通过 SYSTEMVERILOG 复用快速创建便携式激励测试

MATTHEW BALLANCE，MENTOR, A SIEMENS BUSINESS

摘要

设计与验证历来都是由特定领域的语言驱动的。关于在同一语言或方法论内复用描述的指导准则早已存在，但却没有关于跨语言复用描述的指导准则。围绕“便携式激励规范”标准的 Accellera 标准化工作，以及业内存在的可将抽象测试规范重定向到多种环境的便携式激励工具，为创建此类指导准则提供了驱动力。本白皮书提供了关于 SystemVerilog 激励和覆盖率规范结构化的指导准则，以期能利用便携式激励规范语言最大程度地实现复用。

Mentor®
A Siemens Business

WHITEPAPER

FUNCTIONAL VERIFICATION

概述

自从验证作为一个独立学科存在以来，复用验证便是一个目标。SystemVerilog 之类的语言和“通用验证方法学”(UVM) 之类的方法论，极大地促进了以事务处理为导向的仿真和加速仿真验证环境中的自动化和复用。将测试创建自动化和复用带到更广泛的环境和执行平台，是 Accellera 便携式激励规范 (PSS) 标准的目标。

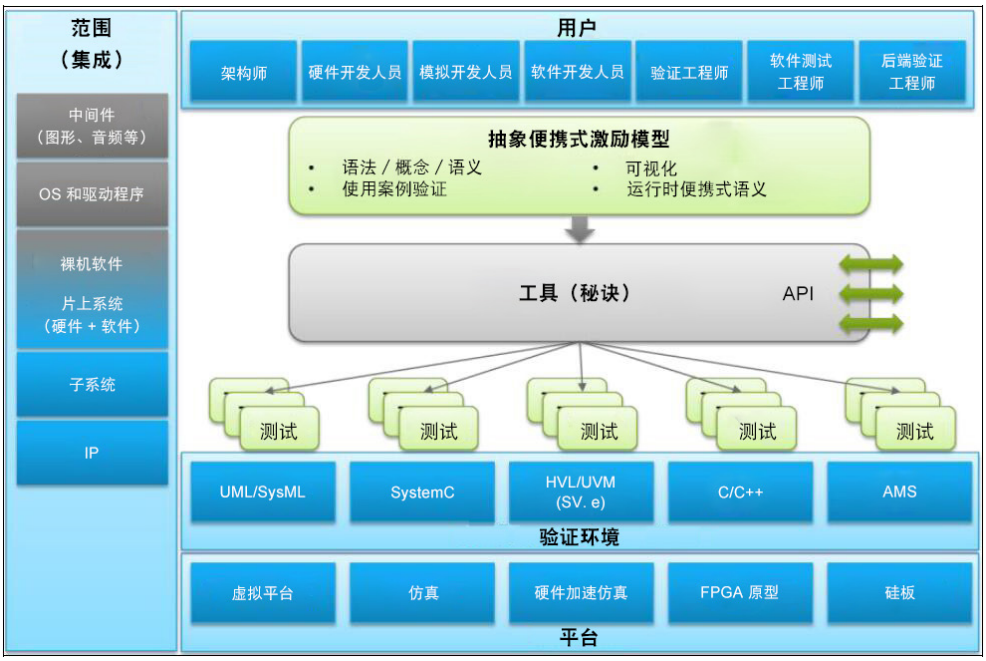

图 1 - 便携式激励规范目标

图 1[1] 总结了该标准的目标，即让多个用户群体能够跨不同平台使用和复用关于测试激励、预期结果和覆盖率目标的同一规范，包括仿真、硬件加速仿真和原型。

制定中的便携式激励规范本质上是一种声明性语言，其包括数据结构、约束、覆盖率规范特性和图表，因此即为可进行形式分析的测试程序规范。像验证领域的其他标准一样，制定中的便携式激励规范主要吸取了多家供应商和众多用户使用该领域中已经存在的工具的经验。

能够复用现有描述对新语言标准的推广和采用大有裨益，便携式激励规范也不例外。考虑到当今大部分验证是利用 SystemVerilog 完成的，因此了解 SystemVerilog 描述的哪些元素可以轻松可靠地在便携式激励规范描述中复用就显得尤为必要。本白皮书探讨了最容易在便携式激励规范中复用的 SystemVerilog 结构体，并提供了编码指导准则，以便让复用更为简单可靠。

复用目标

复用是一个相当宽泛的说法，涵盖了各种各样的工作流程。复用一个描述可以像动手重新实现一个算法一样简单（科技含量很低），原先用某种语言实现，而现在用不同语言实现。显然，原始作者的思维过程是有利用价值的，但这很难成为一个可自动化的流程。

当要在便携式激励规范描述中复用 SystemVerilog 内容时，高度自动化就是关键所在。SystemVerilog 源描述容易变动，且在项目生命周期内会不时发生改变。若由人来识别对 SystemVerilog 相关源描述的改动，然后据此更新便携式激励规范表示，将需要耗费大量人力，而且容易出错。

图 2 显示了 SystemVerilog 到便携式激励规范的典型复用情形。共享数据结构的主源描述以及对这些数据结构的约束是在 SystemVerilog 中予以维护的。这些数据结构和约束的便携式激励规范表示，是直接从 SystemVerilog 表示自动派生的。所产生的这种便携式激励规范描述随后可在用户创建的便携式激励规范描述中加以利用。

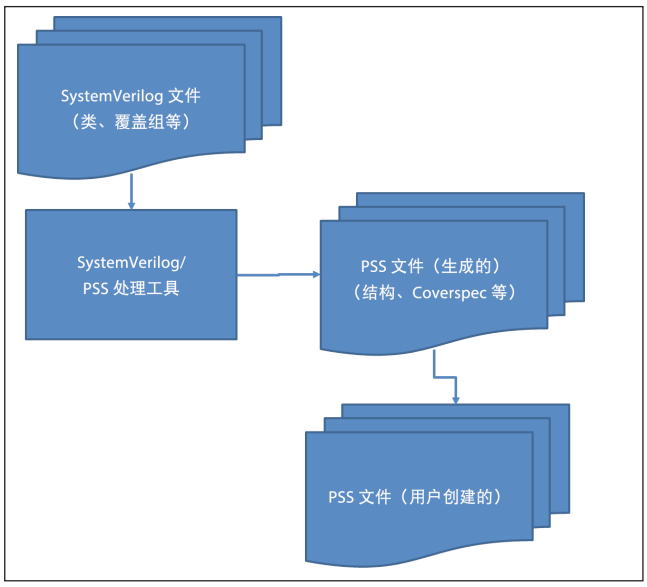

图 2 - 典型 SystemVerilog 到便携式激励复用

当然还有一种复用流程，那就是在便携式激励规范语言中声明所有共享数据结构和对这些数据结构的约束，并自动派生 SystemVerilog 表示，如图 3 所示。随着新兴便携式激励规范标准获得行业更高程度的接受和采纳，这种流程将会更为常见。然而，在目前以及可预见的未来，用 SystemVerilog 编写且能用作便携式激励规范描述一部分的内容要多得多。

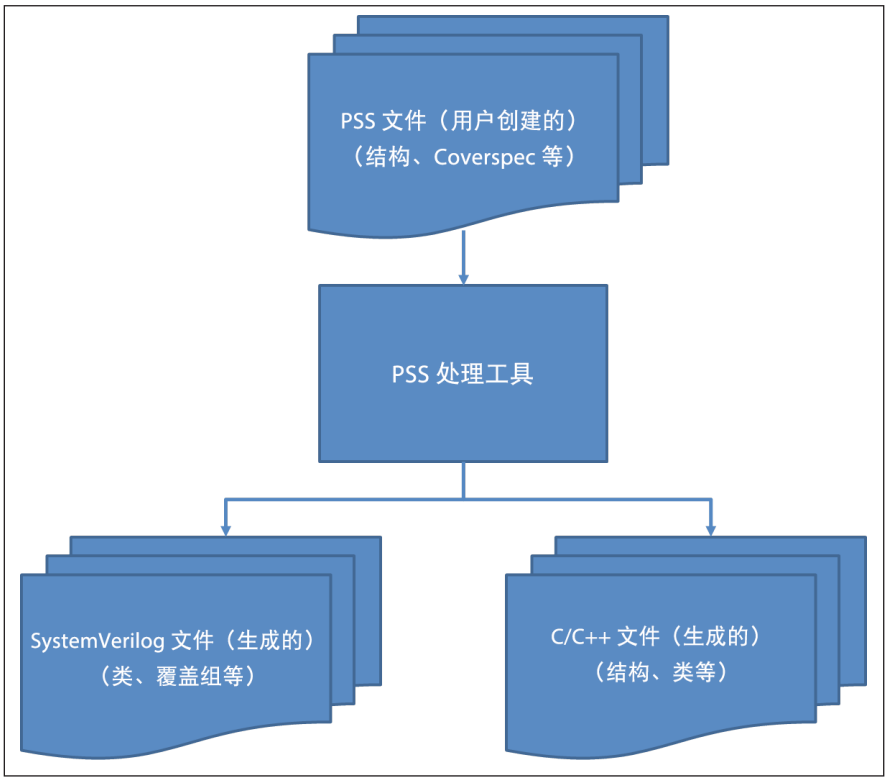

图 3 - PSS 主源描述复用

本白皮书提出的关于 SystemVerilog 结构化以供复用的指导准则，反映了对通过全自动化流程复用所捕获信息的侧重，以及从 Mentor Graphics Questa inFact 便携式测试工具的使用经验发展出的最佳实践。

SYSTEMVERILOG 结构化以供复用

SystemVerilog 是一种针对寄存器传输级 (RTL) 设计和验证的功能完备的语言 [2]。因此，它含有从 Verilog 语言（用于描述硬件结构体）继承的结构体。它还含有面向对象的编程结构体，以及用于自动生成激励和收集功能覆盖率的结构体，这些结构体皆可用于验证硬件描述。新兴便携式激励规范语言主要是一种声明性语言，其过程结构体非常少。相比之下，SystemVerilog 则主要是一种过程语言，其声明性结构体较少。

SystemVerilog 描述中最容易复用为便携式激励规范描述一部分的结构是声明性结构体，具体而言就是数据结构和对这些数据结构字段的约束，以及用于功能覆盖率目标建模的覆盖组。

复用最佳实践的存在时间同语言互操作的历史一样长。例如，为了从 Java 通过 Java 本地接口 (JNI) 最佳地复用 C 库，就存在相应的 C 库结构化最佳实践。*Best Practices for using the Java Native Interface*（“使用 Java 本地接口的最佳实践”）一文非常好地总结了这些最佳实践 [3]。SystemVerilog 结构化以供复用的目标是创建独立自足的描述元素，它们既可以从 SystemVerilog 描述中予以提取，亦可添加到便携式激励规范描述中。一般而言，不利于 SystemVerilog 描述复用的设计模式是将过程描述和声明性描述混在一起的模式，或是会影响声明性描述封装的模式。多数情况下，有助于复用的最佳实践也是能产生良好封装代码的 SystemVerilog 优秀编码实践。当然，有些情况下使用限制复用的结构体和模式是有实用价值或极为必要的。即便在这些情况下，只需一点远见就能保持可复用结构体与实效设计模式之间的合理平衡。

内嵌随机化约束

当一个对象随机化时，SystemVerilog 允许指定内嵌约束。这样一来，可以方便地自定义随机事务，从而创建定向随机测试序列。图 4 显示了一个定向随机测试，其利用现有事务类和内嵌约束来执行一系列读写操作。

```
class transaction extends uvm_sequence_item;
       `uvm_object_utils(transaction)

       rand bit[31:0]        addr;
       rand bit[31:0]        data;
       rand bit[1:0]                size;
       rand bit                     rnw;
endclass


class rw_test_sequence extends uvm_sequence;
       `uvm_object_utils(rw_test_sequence)

       task body();
              transaction t1 = transaction::type_id::create("t1");
              transaction t2 = transaction::type_id::create("t2");

              for (int unsigned offset=0; offset<'h1000; offset+=4) begin
                     // Do a write
                     start_item(t1);
                     assert(t1.randomize() with { // Size and data left open
                                   t1.addr == 'h8000+offset;
                                   t1.rnw == 0;
                            });
                     finish_item(t1);

                     // Do a read
                     start_item(t2);
                     t2.addr = t1.addr;
                     t2.rnw = 1;
                     t2.size = t1.size; // Ensure same-sized read is performed
                     finish_item(t2);
              end
       endtask
endclass
```

图 4 – 内嵌随机化示例

在 SystemVerilog 验证平台中使用这种设计模式可能非常实用，但却会限制复用。附加内嵌约束是在一个过程块中引入的，因此常常会包括对过程块内变量的引用。所以说，复用这些内嵌约束即使并非不可能，也非常难以实现。

毫无疑问，内嵌约束还可能会被误用。笔者曾碰到过这样的验证平台环境，其中有高达百行的内嵌约束，目的仅仅是为了确保类字段之间的关系有效。更糟糕的是，这些大型内嵌约束块会被复制到多个测试中，当需要调整这些基本约束时，维护工作将令人头疼不已。

应将常见约束从内嵌约束块中剔除，以提升代码可维护性和复用。图 5 显示了通过创建读事务和写事务类来剔除读 / 写内嵌约束的方法。这样可以提高测试的可读性和可维护性，并提供更为具体的元素以便在便携式激励规范描述中复用。

```
class read_transaction extends transaction;
  `uvm_object_utils(read_transaction)

  constraint read_c { rnw == 1; }
endclass

class write_transaction extends transaction;
  `uvm_object_utils(write_transaction)

  constraint write_c { rnw == 0; }
endclass
```

图 5 - 剔除内嵌约束

动态启用 / 禁用约束块

SystemVerilog 过程代码与声明性约束之间的另一种相互作用是支持动态启用和禁用约束块。在理想情况下，动态禁用约束的使用范围非常有限，主要是为了完成启用非法事务生成之类的事情。在使用动态控制约束块时，为了成功实现随机化，必须禁用至少一个块，而这种使用方法存在极大的问题。图 6 显示了一个操作序列-项类，其指定两个彼此冲突的约束：一个约束设置“正常”工作模式，另一个约束设置“扩展”工作模式。此类的用户必须知道，在将对象随机化之前，应禁用至少一个 normal_mode_c 或 ext_mode_c 约束。

```
typedef enum { MODE_NORMAL_1, MODE_NORMAL_2, MODE_EXT_1, MODE_EXT_2} mode_t;

class op_item extends uvm_sequence_item;
  `uvm_object_utils(op_item)

  rand mode_t          mode;
  // ... Other fields

  constraint normal_mode_c {
    mode inside {MODE_NORMAL_1, MODE_NORMAL_2};
  }

  constraint ext_mode_c {
    mode inside {MODE_EXT_1, MODE_EXT_2};
  }

endclass
```

图 6 - 动态控制的冲突约束

从复用观点看，使用动态控制的约束存在两个挑战。目前的便携式激励规范语言并未提供动态控制约束的条件。所以，至少要在 PSS 描述中做出额外变更，以改写从 SystemVerilog 导入的描述。此外，使用彼此冲突的约束对 SystemVerilog 和便携式激励规范描述均提出了可用性挑战。启用 / 禁用约束块的有效组合必须记录在某处。在最好的情况下，有效的已启用 / 禁用约束块会作为注释予以捕获，其不大可能传播到便携式激励规范描述。在最差情况下，启用 / 禁用约束块的有效组合是在工作测试序列中捕获的“部落文化”。

用类层级和约束重载代替动态启用 / 禁用约束，得到的SystemVerilog序列项更易于使用，且SystemVerilog描述易于复用为便携式激励规范的一部分。利用类层级结构化来处理常常通过动态启用 / 禁用约束实现的案例，有两种等效方法。第一种方法是加性约束重构，如图 7 所示。采用这种方法时，派生类仅加入实现预期目的所需的约束。因此，normal_op_item 类会增加约束以确保仅产生“正常”模式，而 ext_op_item 类则会增加约束以确保仅产生“扩展”模式。这是一种很好的通用方法。

```
typedef enum { MODE_NORMAL_1, MODE_NORMAL_2, MODE_EXT_1, MODE_EXT_2} mode_t;

class op_item extends uvm_sequence_item;
  `uvm_object_utils(op_item)

  rand mode_t          mode;
  // ... Other fields
endclass

class normal_op_item extends op_item;
  `uvm_object_utils(normal_op_item)

  constraint normal_mode_c {
    mode inside {MODE_NORMAL_1, MODE_NORMAL_2};
  }
endclass

class ext_op_item extends op_item;
  `uvm_object_utils(ext_op_item)

  constraint ext_mode_c {
    mode inside {MODE_EXT_1, MODE_EXT_2};
  }
endclass
```

图 7 - 加性约束重构

第二种方法是减性约束重构。这种方法更常用于创建异常使用案例，例如错误注入。以当前的例子为例，假设“正常”模式是大部分时间使用的模式。

```
typedef enum { MODE_NORMAL_1, MODE_NORMAL_2, MODE_EXT_1, MODE_EXT_2} mode_t;

class normal_op_item extends uvm_sequence_item;
  `uvm_object_utils(op_item)

  rand mode_t          mode;
  // ... Other fields

  constraint normal_mode_c {
    mode inside {MODE_NORMAL_1, MODE_NORMAL_2};
  }

endclass

class ext_op_item extends normal_op_item;
  `uvm_object_utils(ext_op_item)

  constraint normal_mode_c {
    // empty constraint - follows reuse guideline #2
  }

  constraint ext_mode_c {
    mode inside {MODE_EXT_1, MODE_EXT_2};
  }

endclass
```

图 8 - 减性约束重构

图 8 显示了 normal_op_item 类的定义，其中含有实施“正常”操作的约束。为了创建“异常”条件（原本通过禁用 'normal_mode_c' 约束来创建），ext_op_item 类用空约束覆盖 normal_mode_c 约束，从而以声明方式有效禁用此约束。

使用类层级和约束重构得到的类在 SystemVerilog 环境中更易于复用。它还能产生可在便携式激励规范描述中复用的 SystemVerilog 描述，只需将控制语句从过程描述移动到描述的声明部分即可。

引用全局数据

引用类外部的全局数据项给复用带来了挑战，这在 SystemVerilog 中和便携式激励规范描述中均是如此。图 9 显示了这样一种情况，其中 transform_item 类中的 coeff 字段受 global_data_pkg 包中的 min 和 max 全局字段限制。

```
package global_data_pkg;
  bit[31:0]           coeff_max;
  bit[31:0]           coeff_min;

endpackage

class transform_item extends uvm_sequence_item;
  `uvm_object_utils(transform_item)

  rand bit[31:0]        coeff;

  constraint coeff_c {
    coeff >= global_data_pkg::coeff_min;
    coeff <= global_data_pkg::coeff_max;
  }
endclass
```

图 9 - 对全局数据的约束引用

当然，这有它方便的一面，每个验证平台环境中只需对系数界限设置一次。缺点是这会让复用变得更困难。也许系数界限在原始验证平台环境中是全局性的，但情况并非总是如此。或许在下一个验证平台环境中会有两个 UVM 代理使用 transform_item 类。而每个代理可能需要使用不同的系数界限。

```
class transform_item extends uvm_sequence_item;
  `uvm_object_utils(transform_item)

  bit[31:0]             coeff_min;
  bit[31:0]             coeff_max;

  rand bit[31:0]        coeff;

  constraint coeff_c {
    coeff >= coeff_min;
    coeff <= coeff_max;
  }
endclass
```

图 10 - 移除全局数据引用

图 10 显示了一种移除全局数据引用的方法，即在类内部创建字段。SystemVerilog 过程代码可以随机化重构的 transform_item 类，不过在随机化该类之前，需要设置 coeff_min 和 coeff_max。但是，重构代码得到的类是独立自足的，因此更易于复用。独立自足的类在便携式激励规范中同样更容易复用，因为其没有对外部实体的引用。

可复用功能覆盖率

SystemVerilog 覆盖组为监视和聚合验证平台数据提供了一个异常灵活的结构体。然而，该结构体的非凡灵活性会影响其可复用性。

```
class config_item extends uvm_object;
  `uvm_object_utils(config_item)

  rand bit[3:0]             mode1;
  rand bit[15:0]            mode1_coeff;
  rand bit[3:0]             mode2;
  rand bit[15:0]            mode2_coeff;

endclass

covergroup config_cg(ref config_item item);

  mode1_cp : coverpoint (item.mode1) {
    bins mode1[] = {[0:15]};
  }

  mode1_coeff_cp : coverpoint (item.mode1_coeff) {
    bins mode1_coeff_min = {0};
    bins mode1_coeff_mid[14] = {[1:'hfffe]};
    bins mode1_coeff_max = {'hffff};
  }

  mode1_coeff_cross : cross mode1_cp, mode1_coeff_cp;

  mode2_cp : coverpoint (item.mode2) {
    bins mode2[] = {[0:15]};
  }

  mode2_coeff_cp : coverpoint (item.mode2_coeff) {
    bins mode2_coeff_min = {0};
    bins mode2_coeff_mid[14] = {[1:'hfffe]};
    bins mode2_coeff_max = {'hffff};
  }

  mode2_coeff_cross : cross mode2_cp, mode2_coeff_cp;

endgroup
```

图 11 - 配置类和功能覆盖率

图 11 显示了一个 SystemVerilog 覆盖组，其结构非常适合在便携式激励规范描述中复用。所有覆盖点和交互覆盖点都会引用 config_item 类字段项，其被指定为覆盖组的一个参数。这样一来，便可确保覆盖组的独立自足。此外，覆盖组所监视的类就是用于生成激励的类。这样就能生成可高效实现覆盖目标的配置，无需用户提供其他指令来将激励生成类同用于捕捉覆盖率数据的类联系起来。

最佳实践总结

针对本部分所述模块性和复用的最佳实践，有助于实现跨 SystemVerilog 环境和便携式激励规范描述的复用：

1. 避免使用内嵌约束。
2. 避免从过程代码启用和禁用约束。
3. 避免从约束引用类外部变量。
4. 使用结构化功能覆盖率描述，以明确引用单个类，最好是激励生成类。

以上章节说明，替代性建模方法可以实现常常由上述妨碍复用的结构体处理的一些情形。

结语

新兴便携式激励规范标准允许用一个可跨测试环境（从 UVM 到软件驱动的测试环境）复用的描述，捕获测试激励、预期结果和覆盖率目标的单一表示。利用部分现有的 SystemVerilog 描述可以加速便携式激励规范描述的创建，因为有非常多的内容已经通过 SystemVerilog 予以创建。遵循本白皮书所述的一些简单编码指导准则，可以让 SystemVerilog 类、约束和覆盖组更好地在便携式激励规范描述中自动复用。通过复用现有的大量 SystemVerilog 描述，如事务处理和设备配置类，用户可以更迅速地从便携式激励工具获益，例如验证目标高效收敛和自动创建 C 测试等。

参考文献

[1] “Proposed Portable Stimulus Diagram”, Accellera Portable Stimulus Working Group, December 2014

[2] “IEEE 1800-2012: SystemVerilog – Unified Hardware Design, Specification, and Verification Language,” IEEE Standards Association, February 2013.

[3] M. Dawson, G. Johnson, A. Low, “Best Practices for Using the Java Native Interface”, IBM developerWorks, July 2009.

公司总部
Mentor Graphics Corporation
8005 S.W. Boeckman Road
Wilsonville, Oregon 97070 USA
电话：+1-503-685-7000
传真：+1-503-685-1204

销售和产品信息
电话：+86-21-6101-6301
sales_info@mentor.com

上海
明导（上海）电子科技有限公司
上海市浦东新区杨高南路 759 号
陆家嘴世纪金融广场 2 号楼 5 楼
邮编：200127
电话：+86-21-6101-6301
传真：+86-21-5047-1379

北京
明导（上海）电子科技有限公司
北京办事处
北京市南礼士路 66 号
建威大厦 1512 室
邮编：100045
电话：+86-10-5930-4001
传真：+86-10-6808-0319

深圳
明导（上海）电子科技有限公司
深圳办事处
深圳市福田区金田路 3088 号
中洲大厦 24 楼 2401 室
邮编：518040
电话：+86-755-8282-2700
传真：+86-755-8826-7750

MGC 11-17 TECH15920-w-CN