Computer Science and Application 计算机科学与应用, 2022, 12(11), 2442-2450
Published Online November 2022 in Hans. http://www.hanspub.org/journal/csa
https://doi.org/10.12677/csa.2022.1211250

Hans汉斯

基于语义分析的Verilog缺陷自动检测技术研究

汤泽宇，李丽华，赵　静，王　栋

中国航天科工集团第三研究院第304研究所，北京

收稿日期：2022年10月8日；录用日期：2022年11月7日；发布日期：2022年11月14日

摘　要

本文对Verilog的语义级缺陷进行了研究，并提出了一种基于语义缺陷的自动化检测方法。通过对Verilog代码进行转换，得到两种包含不同语义信息的中间表达，并将其运用于语义级缺陷的自动检测方法。笔者们对Verilog设计中几种常见的语义级缺陷进行了说明与分析，并通过两种中间表达，从中提取了各种缺陷的特征。检测方法对转换后中间表达的语义信息进行提取，与缺陷的特征进行比较，从而实现对缺陷的检测。经实际应用，该方法满足了可编程逻辑器件的测试工作需求，提升了静态测试工作的分析效率与准确率。

关键词

可编程逻辑器件，静态测试，缺陷，语义分析

Study on Verilog Defect Auto Detection Technology Based on Semantic Analysis

Zeyu Tang, Lihua Li, Jing Zhao, Dong Wang

The No. 304 Research Institute, Third Academy, China Aerospace Science & Industry Corp., Beijing

Received: Oct. 8th, 2022; accepted: Nov. 7th, 2022; published: Nov. 14th, 2022

Abstract

This article carries out research on semantic-level defects of Verilog, and presents a semantic-based defect auto-analysis method. Two Intermediate Expressions (IRs) which contain different semantic-level information can be generated by converting Verilog code. The authors demonstrate and analyze several common defects in Verilog designs, and extract the characteristics of various de-

文章引用: 汤泽宇, 李丽华, 赵静, 王栋. 基于语义分析的Verilog缺陷自动检测技术研究[J]. 计算机科学与应用, 2022, 12(11): 2442-2450. DOI: 10.12677/csa.2022.1211250

fects through two Intermediate Expressions. The detection method extracts the semantic information of the intermediate expression after transformation and compares it with the feature of the defect, so as to realize the detection of the defect. Through practical application, this method can satisfy the requirement of programmable logic device testing and improve the analysis efficiency and accuracy of static testing.

Keywords

Programmable Logic Device, Static Testing, Defect, Semantic Analysis

1. 引言

随着可编程逻辑器件的技术越来越成熟，在军事、国防、航空、航天等领域的应用也越来越广泛，尤其在控制、数据处理和存储、仪器仪表、通信和数字信号处理等方面。可编程逻辑器件软件具有代码紧凑、可靠、对实时性有严格要求等特点，传统的检测手段主要包括动态测试和静态测试。动态测试主要依靠仿真测试手段对设计的功能、性能、接口等正确性和对需求的符合性进行检测，很难做到代码级检测；而静态测试主要针对代码本身缺陷进行检测，不需要运行代码程序，可对软件代码的规范性、准确性、一致性等特性进行综合检测，因此，静态测试成为检测可编程逻辑器件软件代码缺陷的重要手段。使用自动化测试方法开展可编程逻辑器件软件代码缺陷检测是提高静态测试效率的必要手段。因为硬件描述语言(HDL)非常复杂，为了实现自动化检测，需要借助一种中间表示(IR)形式，在中间表示形式上定义相应的检查规则来实现。

在 Verilog 代码分析测试工作中，许多包含一定特征与规律的语义类缺陷，如异步复位信号未同步释放，输入信号产生毛刺经常被检出，对这样的缺陷进行自动化发现与定位，将能够极大地提高测试工作的效率。然而作为一种常见的 HDL，Verilog 由于其复杂性，不能够作为现代电路设计流程的中间结构。相反，每个设计自动化工具都会转换 HDL 到其自己的中间表示(IR)。这些工具是一个整体并且大部分技术都是专有的，他们在 HDL 的实现方面存在分歧。虽然有很多 IR，但还不存在一种 IR 可以用于整个电路设计流程。且这些 IR 的结构与使用往往是商业机密，难以基于它们进行进一步开发。本文提出一种 HDL 的语义级代码缺陷检测方法，对 Verilog HDL 代码进行结构转换，使用抽象语法树表达与底层硬件描述表达(LLHD)两种 IR，能够从两种 IR 中提取语义结构信息，检测语义级 HDL 代码缺陷，并支持语义级代码缺陷检测规则的定制。

2. 当前缺陷检测方法的发展

对 Verilog 进行语义分析的问题早在 1995 年就被提出[1]。将 Verilog 翻译到部分中间语言，并进行语义分析的努力始终存在，并已经可以实现类型检查、一致性检查、信号路径追踪等[2] [3] [4]；同时，随着近些年机器学习的大发展，涌现出了一批新的解决方案，如通过支持向量机、粒子群算法来检测硬件木马[5]。但是这些工作要么还不能解决工作中更复杂的问题，要么太过于专门且复杂，测试人员只能检测有限的缺陷，使用随软件提供的规则，且难以自己进行扩展，对未包含在内的许多缺陷都需要依赖软件供应商未来的更新才能支持。这些方法都不便于在 Verilog 设计的静态测试工作中使用，需要提出一

种更易于扩展的新方法。

3. 测试方法总体概述

如下图 1，本文所提出的静态测试方法主要包括两步：1) 将 Verilog 原始代码转换为标记了语义信息的，适合检测的两种 IR——CST 表达与 LLHD 表达(见第 3 节)。2) 运用缺陷检测规则，匹配 IR 中符合缺陷特征的语义信息(见第 4，5，6 节)。下文将详述检测方法中各步骤。

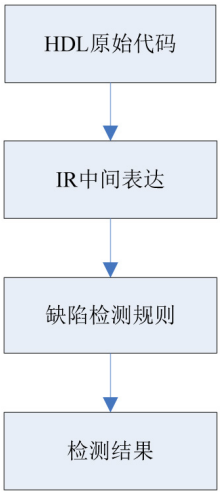

Figure 1. Flow chart of semantic defect auto detection
图 1. 语义缺陷自动检测流程图

4. Verilog 代码的转换

为了分析并提取 HDL 代码中的各种语义级缺陷，我们将 Verilog 代码转换为两种表达，CST 表达与 LLHD 表达。转换得到的这两种表达各自包含了对 Verilog HDL 进行分析的来的语义信息，从而为进一步的检测提供支撑。

4.1. CST 表达

CST 表达是对 Verilog 代码逐词地进行转换，并标记一些 Verilog 语言的保留字与结构，如 begin，@开始的敏感列表等等。如示例 Verilog 代码

always @(posedge clk_in or negedge rst_n) begin

会被转换为图 2、图 3 所示的 CST 表达。

会被转换为如下的 CST 表达

```
637   “children”: [
638         {
639               “end”: 456, //单词的结束位置，下同
640               “start”: 450, //单词的起始位置，下同
641               “tag”: “always” //单词的字面值，此处为 always
642         },
643         {
644                “children”: [
```

```
645                 {
646                     “children”: [
…
760                     “tag”: “kEventControl” //标记事件控制，即敏感列表的开始
761                 },
762                 {
763                     "children": [
764                     {
765                       "children": [
766                         {
767                           "end": 497,
768                           "start": 492,
769                           "tag": "begin"
770                         }
771                       ],
772                       "tag": "kBegin" //标记代码块的开始
773                     },
```

如上述代码所示，此处 always 敏感列表语句被 637~773 行的 JSON 数组的一部分所表示，以 JSON 数组标签“tag”表达原始的 Verilog 代码内容，“end”与“start”表达该词在 Verilog 代码中的位置，并可见 765~771 行描述的 begin 关键字的 JSON 元素整体被 772 行的标签“tag: "kBegin"”标记，表明其是一个 Verilog 代码块的开始。

从 644 行开始，字符“@”开始的敏感列表事件描述被 760 行标记为“tag: "kEventControl"”，下述代码将展示该 JSON 数组节点的部分内容，是一系列 JSON 数组的元素，依序分别表示了左括号、关键词 postage、信号 clk_in 等。整个 Verilog 代码文件将被组装为一个庞大的 JSON 数组，其中，层层嵌套了每一个节点。

```
653   "children": [
654         {
655           "end": 458,
656           "start": 457,
657           "tag": "@"
658         },
659         {
660           "children": [
661               {
662                "children": [
663                   {
664                     "end": 466,
665                     "start": 459,
666                     "tag": "posedge" //上升沿
667                   },
```

```
668                 {
669                    "children": [
670                       {
671                          "children": [
672                             {
673                                "children": [
674                                   {
675                                      "children": [
676                                         {
677                                            "children": [
678                                               {
679                                                  "end": 473,
680                                                  "start": 467,
681                                                  "tag": "SymbolIdentifier", //标记单词表示信号的名称
682                                                  "text": "clk_in" //单词字面值，这里是信号名
683                                               }
…
```

4.2. LLHD 表达

LLHD 为 HDL 和工具的创新建立了一个创新基础，无需构建冗余编译器或以及支持各种不相交的 IR。LLHD 模拟器的运行速度比商业模拟器快 2.4 倍，产生等效的、周期准确的结果[6]。

LLHD 表达将 HDL 分为控制流与数据流并分开表述。下述 LLHD 代码展示了同一个 Verilog 设计对应的 LLHD 表达的数据流，其中，表示了设计中的内部变量、各控制流模块、组合逻辑直接赋值、以及该模块中所包含的控制流等信息，接下来取第 120 行所描述的控制流(其正对应着 4.1 节示例中的代码)，加以说明 LLHD 表达控制流的方式。

```
97    u2: entity @FPGA_DEFECT_4_1 (i1$ %clk_in, i1$ %rst_n, i1$ %en_i) -> (i1$ %en_o) {
98 end:
99       %v4 = const i8 0 //%v[数字]开头的都是内部创建的中间变量，下同；const 表示常数；i8 表示 8 位整数
100         %count = sig i8$ %v4
101         %v6 = const i1 0
102         %rst_reg1 = sig i1$ %v6
103         %rst_reg2 = sig i1$ %v6
104         %en_i.prb = prb i1 %en_i    //表示读取信号 en_i 的值，赋给变量 en_i.prb
105         %count.prb = prb i8 %count
106         %v11 = exts i1 %count.prb, 7, 1
107         %v12 = const i1 0
108         %v13 = neq i1 %en_i.prb, %v12
109         %v14 = const i1 0
110         %v15 = neq i1 %v11, %v14
```

```
111         %v16 = and i1 %v13, %v15
112         %rst_n.prb = prb i1 %rst_n
113         %v18 = const i1 0
114         %v19 = neq i1 %v16, %v18
115         %v20 = const i1 0
116         %v21 = neq i1 %rst_n.prb, %v20
117         %v22 = and i1 %v19, %v21
118         %v23 = const time 0s 1d
119         drv %en_o, %v22, %v23
120         inst %FPGA_DEFECT_4_1.always.0 (%clk_in, %rst_n) -> (%rst_reg1, %rst_reg2) //表示一个控制流，其输入为信号 clk_in 与 rst_n，输出为信号 rst_reg1 与 rst_reg2
121             inst %FPGA_DEFECT_4_1.always.1 (%clk_in, %rst_reg2) -> (%count)
122     }
```

接下来的代码块将展示前述代码行“always @(posedge clk_in or negedge rst_n) begin”及之后代码块(一个异步复位信号同步块)转换为 LLHD 表达后的控制流与数据流。观察代码中第 3 行可见，LLHD 表达以人类程序员易于理解的，近似函数的形式表述了该代码块的输入输出，并将该 always 代码块分割为初始化 init、判断条件 ckeck、根据判断结果进入的各分支 event、if_true、if_false 等等。LLHD 表达还对 Verilog 的指令进行了分类标记，形成了一套人类能理解的，类似 Verilog 关键字的语法结构，来表达电路的行为，如分别对应下述代码块中 13、22~24、25 行的读取信号值，逻辑运算，根据条件判断执行分支等操作。

```
3 u0: proc %FPGA_DEFECT_4_1.always.0 (i1$ %clk_in, i1$ %rst_n) -> (i1$ %rst_reg1, i1$ %rst_reg2) {
4 bb0: //一个基本控制块，其中没有更深的控制流模块，保证控制流从该块的顶部进入，从底部退出
5       %v4 = prb i1 %rst_reg1
6       %rst_reg1.shadow = var i1* %v4      //表示一个指向 1 位整数型变量的指针
7       br init
8   init: //表示读取控制流的输入信号
9       %clk_in.prb = prb i1 %clk_in
10      %rst_n.prb = prb i1 %rst_n
11      wait %clk_in, %rst_n, check     //表示一个名为 check 的块，其行为是等待信号 clk_in，rst_n 的变化
12      check: //本块的行为是判断上述 2 个信号的变化是否符合要求的边沿，并执行下一步
13      %v8 = prb i1 %rst_reg1
14      st %rst_reg1.shadow, %v8
15      %clk_in.prb1 = prb i1 %clk_in
16      %v10 = const i1 0
17      %v11 = eq i1 %clk_in.prb, %v10
18      %v12 = neq i1 %clk_in.prb1, %v10
19      %posedge = and i1 %v11, %v12
20      %rst_n.prb1 = prb i1 %rst_n
21      %v15 = const i1 0
```

```
22      %v16 = neq i1 %rst_n.prb, %v15
23      %v17 = eq i1 %rst_n.prb1, %v15
24      %negedge1 = and i1 %v16, %v17
25      %event_or = or i1 %posedge, %negedge1
26      br %event_or, init, event //表示进行判断，根据 event_or 的值，决定返回 init 块或进入 event 块
27  event:
28      %rst_n.prb2 = prb i1 %rst_n
…
```

5. 自动化缺陷检测静态分析的实现

在第 2、第 3 节所述的两种 IR 的支持下，本文实现了输入 RTL 级 Verilog 代码，自动生成其转换后的 CST 表达与 LLHD 表达，并针对这两种 IR，使用设计的缺陷检测函数进行缺陷特征的提取匹配，从而检测到可能存在的语义级缺陷。

如下图 2、图 3 所示，首先，程序使用开源中间件 Verible，将输入的 Verilog 原始代码转换为 CST 表达的形式。之后，程序可以将 CST 表达翻译为 LLHD 表达。最后，我们将 CST 表达与 LLHD 表达传入具体缺陷的检测函数，检测函数将根据具体缺陷的检测算法，提取符合缺陷特征的 CST 表达的语义标签序列或 LLHD 表达的指令中变量与操作，最终得出特定缺陷在 context 中 CST 表达与 LLHD 表达所代表的 Verilog 设计中是否存在。

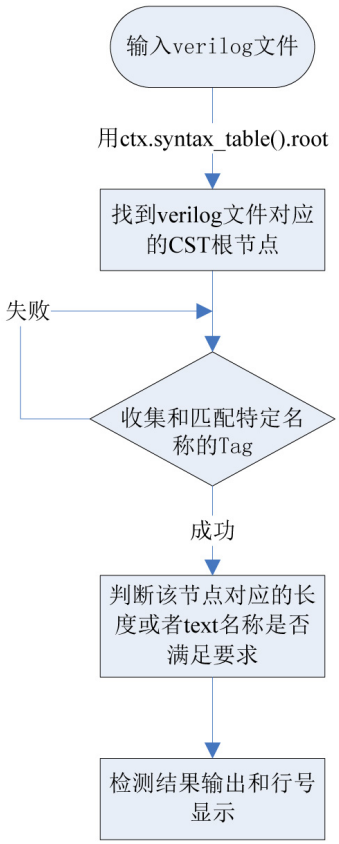

Figure 2. Flow chart of defect detection expressed via CST
图 2. 通过 CST 表达进行缺陷检测的流程图

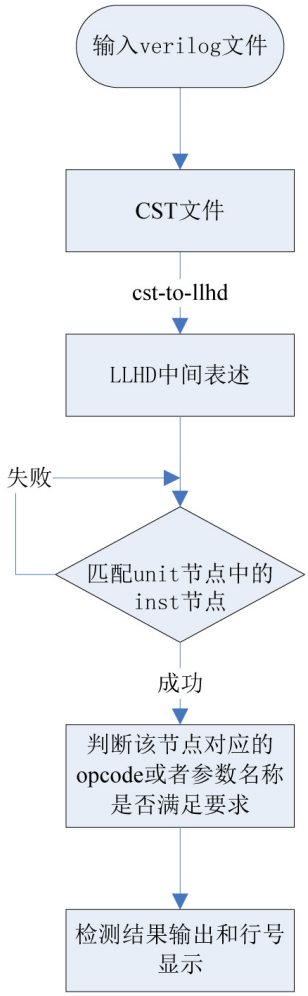

Figure 3. Flow chart of defect detection expressed via LLHD
图 3. 通过 LLHD 表达进行缺陷检测的流程图

6. 语义级缺陷的检测

通过两种方式，测试人员可以分别从两种表达中提取语义级信息。在 CST 表达中，使用设计好的缺陷检测函数，可以提取符合一定特征的标签组合，加以比较并返回结果。根据比较的结果可以判断是否存在函数所描述的缺陷；在 LLHD 表达中，使用设计好的缺陷检测函数，可以提取符合一定特征的指令结构，加以比较并返回结果。根据比较的结果可以判断是否存在函数所描述的缺陷。

本节将列举 3 种语义级缺陷，描述如何使用 CST 表达与 LLHD 表达，提取信息并实现对缺陷的检测。为了体现本文所得成果的实际使用效果，我们编写了一套预埋多种缺陷的 Verilog 用例用作测试，并使用业界常见的 Verilog 代码检查工具 HDL Designer 做对比。

6.1. 缺陷检测－输入信号缺少防毛刺设计

该缺陷的检测算法可以从 LLHD 表达中，提取各代码块对应的控制流中的从信号取样的指令与驱动信号的指令。通过在 LLHD 表达中检测各模块的输入信号，并统计各输入信号在模块中被连续寄存的情况，可以判断输入信号是否进行了防毛刺的设计。如果一个信号被连续取值再输出驱动至少 3 次，则认为该信号进行了足够的防毛刺设计；反之，则认为该信号缺少防毛刺设计。

6.2. 缺陷检测－使用一位逻辑“0”和“1”表示安全关键信息

考虑到可能使用一位逻辑的判断条件不只有安全关键信息，该缺陷的检测算法可以设置认为代表了安全关键信息的信号名。可以从 CST 表达中，提取 if 语句中，条件表达式中该指定信号的字面值。如果发现匹配到的字面值不为 0 或 1，即认为不存在相关缺陷；反之，则认为使用了一位逻辑表示了安全关键信息。

6.3. 缺陷检测－异步复位未同步释放

该缺陷的检测算法可以通过CST表达，提取各语句块中always语句中的上升沿、下降沿与电平信号，从而形成所有可能的复位信号。然后对每一个可能的复位信号，分析是否对该信号进行了从线网读取/驱动线网、从线网读取/写入寄存器，读取寄存器/驱动线网三类操作，或者说读取并释放可能是复位信号的线网变量或寄存器变量的任意一种。如果有至少一种，则认为可能的复位信号均被释放；反之，则认为存在未被释放的复位信号，并提示测试人员注意。

6.4. 缺陷检测结果对比

经过测试，对所使用的用例，HDL Designer 工具不能检出以上 3 类缺陷，而本文实现的程序可以正确地检测并报出所有存在这 3 类问题的用例中问题出现的具体位置(针对输入信号未做防毛刺、异步复位信号未同步释放的问题，会报出有问题信号的名称；一位逻辑的安全关键信息的问题，会报出具体进行了逻辑判断的条件语句)。

7. 结束语

本文着眼于可编程逻辑器件的代码静态分析工作，为了更高效地检测 Verilog 设计中的语义级缺陷，提出了一种转换 Verilog 代码到包含语义信息的 IR 中，并提取语义信息，与语义缺陷的特征进行比对。作者们期望在未来实现更多语义级缺陷的特征提取与匹配，从而提高 Verilog 代码的静态分析工作的分析效率与准确率。

参考文献

[1] Gordon, M. (1995) The Semantic Challenge of Verilog HDL. *Proceedings of Tenth Annual IEEE Symposium on Logic in Computer Science*, San Deigo, CA, 26-29 June 1995, 136-145.

[2] 刘静. Verilog-to-MSVL 序翻译软件的实现[D]: [硕士学位论文]. 西安: 西安电子科技大学, 2014.

[3] 姚化吉. 基于 64 位 Linux 系统的 MSVL 编译器设计开发与测试[D]: [硕士学位论文]. 西安: 西安电子科技大学, 2020.

[4] 张荣, 王丽娟, 于宗光. 基于结构特征的 IP 软核硬件木马检测方法[J]. 电子设计工程, 2020, 28(15): 23-28.

[5] 李莉云. 基于机器学习的门级网表硬件木马检测[D]: [硕士学位论文]. 兰州: 兰州交通大学, 2021.

[6] Schuiki, F., Kurth, A., Grosser. T. and Benini, L. (2020) LLHD: A Multi-Level Intermediate Representation for Hardware Description Languages. *Proceedings of the* 41*st ACM SIGPLAN Conference on Programming Language Design and Implementation*, London, June 2020, 258-271. https://arxiv.org/abs/2004.03494 https://doi.org/10.1145/3385412.3386024