利用形式验证检查 SoC 连通性的正确性

MARK HANDOVER，MENTOR GRAPHICS 欧洲区应用工程师
ABDELOUAHAB AYARI，MENTOR GRAPHICS 欧洲区应用工程师

Mentor®
A Siemens Business

VERIFICATION WHITEPAPER

www.mentor.com

简介

连通性检查涉及验证器件布线。它相当于问这样一个问题："设计元素是否被正确装配？"更准确地说，它是在验证设计中的逻辑模块之间的连接是否正确，例如：模块 B1 上的输出 A 是否正确连接到模块 B2 上的输入 A'。这常常是很困难的验证任务。设计包含数以千计的导线，这些导线的正确性可能都需要检查，因此要检查的连接数量是一个问题。

调试提出了另一个次要的但常常同样具有挑战性的问题。原因是，虽然采用定向或约束随机方法通过动态测试检查连通性肯定能发现一些连通性错误，但问题只会表现为被测模块内部的功能性问题，而不一定能帮助查明问题连接。使用断言可以在源头捕获设计错误，从而减轻调试问题。但是，所需的检查量仍然可能令人瞠目。

为应对此类挑战，形式验证为我们提供了一种快速、详尽且支持高效调试的解决方案。传统上，芯片级形式验证确实不可行。该方法通常以模块级别为目标，使状态空间的规模保持在适当水平。但是，鉴于连通性检查仅集中在布线上（与模块级别的复杂度相比，布线一般是器件的简单部分），借助一些假设，状态空间可以减小到可管理的规模。这种简化的性质取决于所需检查的类型。

本文首先会概述几种类型的连通性检查，然后详细介绍一种新型半自动验证流程（包括代码）。已有一些 Mentor Graphics 使用该流程来简化连通性检查。该流程基于一个脚本环境，围绕该环境提供了充足的信息以方便用户开始实施新的验证方法。

点对点连通性检查的类型

直接点对点检查

连通性检查的最简单形式是点对点检查 —— 端口 A 是否连接到端口 B？这是就同一层次结构而言的。

例如，如果一个设计有八个模块，所有模块都位于顶层，那么验证只需要检查这八个模块与顶层之间的连接。

在这种情况下，我们只需把八个子模块进行黑盒化处理，而不必对整个器件进行建模。检查不依赖于模块内容，因此无需读取这些模块的 HDL。

跨层次结构的直接点对点检查

这在本质上与简单检查方法相似，不过检查的是位于一个层次结构中一个模块上的端口是否在物理上正确连接到位于另一层次结构中的一个模块，或者位于信号源的单个端口是否连接到多个端点。

以对存储器的写使能为例。它可能起源于单个顶层输入管脚，但可以连接到跨许多不同层次位置的许多存储器实例。

现在涉及层次结构，因此无法将上方的模块实例统一进行黑盒化处理。黑盒化处理应该在最高层级上执行，但只能用于那些不在写使能路径上或可能影响写使能连通性的逻辑路径上的模块。尽管更具挑战性并需要一些设计知识，但这种更具选择性的黑盒方法仍然可以显著简化状态空间。

其他类型的检查

验证器件的模块间连通性可能需要进行多种类型的检查。到目前为止，我们仅考虑了点对点检查，即所有条件下 A = B，层级可以相同或不同。许多连接都是这种性质的，但也可能需要其他类型的检查。

我们来看几个例子。

条件点对点检查

两点之间的连接可能取决于系统中的其他行为或另一个信号的状态。例如，当验证管脚多路复用时，所选的 IO 路径将取决于控制信号的值。令情况变得复杂的是，信号的目的地可能是一个相反值，执行检查时可能还需要考虑这一点。

有延迟的点对点

某些情况下需要点对点连接，但传播可能要花费若干周期，而不是立即发生。因此，这就需要完善点对点检查。

无延迟的点对点

这类似于前面所述的简单点对点检查，但有一个重大区别：用户要求检查明确验证不仅 A 连接到 B，而且路径上没有时序逻辑。

构造检查

鉴于需要创建大量检查才能全面检查器件连通性情况，用户如何创建所需的断言？

一种常见方法是使用格式特别编制的电子表格，其中详细说明了应连接的各个点、涉及的路径延迟、反转、条件等。然后，工具或脚本解析电子表格并将其转换为断言语言，例如 SystemVerilog 断言 (SVA) 或属性说明语言 (PSL)。图 1 显示了一个带有一些连通性信息的电子表格描述范例。

连通性信息（电子表格描述）

检查类型	输入 1	输入 2	Condition	延迟
cond	command_outa	command_out	gnta	0
cond	transid_outa	transid_out	gnta	0
cond	command_outb	command_out	gntb	0
cond	transid_outb	transid_out	gntb	0
connect	data_in	data_cmd		0
connect	addr_in	addr_cmd		0

图 1

我们来浏览一下该电子表格。我们指定了两种检查类型："cond" 指条件连接，"connect" 指无条件的直接连接。这将允许我们在创建检查器期间创建不同断言类型。"输入1" 和 "输入2" 字段详细列出了设计中要进行连通性检查的起点和终点。"条件" 列用于详细说明需要设置什么信号才允许点对点连接为真。"connect" 检查没有条件，检查将是直接、无条件的。最后，所有延迟字段都是 0，表示所有连接都没有延迟。

一旦电子表格格式固定并填充内容，便可使用适当的工具或脚本来解析电子表格和创建断言，而断言将作为目标送入形式化工具。一种方法是使用通用属性模板，然后在单独的检查器描述中添加每个属性实例的连通性信息。这样就可以将其绑定（使用 SystemVerilog 的 bind 结构体）到设计的顶层。

图 2 显示了两个通用属性模板。

```
property cond_p (clk, rst, cond, inA, inB);
  @(posedge clk) disable iff (rst)
      cond |-> inA == inB;
endproperty:cond_p
property connect_p (clk, rst, inA, inB);
  @(posedge clk) disable iff (rst)
   inA == inB;
endproperty:connect_p
```

图 2

属性 cond_p 支持条件检查，而 connect_p 支持直接无条件检查。

此模板文件可以包含许多独特类型的连通性检查，一旦明确便无需用户编辑。该文件不包含任何设计信息，因而与项目无关，可以重复使用。

从电子表格自动创建的源就是检查器详细信息，其中包含模板文件中不同检查的实例，并添加了适当的信号名称。一个例子如图 3 所示。

```
check_cond_1:
   assert property (cond_p(.clk(clk), .rst(rst), .cond(gnt_a),
                    .inA(command_outa), .inB(command_out)));

check_connect_1:
   assert property (connect_p(.clk(clk), .rst(rst),
                    .inA(transid_outb), .inB(transid_out)));
```

图 3

其他注意事项

时钟

设计不可避免地包含多个时钟。通过形式验证，未定义的时钟会产生与这些时钟明确相关的设计逻辑和任何断言的抽象。来自抽象域的信号成为形式验证控制点，这可能会导致意外激发。

为了避免工具执行任何抽象，必须定义所有时钟。但是，某些设计有很多 10s 的时钟，所以这可能很麻烦。连通性检查常常不验证时钟逻辑，因此定义时钟貌似是不必要的任务。然而，为检查连通性而创建的断言会使用时钟。

理论上讲，用户只需定义那些与断言相关的时钟，以及那些影响断言所检查路径上的时序逻辑的时钟。不过，鉴于难以识别路径上的时序逻辑，这可能不是一个容易执行的简化操作。

实践中，显式指定和定义设计中的所有时钟可能会更容易。由于连通性检查通常不检查设计的时序行为，或者至多检查连接是否存在延迟（或没有延迟），因此一般可以给所有时钟指定同一频率。这就大大简化了用户为形式工具定义时钟信息的任务。

分阶段测试

被测器件可能有多种工作模式，这些模式可能会影响可激活的连通性路径或设计逻辑。测试应确保每种有效模式都得到测试，同时还要充分利用所有设计最小化（即黑盒化处理）的机会 —— 针对具体模式进行配置时可能会有这种机会。

用户还应注意所执行测试的方面，并相应地对测试进行分组以便支持分阶段方法，这样测试环境的设置会更简单。例如，如果存储器连接测试属于一组必须进行的连通性检查，并且所有存储器仅存在于少数几个子模块中，则除这些子模块外的所有设计都可以进行黑盒化处理。这种特定的最小化对于连通性检查的另一个方面（例如检查 IP 接口或网桥连接）可能是不可行的。其他测试方面可能需要不同的最小化策略，因此可以在测试策略的第二阶段中定义，并在第三阶段和第四阶段中进一步定义设置策略。

详细信息：实施更高效流程的工具

某些 Mentor Graphics 客户使用的方法（比如上面的例子）一般遵循一套通用步骤：首先，声明检查器的一个实例。然后，在适当的字段中添加适当的信号名称。使用 Questa Formal，此方法适用于 Verilog、VHDL、混合语言设计和混合语言层次结构。本文是我们努力让其他工程团队能够以最少的工作使用类似验证流程的一部分。在我们的方法中，我们定义了连通性规范电子表格的格式，并且编写了一个脚本来创建 SVA 或 PSL 检查器。我们还创建了一组属性模板，以便支持多种类型的连通性检查。该半自动化流程的详细信息（包括代码）说明如下。

为了能够更好地部署这种连通性检查方法，我们基于脚本的新环境允许自动创建各种所需文件。我们开发了一个 Perl 脚本 GenConn.pl，利用它来解析连通性信息的文本文件，创建 SVA 或 PSL 检查器，还可以创建 Questa Formal 的 makefile。为此需要定义连通性数据的格式，然后作为制表符分隔值 (TSV) 或逗号分隔值 (CSV) 的文件提供给脚本。

目前，脚本可以支持和创建七种类型的连通性检查：

- 点对点，有或无延迟
- 条件点对点，有或无延迟
- 互斥信号
- 接高电平的信号
- 接低电平的信号

要创建这些类型的检查器，用户需要填充连通性规范文件。该文件的格式详见图 4。

连通性规范

检查器关键字	参数 1	参数 2	参数 3	参数 4
connect	信号 1	信号 2		
connect_delay	信号 1	信号 2	延迟值	
cond	信号 1	信号 2	条件信号	
cond_delay	信号 1	信号 2	条件信号	延迟值
mutex	信号 1	信号 2	信号 3	信号 4
tie_hi	信号 1			
tie_lo	信号 1			

图 4

“检查器关键字”表示用户希望推断的检查类型，“信号 ...”和“条件信号”条目是指向要检查连通性的设计信号或端口的层次路径。“延迟值”是时序延迟周期数，须为整数。

例如，假设我们要检查信号 top.en 到 top.u1.u2.enable 的连通性，并且该路径上应有两个周期的时序延迟。图 5 显示了规范文件中该条目的样子。请注意，对于互斥检查器，虽然表中显示了四个连接，但实际上可以指定任意数量的连接。

connect_delay	top.en	top.u1.u2.enable	2

图 5

除连通性信息外，规范文件还应包括被测设计的名称、时钟的名称以及检查器将使用的复位。无论高电平有效还是低电平有效，都需要提供复位感测。这些信息应按照如下格式指定：

- Design <DUT 名称>
- Clock <检查器时钟名称>
- Reset <检查器复位名称>
- Reset_sense <低或高>

连通性规范文件需要以 TSV 或 CSV 格式传递给脚本。完整 TSV 格式连通性规范文件的例子如图 6 所示。

```
# Connectivity Specification file v1.0

design        cpu_top
clock         clk
reset         rst
reset_sense   low

connect       cpu_top.cpu_if1.req        cpu_top.arbiter1.req_a
cond          cpu_top.mux1.ux.uy.a       cpu_top.cmdack_outa      cpu_top.ack
cond_delay    cpu_top.cmd_outa           cpu_top.outp[7:0]        cpu_top.gnt_a        2
mutex         cpu_top.gnta               cpu_top.gntb             cpu_top.gntc
tie_hi        cpu_top.mem1.ce
tie_lo        cpu_top.mem2.enb
```

图 6

一旦以正确格式描述了完整的连通性规范，便可将其传递给脚本以创建检查器。

该脚本可以接受多个参数，如图 7 所示。

```
Usage : GenConn.pl [options] <input specification file>
 Options:
          -tsv : Format of specification file is tab separated values (Default)
          -csv : Format of specification file is comma separated values
          -sva : Output checkers required as SVA (Default)
          -psl : Output checkers required as PSL
          -o   : Specify output checker filename
          -l   : Specify output log filename
          -s   : Specify output connectivity specification filename
          -m   : Generate Questa Formal Makefile
```

图 7

默认情况下，预期输入格式为 TSV，检查器输出文件（名为 “checkers.sv”）将采用 SVA 格式。不过，用户可以通过指定适当的选项来更改默认行为。

脚本会自动创建输出连通性规范文件。它是 TSV 或 CSV 规范输入的副本，但每个条目都包括所创建检查器的名称。该文件的默认名称为 checker_conn_spec，可使用 -s 开关予以覆盖；扩展名为 .tsv 或 .csv，具体取决于输入文件的格式。

输出文件 checkers.sv 包含用于构建检查器的所有必要信息。为了简化使用，先前说明的 “属性模板” 已经固定，并由脚本自动创建。检查器模板本身、检查器实例化和绑定信息都包含在一个检查器文件中。图 8（注意下一页仍有代码）显示了 SVA 风格 checkers.sv 文件输出的例子。

```
package connectivity_pack;

  property cond_p (clk, rst, cond, inA, inB);
    @(posedge clk) disable iff (rst)
      cond |-> inA == inB;
  endproperty:cond_p

  property cond_dly_p (clk, rst, cond, inA, inB, delayC);
    @(posedge clk) disable iff (rst)
      cond |-> ##delayC (inB == $past(inA,delayC));
  endproperty:cond_dly_p

  property connect_p (clk, rst, inA, inB);
    @(posedge clk) disable iff (rst)
      inA == inB;
  endproperty:connect_p

  property connect_dly_p (clk, rst, inA, inB, delayC);
    @(posedge clk) disable iff (rst)
      1'b1 ##delayC 1'b1 |-> (inB == $past(inA,delayC));
  endproperty:connect_dly_p

  property mutex_p (clk, rst, inA);
    @(posedge clk) disable iff (rst)
      $onehot0(inA);
  endproperty:mutex_p

  property tied_high_p (clk, rst, inA);
    @(posedge clk) disable iff (rst)
      &(inA);
  endproperty:tied_high_p

  property tied_low_p (clk, rst, inA);
    @(posedge clk) disable iff (rst)
      ~|(inA);
  endproperty:tied_low_p

endpackage:connectivity_pack
```

```
module checkers(input logic clk, rst);

 import connectivity_pack::*;

  // Connect: cpu_top.mux1.ux.uy.a cpu_top.cmdack_outa
       CHECK_CONNECT_2: assert property (
         connect_p(.clk(clk),.rst(rst),.inA(cpu_top.mux1.ux.uy.a),
                      .inB(cpu_top.cmdack_outa)));

 // Cond: cpu_top.command_outa cpu_top.mux1.outp[73:70] cpu_top.arbiter1.gnt_a
       CHECK_COND_1: assert property (
         cond_p(.clk(clk),.rst(rst),.cond(cpu_top.arbiter1.gnt_a),
                      .inA(cpu_top command_outa), .inB(cpu_top.mux1.outp[73:70])));

 // Mutex: cpu_top.gnta,cpu_top.gntb,cpu_top.gntc
       CHECK_MUTEX_1: assert property (
         mutex_p(.clk(clk),.rst(rst),.inA({cpu_top.gnta,cpu_top.gntb,cpu_top.gntc})));

endmodule: checkers

module bindings;

 bind cpu_top checkers u_checkers (.*);

endmodule:bindings
```

图 8

选择生成的 makefile 允许用户编译所创建的 SVA 或 PSL 检查器文件以用于 Questa Formal，然后运行形式分析。

该 makefile 名为 `Makefile _ ConnCheck`。它有三个条目：

- `compile _ checkers`：编译 SVA 或 PSL
- `compile _ formal _ model`：运行 CSL 流程以构建形式模型
- `run _ formal`：运行 Questa Formal "证明" 流程

还有一个 run_all 条目，它允许依次执行所有三个步骤。为了运行 makefile 中的所有步骤，用户需要执行：

```
make -f Makefile _ ConnCheck run _ all
```

运行形式编译和证明步骤的结果分别放在目录 "results/csl" 和 "results/prove" 中。

Makefile _ ConnCheck 文件具有成功编译和运行 Questa Formal 所需的基本条目，但它更多地是作为模板提供，用户在使用之前很可能需要进行编辑。

例如，makefile 没有引用形式验证的控制文件（用于定义时钟、设置约束等），因此可能需要创建和指定该文件。

还有一个附加脚本 GenDoc.pl。此脚本的作用是将形式结果注释到 checker _ conn _ spec 文件上，该文件是自动生成并加注了检查器名称的连通性规范。GenDoc 脚本应在获得形式 "证明" 结果后运行。

该脚本可以接受多个参数，如图 9 所示。

```
 Usage : GenDoc.pl [options]
Options:
        -i : Connectivity specification input filename
        -o : Connectivity specification output filename
        -r : Prove report filename
```

图 9

默认输入和输出文件名为：

- 连通性规范输入文件名：checker _ conn _ spec.tsv
- 连通性规范输出文件：conn _ spec _ results（后缀取决于输入文件格式）
- 证明报告文件名：results/prove/0in _ prove.rpt

所有这些默认值都可以使用适当的开关予以覆盖。

在 "证明" 形式运行之后，生成的输出文件会详细说明每个连通性规范条目以及检查器名称和状态，如图 10 所示。

PROOF	CHECK_CONNECT_1	connect	cpu_top.cpuif1.req	cpu_top.arb1.req_a	
FIRING	CHECK_TIE_LOW_1	tie_lo	cpu_top.arb1.gnt_b		
INCONCLUSIVE	CHECK_MUTEX_1	mutex	cpu_top.data_out_ena	cpu_top.data_out.enb	
PROOF	CHECK_MUTEX_2	mutex	cpu_top.gnta	cpu_top.gntb	cpu_top.gntc

图 10

下一页上的图 11 给出了可用于运行完整连通性检查流程的命令示例，图 12 显示了整个流程。

```
# Compile RTL design files

vlog -f filelist.vl -work work


# Run checker creation script

GenConn.pl conn_spec.tsv -s my_checker_conn_spec -m -tsv


# Run generated Makefile for CSL and Prove steps

make -f  Makefile_ConnCheck run_all


# Run GenDoc to annotate Prove results

GenDoc.pl -i my_checker_conn_spec.tsv -o conn_spec_results
```

图 11

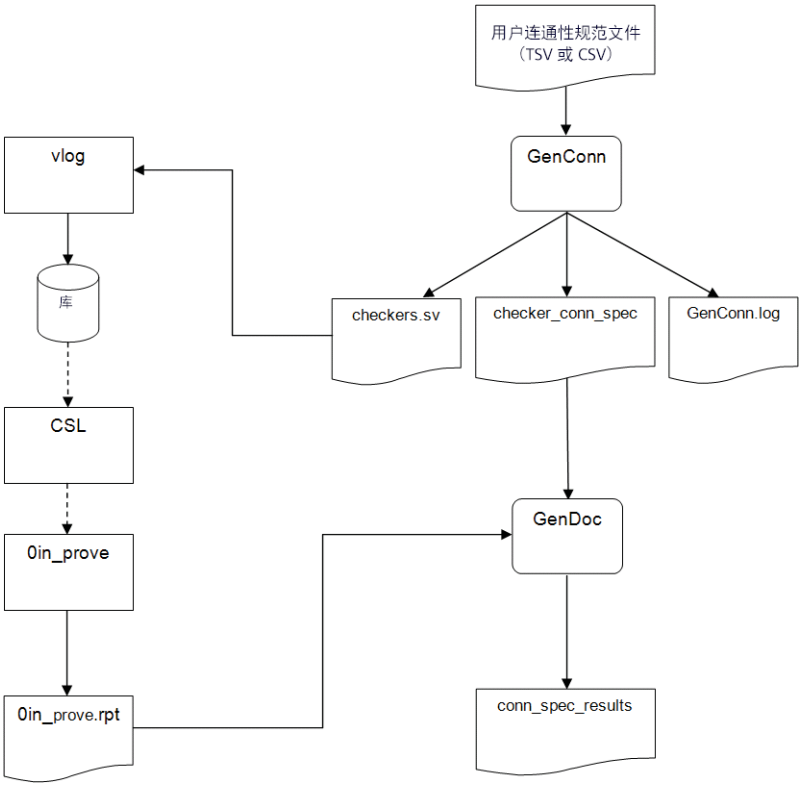

图 12

其他应用

连通性检查在许多应用中都很有价值。下面介绍几个例子。

焊盘环检查

复杂器件具有多种配置，SoC 中的 IO 不可避免地会涉及复杂的多路复用焊盘。必须验证所有配置下的焊盘环，检查每种模式下是否都存在正确连接。利用形式技术检查这种连通性会穷尽所有可能性，发现极端情况并带来自动化功能，而仿真技术常常无法做到这一点。

存储器 BIST 检查

设计常常会包含由内建自测试 (BIST) 逻辑测试的存储器，其中 BIST 逻辑是在 RTL 阶段插入。可能是许多存储器（常常位于不同层级）连接到单个主 BIST 控制器。

来自 BIST 控制器的控制信号连接到各种存储器或存储器控制器，这些连接可以是共用的。例如，来自 BIST 控制器的 `write _ enable` 可以连接到许多存储器上的 `write _ enable` 管脚。

形式连通性检查是一种有效替代方法，用户无需编写动态测试来检查存储器 BIST 连接并在每次更改 RTL 时重新运行测试。

此外，形式检查还能确保这些存储器／MBIST 连接上没有放置时序逻辑，这常常是一个设计要求。

JTAG 检查

与 MBIST 检查类似，设计人员可以在设计中添加 JTAG 电路，这常常也在 RTL 阶段进行。JTAG 的潜在用途包括：创建对设计的测试访问，启动全扫描检查，或控制 MBIST 电路。

JTAG 逻辑具有固定的规范和多种标准工作模式。连通性检查可用来确保所有正确的设计元素（例如 MBIST 控制器）都连接到预期的 JTAG 控制寄存器。

在某些 JTAG 模式下，边界扫描寄存器形成一条长链。该链的长度由设计团队确定。在连通性规范中将长度指定为延迟周期数，连通性检查便可确保在特定模式下该链的长度是正确的。

结语

本文提供的信息当然是很粗略的。详细记录哪怕是最基本的 SoC 验证流程，也很容易写上数十页甚至更多。然而，尽管简短，但应该还是有充足的材料来供用户开始制定流程，以实现更有效的连通性检查。我们会继续更新本文，以反映我们在与客户合作中所获得的经验教训。有关形式验证的更多信息，请访问 http://www.mentor.com/products/fv/0-in_fv/。您还可以观看 Abdel 的 “高级形式验证” 在线研讨会，网址：http://www.mentor.com/products/fv/events/advanced-formal-verification。要旨如下：形式验证长期用作基于仿真的验证的一般补充，随着 SoC 复杂度不断提高，形式验证将继续以新的方式得到应用。

如需最新信息，请致电联系我们，或者访问： **www.mentor.com**

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

Mentor® A Siemens Business

MGC 10-20 TECH10150-w-CN