекты являются чувствительными к джиттеру тактового генератора . Слегка сдвигая фронты генератора в ту или иную сторону, вы можете убедиться, что ваш проект не слишком чувствителен к небольшим изменениям в тактировании.
Тактовый генератор следует размещать в модуль вне testbench так, чтобы создавались события, в активной области наряду с другими событиями в проекте. Тем не менее, генератор должен иметь параметры, такие как частота и смещение, которое могут быть установлены testbench во время фазы настройки.
(Заметим, что вы ищете функциональные ошибки, а не ошибки синхронизации. Они

лучше проверяются с помощью инструментов временного анализа. Но Ваши testbench не должны нарушать требований установки и фиксации).

6,3 Рандомизация в SystemVerilog

Генерации случайных стимулом в SystemVerilog наиболее эффективна , когда использованы принципы ООП. Вы сначала создать класс для хранения группы родственных случайных переменных, а затем генератор случайных чисел для заполнения их случайными значениями. У Вас могут возникнуть трудности для ограничения допустимых значений случайных данных, или для проверки конкретных особенностей.

Обратите внимание, что вы можете делать случайными отдельные переменные, но этот случай является наименее интересным. Случайные стимулы с ограничениями создаются на уровне транзакций, и не дя одной переменной в каждый момент времени .

6.3.1 Простой класс с случайными величинами

Пример 6-1 показывает класс с случайными величинами, ограничения, и testbench код для использования этого класса.

Example 6-1 Simple random class

```
……..
class Packet;
// The random variables
rand bit [31:0] src, dst, data[8];
randc bit [7:0] kind;
// Limit the values for src
constraint c {src > 10;
src < 15;}
endclass
Packet p;
initial begin
p = new; // Create a packet
assert (p.randomize());
transmit(p);
……..
```

В Этом классе определены четыре случайных величины. Первые три использованы модификатор случайных данных так что каждый раз, когда вы рандомизируете (вызываете randomize) класс, переменным присваиваются новые случайные значения. Подобно бросанию кости: каждый бросок может дать новое значение или повторить текущее. Тип переменной randc, означает, что случайные генерируются «циклически», то есть симулятор не повторяет случайные значения, пока каждое из возможных значений не назначено. Представьте вытягивание карты из колоды: вы вытягиваете каждую карту в случайном порядке, потом тасуете колоду, и раздаете карты в другом порядке.

Обратите внимание, что выражения ограничений сгруппированы в фигурных скобках: {}. Это происходит потому, что этот код является декларативным, не процедурным, который использует BEGIN ... END.

Функция randomize() возвращает 0, если ограничения не удовлетворены.

Процедурные assertions используются для проверки результатов Вам нужно найти инструмент выяснения конкретных причин переключения на assertion и прекратить симуляцию.

Даже если вы не намерены проверять генератор, оператор ASSERT необходим в обязательном порядке

Далее для проверки результатов случайных тестов используются assert, но вы можете проверить результат, специальной процедурой, которая печатает какую-либо полезную информацию, а затем выключает, моделирование.

Ограничение в примере 6-1 задано выражением, которое ограничивает значения для переменной SRC. В этом случае, SystemVerilog выбирает между значениями 11, 12, 13, или 14.

Все переменные в классах должны быть случайными и общедоступными (public). Это дает тесту максимальный контроль над стимулом и сигналами управления тестируемого устройства.

6.3.2 Проверка результатов рандомизации

функция randomize назначает случайные значения любой переменной в классе помеченной как rand или randc, а также гарантирует, что все активные ограничения не нарушены и дает ошибку, если ваш код содержит противоречивые ограничения (см. следующий раздел), так что вы должны всегда проверять исполнение Randomize. Если контроль не предусмотрен, переменные

могут получить неожиданные значения, что приводит к провалу симуляции

Программа проверяет значение случайной величины помощью процедурных assertions. assertion проверяет randomize возвращает 1 или 0. результат и выводит ошибку, если получен 0. Вы должны установить опцию моделировщика на прекращении при обнаружении ошибки. Однако, перед окончанием моделирования вы можете вызвать специальную подпрограмму прекращающую работу , после выполнения некоторых дополнительных действий, таких как печать резюме.

6.3.3 Анализатор Ограничений

Процесс решения выражения ограничения обрабатываются в системе Verilog анализатором ограничений, который выбирает значения, удовлетворяющие ограничениям. Значения берутся из генератора псевдослучайных чисел System Verilog, начиная с ядра. Если задать SystemVerilog симуляцию и testbench от того же ядра, всегда получим тот же результат. Генератор у каждого поставщика моделировщика и для различных ограничений специфичен потому тест не может дать тот же результат при работе на различных системах, или даже в разных версиях одного и того же инструмента. Стандартный SystemVerilog указывает смысл выражений для ограничений и допустимый набор значений которые создаются, но не определяет точный порядок, в котором анализатор должен работать.

Параллельное случайное тестирование

Как вы должны запускать тесты? Направленный тест производит уникальный набор векторов стимулов и реакций. Чтобы изменить стимулы Вам нужно изменить тест.

Случайный тест состоит из testbench-кода плюс случайное зерно. Если вы используете тот же тест 50 раз, каждый с уникальным ядром, вы получите 50 различных наборов стимулов. Запуск с несколькими ядрами расширяет покрытие вашего тестирования и делает вашу работу более эффективной.

Вы должны выбрать уникальное ядро для каждой симуляции. Иногда используют время дня, но это все еще может породить дубликаты. Что делать, если вы используете систему упорядочения очередей в процессорном кластере и заказываете , начало 10 работ в полночь? Всякий раз, когда вы имеете дело со случайными величинами, вы должны понимать вероятностостный характер результата. SystemVerilog не гарантирует точное решение, но вы можете влиять на распределение.

Каждый раз, когда вы работаете со случайными числами, вы должны просмотреть тысячи или миллионы значений для усреднения шума. Изменение версии инструмента или случайные ядра могут привести к разным результатам.

Несколько работ может начаться в то же время, но на разных компьютерах, и таким образом получить то же случайное ядро, и запустить тот же стимулятор. Вы должны внедрить имя процессора в ядро. Если ваш процессор включает в себя кластер многопроцессорных машин, вы могли бы две работы начать в полночь с теми же ядрами , так что вы также должны добавить в идентификатор процесса. Теперь все работы получат уникальные ядра.

Вы должны планировать, как организовать ваши файлы так чтобы поддерживать работ с несколькими моделями. Каждое задание создает множество выходных файлов, таких как файлы отчета и функционального покрытия. Вы можете запустить каждую работу в своем каталоге, или вы можете попробовать дать уникальное имя для каждого файла.

6,4 Подробнее об ограничениях

Полезные стимулы это больше, чем просто случайные значения – это отношения между переменными. В противном случае, создание осмысленных стимулирующих значении может занять слишком много времени, или стимул может содержать недопустимые значения. Для определения этих взаимоотношений в SystemVerilog используют блоки ограничивающие, которые содержат одно или более выражение ограничения. SystemVerilog решает эти выражения одновременно, и выбирает случайные числа, которые удовлетворяют всем выражениям.

По крайней мере, одна переменная в каждом выражении должна быть случайной - либо rand или randc. Класс в следующем примере не допускает рандомизации.

Решение заключается в добавлении модификатора rand или randc переменной son.

Пример 6-2 ограничения, без случайных величин

```
class bad;
bit [31:0] son; // Error – should be rand or randc
```

```
constraint c_teenager {son > 12;
son < 20;}
endclass
```

Функция randomize пытается присвоить новые значения случайным величинам
и убедиться, что все ограничения выполнены. В примере 6-2, поскольку
нет случайных величин, randomize просто проверяет значение son находится ли оно в
границах указанного ограничением c_teenager. Если переменная не попадает в пределах 13:19, randomize
дает сбой.

рассмотрим случай двух переменных без каких-либо ограничений.

Example 6-17 Class `Unconstrained`

```
class Unconstrained;
rand bit x; // 0 or 1
rand bit [1:0] y; // 0, 1, 2, or 3
endclass
```

Есть восемь возможных решений. Поскольку нет никаких ограничений, каждый
имеет такую же вероятностью. При таком подходе приходится запустить тысячи циклов рандомизации,
чтобы увидеть Фактические результаты

6.4.1 Введение Ограничений

пример случайного класса с ограничениями.

Example 6-3 Constrained-random class

```
class Stim;
const bit [31:0] SRC_CONGEST_ADDR = 42;
typedef enum {READ, WRITE, CONTROL} stim_t;
randc stim_t type; // Enumerated var
rand bit [31:0] len, src, dst;
bit congestion_test;
constraint c_stim {
len < 1000;
len > 0;
src inside {0, [2:10], [100:107]};
if (congestion_test) {
dst inside {[CONGEST_ADDR-100:CONGEST_ADDR+100]};
}
}
endclass
```

6.4.6 Взвешеные распределения

SV позволяет создавать **Взвешеные распределения** так что некоторые значения выбираются чаще других .

ОПЕРАТОР **`dist`** содержит список переменных и весов разделенных операторами := или :/
Величины и веса Могут быть константами или переменными. Величины могут задаваться одним значением или диапазоном, например **`[lo:hi]`**. Вес задается числом (не процентом!!)

оператор **`:=`** определяет что вес одинаков для любого значения в диапазоне ,а :/ определяет, что вес должен быть поровну поделен между всеми значениями.

Example 6-10 Взвешенное случайное распределение

```
rand int src, dst;
constraint c_dist {
src dist {0:=40, [1:3]:=60};
// src = 0, weight = 40/220
// src = 1, weight = 60/220
// src = 2, weight = 60/220
// src = 3, weight = 60/220
dst dist {0:/40, [1:3]:/60};
// dst = 0, weight = 40/100
// dst = 1, weight = 20/100
// dst = 2, weight = 20/100
// dst = 3, weight = 20/100
```

```
}
```
В Примере 6-10, SRC получает значение 0, 1, 2 или 3. Вес 0+40,
а 1, 2 и 3 имеют вес 60, в общей сложности 220. Вероятность
выбора 0 это 40/220, а вероятность выбора 1, 2 или 3 -60/220 для каждого.
DST получает значение 0, 1, 2 или 3. Вес 0 равен 40, а 1, 2 и
3 доли одинаковым значением 60, в общей сложности 100. Вероятность выбора 0,
40/100, в то время как вероятность выбора 1, 2 или 3 только по 20/100 .
Напомним, величины и веса могут быть постоянными или переменными. Вы использование переменных весов позволяет изменить распределение на лету или даже исключить выбор какого-то варианта, установив вес нулевым
.
Пример 6-11 Динамическое изменение распределения веса
/ / операция на шине, байт, слово или длинное слово
```
// Bus operation, byte, word, or longword
class BusOp;
// Operand length
typedef enum {BYTE, WORD, LWRD } length_t;
rand length_t len;
// Random weights for dist constraint
int w_byte=1, w_word=3, w_lwrd=5;//можно менять вне класса так
//как переменные public
constraint c_len {
len dist {BYTE := w_byte, // Choose a random
WORD := w_word, // length using
LWRD := w_lwrd}; // variable weights
}
endclass
```
```
  ..w_byte=7
   assert ( len.randomize());
```
В Примере 6-11, длина перечисленных переменных имеет три значения.
по умолчанию наиболее часто выбирается длинное слово , потому что w_lword имеет наибольшее значение.

6.4.7 Двунаправленные Ограничения
Заметим,что блоки ограничения это не процедурный
код, выполняемый сверху донизу. Они носят декларативный код, все активны
постоянно. Если ограничить переменную внутри оператора набором
[10:50] и добавить другое выражение, ограничивающее переменную
«больше, чем 20», SystemVerilog будет выбирать значения между 21 и 50.
в SystemVerilog ограничения являются двунаправленными,

Пример 6-12 двунаправленного ограничения

```
rand logic [15:0] b, c, d;
constraint c_bidir {
b < d;
c == b;
d < 30;
c > 25;
}
```
SystemVerilog моделировщик учитывает все четыре ограничения одновременно. b
должно быть меньше, чем d, которая должна быть не менее 30. Но b должно быть
равными c, что больше, чем 25. Хотя нет прямых ограничений
на наименьшее меньшем значение D, ограничение для c ограничивает выбор.

6.4.8 Условные ограничения
Как правило, все выражения ограничения принимают активное участие в блоке. Что, если
хотят иметь ограничения активными только какое-то время? Например, шина
поддерживает передачи байт, слово и длинное слово на чтение , но можно писать только длинное слово.
SYSTEM-Verilog поддерживает две версии операторов, -> и if-else.
Когда вы выбираете из списка выражений перечислимого типа, форма оператора ->, позволяет создавать case-подобные блоки\

Скобки вокруг выражения не являются обязательными, но делают код более удобным для чтения.

Пример 6-13 блок Ограничений с оператором импликации

```
class BusOp;
...
constraint c_io {
(io_space_mode) ->
  addr[31] == 1'b1;
}
```

Если у вас есть выражение истинно-ложно, if-else оператор может быть лучше.

Пример 6-14 Ограничение с IF-ELSE оператором

```
class BusOp;
...
constraint c_len_rw {
if (op == READ)
len inside {[BYTE:LWRD]};
else
len == LWRD;
}
```

В блоках ограничения, следует использовать фигурные скобки {}, чтобы сгруппировать несколько выражений.

6.4.9 Выбор «правильного» арифметического оператора для повышения эффективности

Простые арифметические операции, такие как сложение и вычитание, экстракция бит, и сдвиги обрабатываются очень эффективно вычислителем ограничений. умножение, деление и модулю с 32-битными значениями очень дороги.

Напомним , что любая постоянная, без явного указания длины кода, рассматривается как32-битная.

Для генерации случайных адресов, которые находятся вблизи границы страницы, длиной 4096 байт, можно написать следующий код, но вычислитель подходящего значения для адреса может занять много времени,.

\

Пример 6-15 «дорогое» ограничение с некалиброванной переменной

```
rand bit [31:0] addr;
constraint slow_near_page_boundary {
addr % 4096 inside {[0:20], [4075:4095]};
}
```

Многие константы в аппаратной являются степенью 2, так что лучше воспользоваться этим путем с помощью удаления битов, а не деление и по модулю. Кроме того, умножение на степень двойки можно заменить сдвигом.

Пример 6-16 Эффективное ограничение с экстракцией бит

```
rand bit [31:0] addr;
constraint near_page_boundry {
addr[11:0] inside {[0:20], [4075:4095]};
}
```

6.5.2 Импликации

В Примере 6-18, значение y зависит от значения x. На это указывает оператор ипликации в ограничении. Этот пример, как и остальные в этом разделе также ведут себя подобно IF-импликации

Example 6-18 Class with implication

```
class Imp1;
rand bit x; // 0 or 1
rand bit [1:0] y; // 0, 1, 2, or 3
constraint c_xy {
(x==0) -> y==0;
}
endclass
```

В таблице приведены все возможные решения и их вероятности. Вы видите, что решатель показывает восемь комбинаций x и y, но все решения для, где x == 0 (A-D) были объединены вместе.

Таблица 6-2. Решения для класса **imp**

Solution	*x*	*y*	*Probability*
A	0	0	1/2
B	0	1	0
C	0	2	0
D	0	3	0
E	1	0	1/8
F	1	1	1/8
G	1	2	1/8
H	1	3	1/8

6.5.3 Импликация и двунаправленные ограничения

Предположим, что оператор импликации утверждает что при `x==0,y принуждается к 0`
Но когда **`y==0`**, ограничения на **`x отсутствуют`**. Однако импликации двунаправленные
А потому если **`y`** устанавливается в ненулевое значения , то **`x`** должно принять значение 1.
В Example 6-19 установлено ограничение **`y>0`**, значит **`x`** не может стать 0.

Example 6-19

```
class Imp2;
rand bit x; // 0 or 1
rand bit [1:0] y; // 0, 1, 2, or 3
constraint c_xy {
y > 0;
(x==0) -> y==0;
}
endclass
```

Table 6-3. Solutions for **`Imp1`** class

Solution	*x*	*y*	*Probability*
A	0	0	0
B	0	1	0
C	0	2	0
D	0	3	0
E	1	0	0
F	1	1	1/3
G	1	2	1/3
H	1	3	1/3

6.5.4 управление вероятностью используя конструкцию `solve...before`

Можно управлять SystemVerilog моделировщиком, используя ограничение "**`solve...before`**"
Как показано в следующем примере Example 6-20.

Example 6-20 Class with implication and `solve...before`

```
class SolveBefore;
rand bit x; // 0 or 1
rand bit [1:0] y; // 0, 1, 2, or 3
constraint c_xy {
(x==0) -> y==0;
solve x before y;
}
endclass
```

`solve...before` не ограничивают область решений а тольковероятности решений.
Моделировщик выбирает значение **`x из множества`** (0, 1).

Из 1000 вызовов **randomize**, **x** = 0 около 500 раз, а 1 около 500 раз
Если **x** = 0, **y** должно быть 0. Если **x** i- 1, **y** может быть 0, 1, 2, или 3 равновероятно
Table 6-4. Solutions for **Imp2** class

Solution	*x*	*y*	*Probability*
A	0	0	1/2
B	0	1	0
C	0	2	0
D	0	3	0
E	1	0	1/8
F	1	1	1/8
G	1	2	1/8
H	1	3	1/8

Но если записать **solve y before x**, распределние будет таким

Solution	*x*	*y*	*Probability*
A	0	0	1/8
B	0	1	0
C	0	2	0
D	0	3	0
E	1	0	1/8
F	1	1	1/4
G	1	2	1/4
H	1	3	1/4

6,6 Контроль при нескольких блоках ограничения

Класс может содержать несколько блоков ограничения,. например класс можно естественно разделить на две группы переменных, таких как данные для сравнения и сигналы управления, так что вы можете ограничить их отдельно. Или вы можете захотеть иметь отдельные ограничения для каждого теста. Возможно, одним ограничением будет ограничение длины данных для создания малых транзакций , а другой для длинных транзакций
Для активизации и отключения ограничения во время выполнения, можно использовать подпрограмму **constraint_mode ()** . ‘0’- отключить ‘1’-разрешить При использовании опции <имя указателя ограничения >.constraint, этот метод контролирует одно ограничение. При использовании только имени объекта , он контролирует все ограничения.
Пример 6-21 Использование constraint_mode

Example 6-21 Using `constraint_mode`

```
class Packet;
rand int length;
constraint c_short {length inside {[1:32]}; }
constraint c_long {length inside {[1000:1023]}; }
endclass
Packet p;
initial begin
p = new;
// Create a long packet by disabling short constraint
p.c_short.constraint_mode(0);
assert (p.randomize());
transmit(p);
// Create a short packet by disabling all constraints
// then enabling only the short constraint
```

```
p.constraint_mode(0);
p.c_short.constraint_mode(1);
assert (p.randomize());
transmit(p);
end
```

Множество допустимых значений случайных данных может задаваться массивом. Функция **randomize** в примере 4.41 случайным образом циклически выбирает элементы из динамического массива **ri.array**

Пример 4.41. Задание множества допустимых значений в виде массива

```
class RandcInside;
        int array[]; // набор вариантов значения
        randc bit [15:0] index; // индекс массива
        function new(input int a[]); // инициализация массива
                array = a;
        endfunction
        //function int pick;
        //      return array[index];
        //endfunction
        constraint c_size {index < array.size; };
endclass
initial begin
     RandcInside ri;
  ri = new('{1,3,5,7,9,11,13});
  repeat (ri.array.size)
      begin
        assert(ri.randomize());
        $display("Picked %2d [%0d]", ri.pick(), ri.index);
      end
end //initial
```

Класс обладает определенной степенью универсальности, поэтому иногда удобнее определять ограничение не в самом классе, а в программе его использующей. Тогда одинаковая структура данных может использоваться с различными ограничениями. При этом в классе объявляется только имя ограничения, как показано в следующем примере.

Пример 4.42. Декларация ограничения вне класса

```
class Packet;
        rand bit [7:0] length;
        rand bit [7:0] payload[];
        constraint c_valid {length > 0; payload.size == length;}
        constraint c_external;
endclass

program test;//ограничение задано в программе, использующей класс
constraint Packet::c_external {length == 1;}
```

...endprogram

Другой способ декларации ограничений вне класса, в том числе вводимых дополнительно к определенным в классе, и дающий возможность модификации ограничений в процессе исполнения теста, например, во времени или по результатам выполненных испытаний, определен в стандарте как in-line constraints. Соответствующее выражение randomization()with позволяет на лету вводить дополнительные ограничения.

6.8 Ограничения on-line
Когда пишут больше тестов, можно в конечном итоге получить большое количество ограничений. Они взаимодействуют друг с другом самым неожиданным образом, дополнительный код для их включения и отключения усложняет тест . Кроме того, постоянное добавление и Редактирование ограничений в класс может вызвать проблемы при корпоративной проектировании .
Многие тесты рандомизируют объекты в только одном месте в коде. SystemVerilog позволяет добавлять дополнительные ограничения используя функцию **randomize**()**with** . Это эквивалентно введению ограничений дополнительно к любым имеющимся.

Пример 6-23 показывает базовый класс с ограничениями, затем два **randomize()with** для введения дополнительных ограничений.

```
class Transaction;
rand bit [31:0] addr, data;
constraint c1 {addr inside{[0:100],[1000:2000]};}
endclass
Transaction t = new();
initial begin
int s;
t = new();
// addr is 50-100, 1000-1500, data < 10
assert(t.randomize() with {addr >= 50; addr <= 1500;
data < 10;});
driveBus(t);
// force addr to a specific value, data > 10
assert(t.randomize() with {addr == 2000; data > 10;});
driveBus(t);
end
```

Дополнительные ограничения добавляются к уже активищированным существующим Используйте constraint_mode, если вам нужно отключить конфликтующие ограничения. Обратите внимание, что внутри скобок with {} , SystemVerilog использует имена определенные в классе.
Поэтому Пример 6-23 используется только addr, а не t.addr.
Распространенной ошибкой является окружения строк ограничения обычными скобками вместо фигурных скобок {}. Помните,
для блоков ограничений следует использовать фигурные скобки,

SystemVerilog позволяет задавать нелинейные функции плотности распределения, например такие:

$dist_exponential — экспоненциальное распределение;

$dist_normal — гауссово распределение;

$dist_poisson — пуассоновское распределение;

$dist_uniform — равномерное распределение;

$random — равномерное в области 32-разрядных целых чисел;

$urandom — равномерное в области 32-разрядных положительных чиселж

$urandom_range — равномерное в заданном диапазоне.

Эти системные функции вызывают из встраиваемой в класс функции pre_randomize(), как показано в примере 4.47, чем обеспечивается предварительная подготовка данных в соответствии с требуемым распределением.

Пример 4.47. Определение вида распределения

```
class expon_distribution;
int value; // служебная переменная, задающая вид распределения
int WIDTH = 50, DEPTH=4, seed=1;
    function void pre_randomize();// определение вида распределения
        value = $dist_exponential(seed, DEPTH);
    endfunction
endclass
```

В общем случае функция pre_randomize() (если определена в классе) автоматически выполняется перед каждым вызовом randomize() объекта класса и используется для исполнения тех или иных операций, связанных с дальнейшим использованием рандомизированных данных, например генерацию сообщений, генерацию управляющих сигналов.

Подобная функция post_randomize() выполняется после исполнения randomize()и может задавать, например подсчет контрольной суммы или анализ иных свойств выданной совокупности случайных данных**.**

Ограничения массивов случайных данных

Самое очевидное ограничение случайного массива это задание диапазона его индексов, которое демонстрирует следующий пример.

Пример 4.54. Ограничение размера динамического массива

```
class dyn_size;
        rand reg [31:0] d[];
        constraint d_size {d.size inside {[1:10]}; }
endclass
```

Следует в обязательном порядке задавать верхнюю границу размера, иначе придется иметь дело с размерностью в сотни миллионов элементов, что существенно замедлит процесс моделирования

Если рандомизации подвергается массив, то все его элементы принимают случайные значения, но для них также может потребоваться введение ограничений. Пример 4.55 показывает возможность введения общего ограничения для всех элементов массива. Этот же пример содержит ограничение на сумму элементов массива

Пример 4.55. Ограничения данных в массиве

```
class random_array;
        rand bit [7:0] len[]; /
        constraint c_len { len.size inside {[1:8]};}
        constraint data_value {foreach len[j]  len[j]  inside {[0:127]};len.sum < 1024;)
                                }
endclass
```

Имеется также возможность генерировать упорядоченные массивы случайных данных. В следующем примере каждый очередной элемент массива меньше предыдущего.

Пример 4.56. Случайный массив упорядоченных данных

```
class Ascend;
    rand uint d[10];
    constraint c {
                foreach (d[i]) // для всех элнментов массива
```

```
    if (i>0) // кроме первого
      d[i] > d[i-1]; // следующий не может быть больше
    }
endclass
```

Рандомизация управления
Случайный мсссив является отличным способом, чтобы создать
длинные потоки случайных вклад стимулов. Но иногда это излишне
если все вы хотите сделать, это иногда ввести случайный выбоh nyts в
своей программе
6.14.1 Введение в randcase
Вы можете использовать randcase, чтобы сделать взвешенный выбор между несколькими
действия, без необходимости создания экземпляра и класса. Пример выбирает
одно из трех ветвей в зависимости от веса. Программа SystemVerilog относительные частоты (вес) выбора ветвей (1+8+1 = 10), выбирает значение из этого диапазона, а затем выбирает соответствующую ветввь.
Веса Ветвей и веса переменных не зависимяч от того, и они не должны составлять 100%.

```
int len;

.....
ALWAYS @.....begin
.....
randcase
1: len = $urandom_range(0, 2); // 10%: 0, 1, or 2
8: len = $urandom_range(3, 5); // 80%: 3, 4, or 5
1: len = $urandom_range(6, 7); // 10%: 6 or 7
endcase
.....
end

initial begin
// Level 1
randcase
one_write_wt: do_one_write();
one_read_wt: do_one_read();
seq_write_wt: do_seq_write();
seq_read_wt: do_seq_read();
endcase
end
// Level 2
task do_one_write;
randcase
mem_write_wt: do_mem_write();
io_write_wt: do_io_write();
cfg_write_wt: do_cfg_write();
endcase
endtask
task do_one_read;
randcase
mem_read_wt: do_mem_read();
io_read_wt: do_io_read();
cfg_read_wt: do_cfg_read();
endcase
endtask
```

Следует упомянуть такие предопределенные методы для случайных классов, как randcasez (случайный выбор), randomsequence (генерация случайной последовательности). Стандарт и симуляторы вводят еще целый ряд способов управления распределением генерируемых случайных данных,