

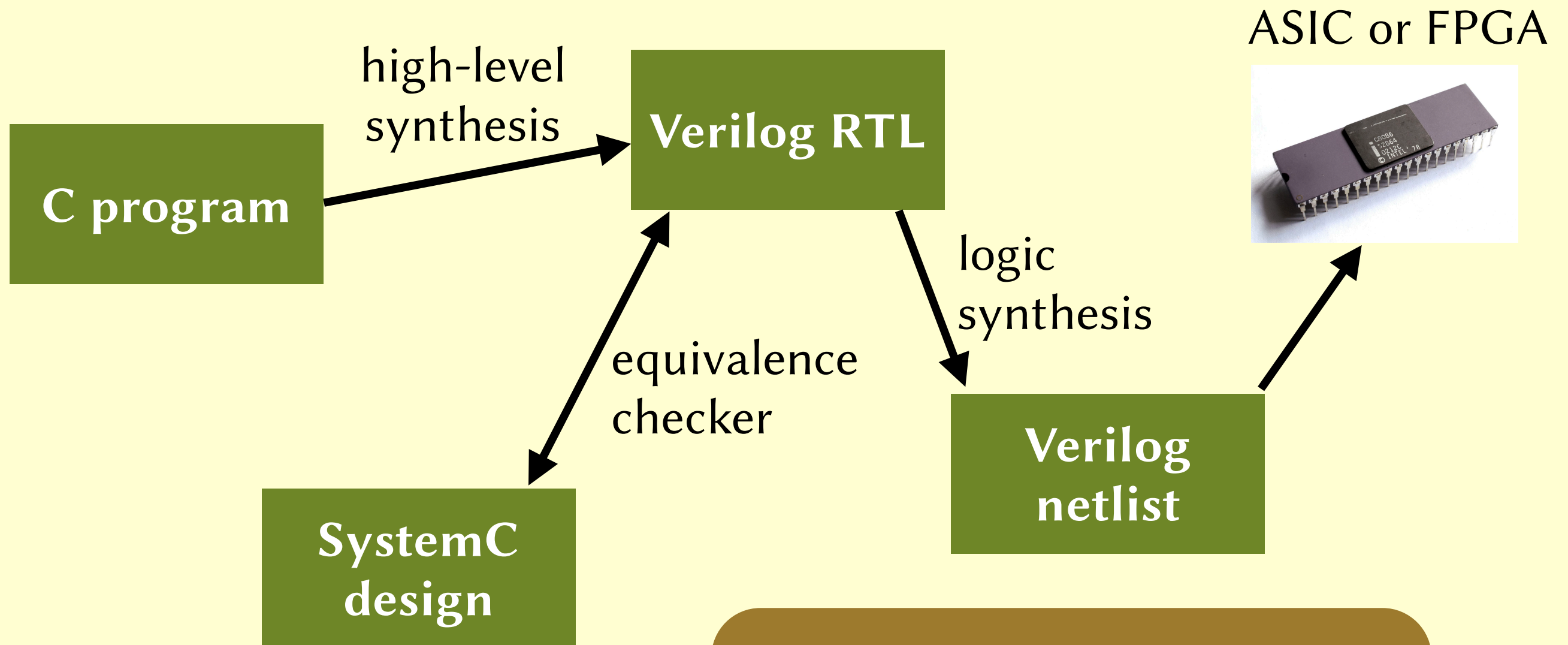
Testing and verifying the tools of hardware design

John Wickerson
Imperial College London

**joint work with Yann Herklotz, Michalis Pardalos, Quentin Corradi,
*George Constantinides, Alastair Donaldson, Emiliano Morini, and Laura Pozzi***

Verification Futures Conference
01 July 2025

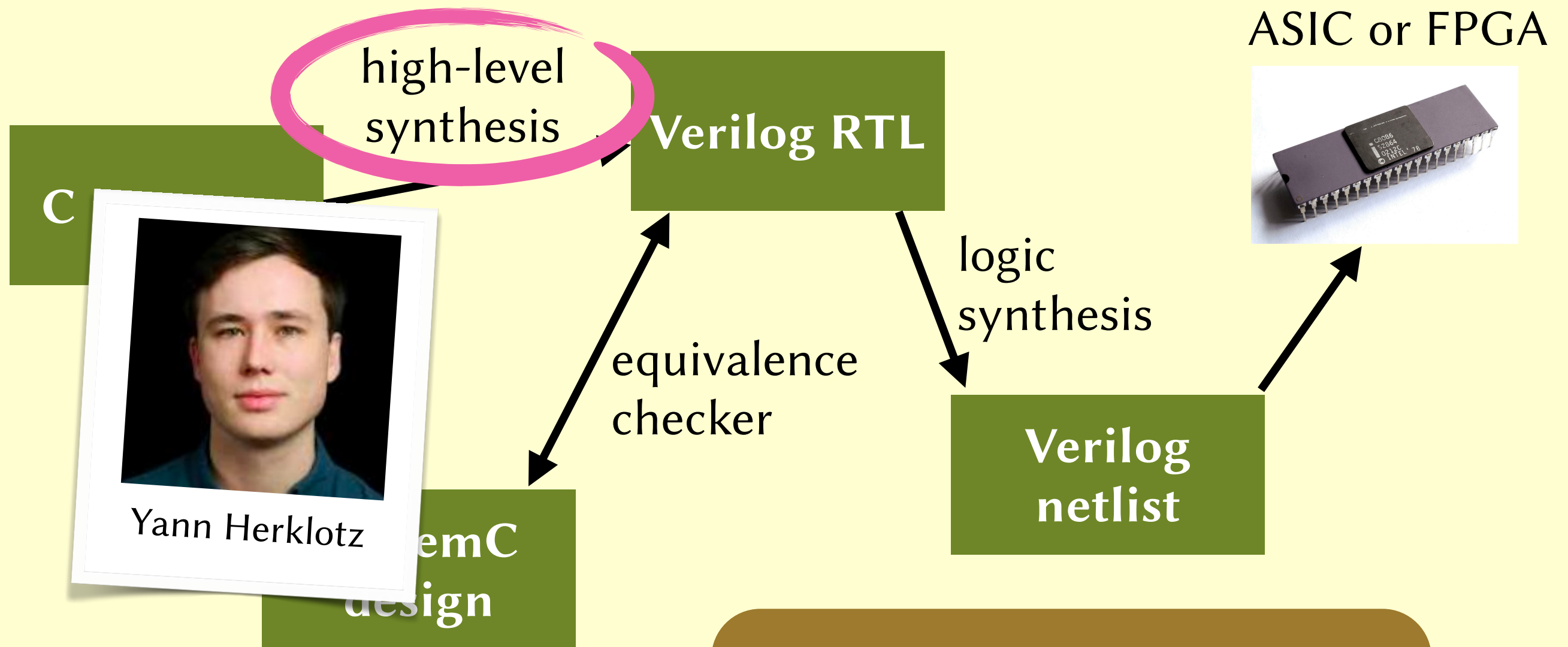
Some EDA tools



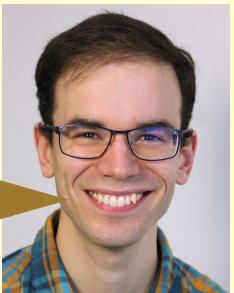
All these tools are too buggy. With more rigorous engineering, they can be made more suitable for safety- and security-critical settings.



Some EDA tools



All these tools are too buggy. With more rigorous engineering, they can be made more suitable for safety- and security-critical settings.

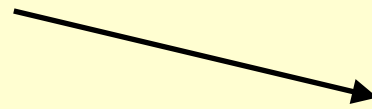


High-level synthesis

```

1  int main() {
2      int x[2] = {3, 6};
3      int i = 1;
4      return x[i];
5  }

```



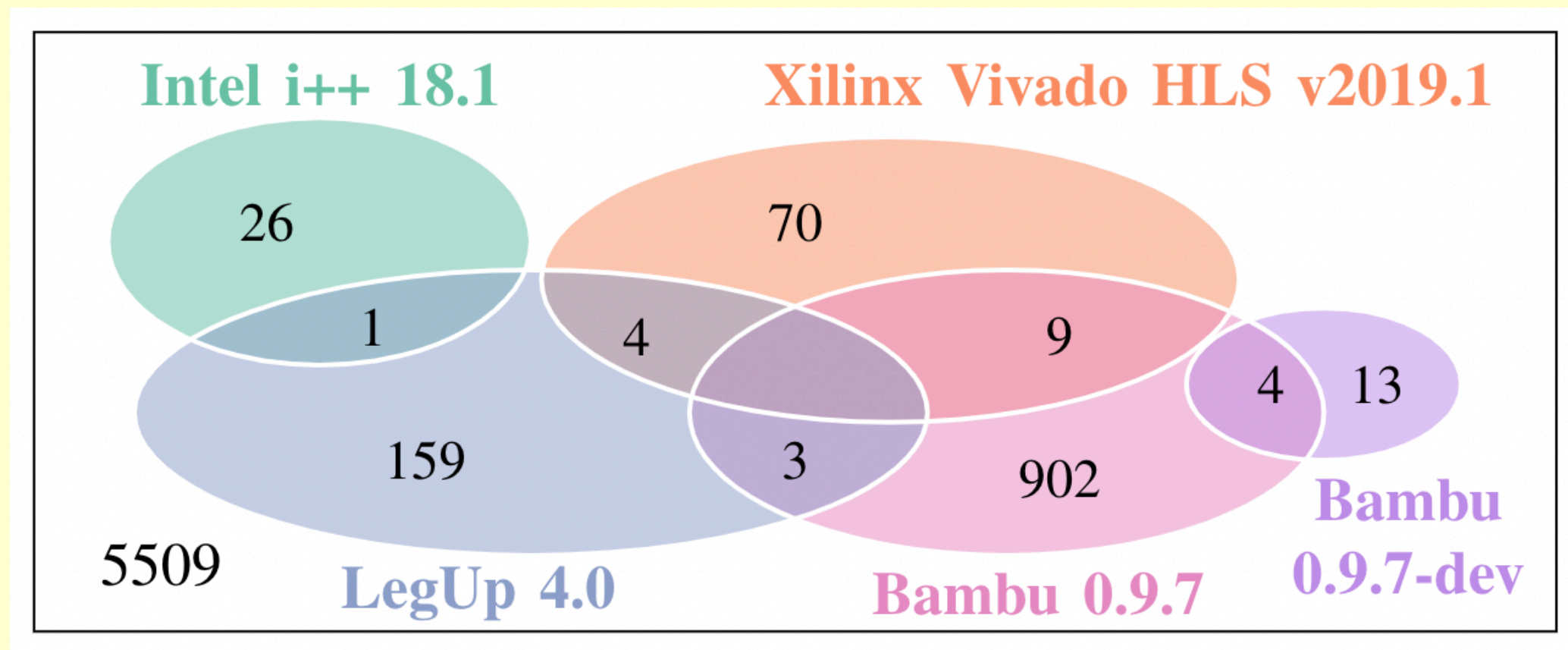
```

1  module main(reset, clk, finish, return_val);
2      input [0:0] reset, clk;
3      output reg [0:0] finish = 0;
4      output reg [31:0] return_val = 0;
5      reg [31:0] reg_3 = 0, addr = 0, d_in = 0, reg_5 = 0, wr_en = 0;
6      reg [0:0] en = 0, u_en = 0;
7      reg [31:0] state = 0, reg_2 = 0, reg_4 = 0, d_out = 0, reg_1 = 0;
8      reg [31:0] stack [1:0];
9      // RAM interface
10     always @(negedge clk)
11         if ({u_en != en}) begin
12             if (wr_en) stack[addr] <= d_in;
13             else d_out <= stack[addr];
14             en <= u_en;
15         end
16     // Data-path
17     always @(posedge clk)
18         case (state)
19             32'd11: reg_2 <= d_out;
20             32'd8: reg_5 <= 32'd3;
21             32'd7: begin u_en <= (~ u_en); wr_en <= 32'd1;
22                     d_in <= reg_5; addr <= 32'd0; end
23             32'd6: reg_4 <= 32'd6;
24             32'd5: begin u_en <= (~ u_en); wr_en <= 32'd1;
25                     d_in <= reg_4; addr <= 32'd1; end
26             32'd4: reg_1 <= 32'd1;
27             32'd3: reg_3 <= 32'd0;
28             32'd2: begin u_en <= (~ u_en); wr_en <= 32'd0;
29                     addr <= {{reg_3 + 32'd0} + {reg_1 * 32'd4}} / 32'd4; end
30             32'd1: begin finish = 32'd1; return_val = reg_2; end
31             default: ;
32         endcase
33     // Control logic
34     always @(posedge clk)
35         if ({reset == 32'd1}) state <= 32'd8;
36         else case (state)
37             32'd11: state <= 32'd1;
38             32'd8: state <= 32'd7;
39             32'd7: state <= 32'd6;
40             32'd6: state <= 32'd5;
41             32'd5: state <= 32'd4;
42             32'd4: state <= 32'd3;
43             32'd3: state <= 32'd2;
44             32'd2: state <= 32'd1;
45             32'd1: ;
46             default: ;
47         endcase
48     endmodule

```


Testing HLS tools

- We generated 6700 random C programs and gave them to four HLS tools (Intel i++, Xilinx Vivado HLS, LegUp, and Bambu).



Some bugs in HLS tools

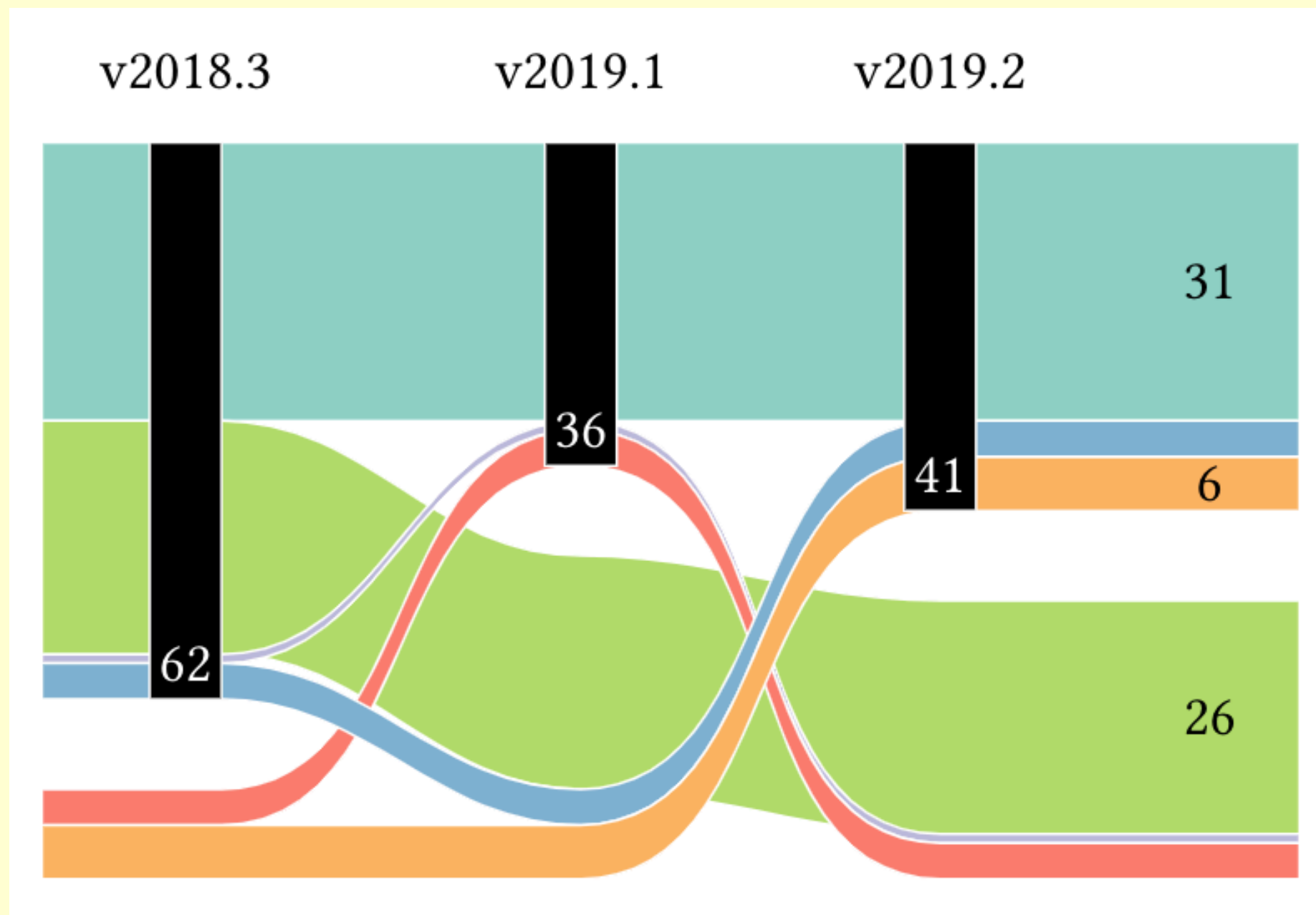
```
1 int a[2][2][1] = {{{0},{1}},{{0},{0}}};  
2  
3 int main() {  
4     a[0][1][0] = 1;  
5 }
```

This code crashes LegUp 4.0

```
1 volatile unsigned int g = 0;  
2 int a[256] = {0};  
3 int c = 0;  
4  
5 void d(char b) {  
6     c = (c & 4095) ^ a[(c ^ b) & 15];  
7 }  
8  
9 void e(long f) {  
10     d(f); d(f >> 8); d(f >> 16); d(f >> 24);  
11     d(f >> 32); d(f >> 40); d(f >> 48);  
12 }  
13  
14 int main() {  
15     for (int i = 0; i < 56; i++)  
16         a[i] = i;  
17     e(g);  
18     e(-2L);  
19     return c;  
20 }
```

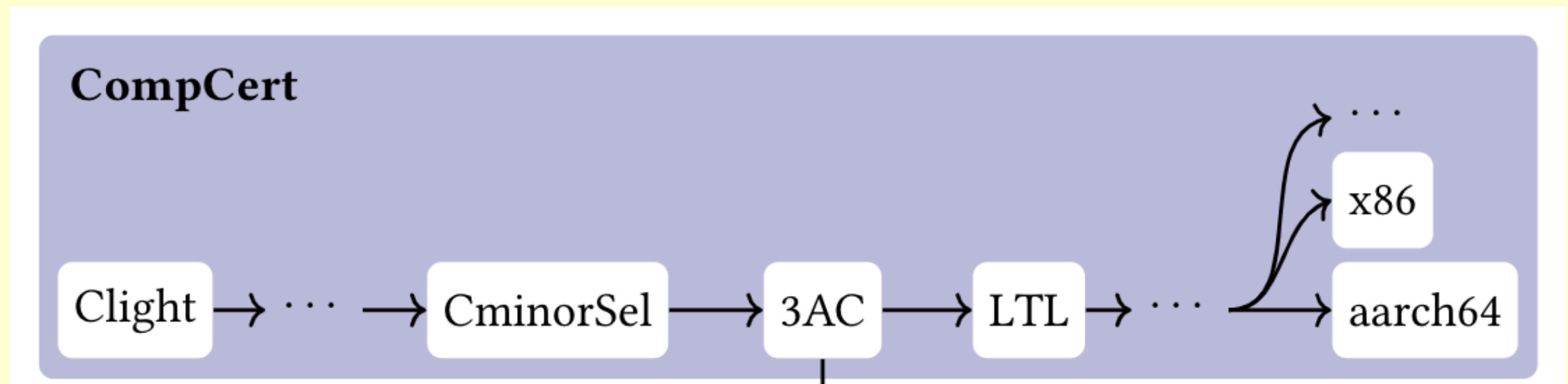
Hardware generated
by Vivado HLS from this code
outputs the wrong value

Aside: bugs over time

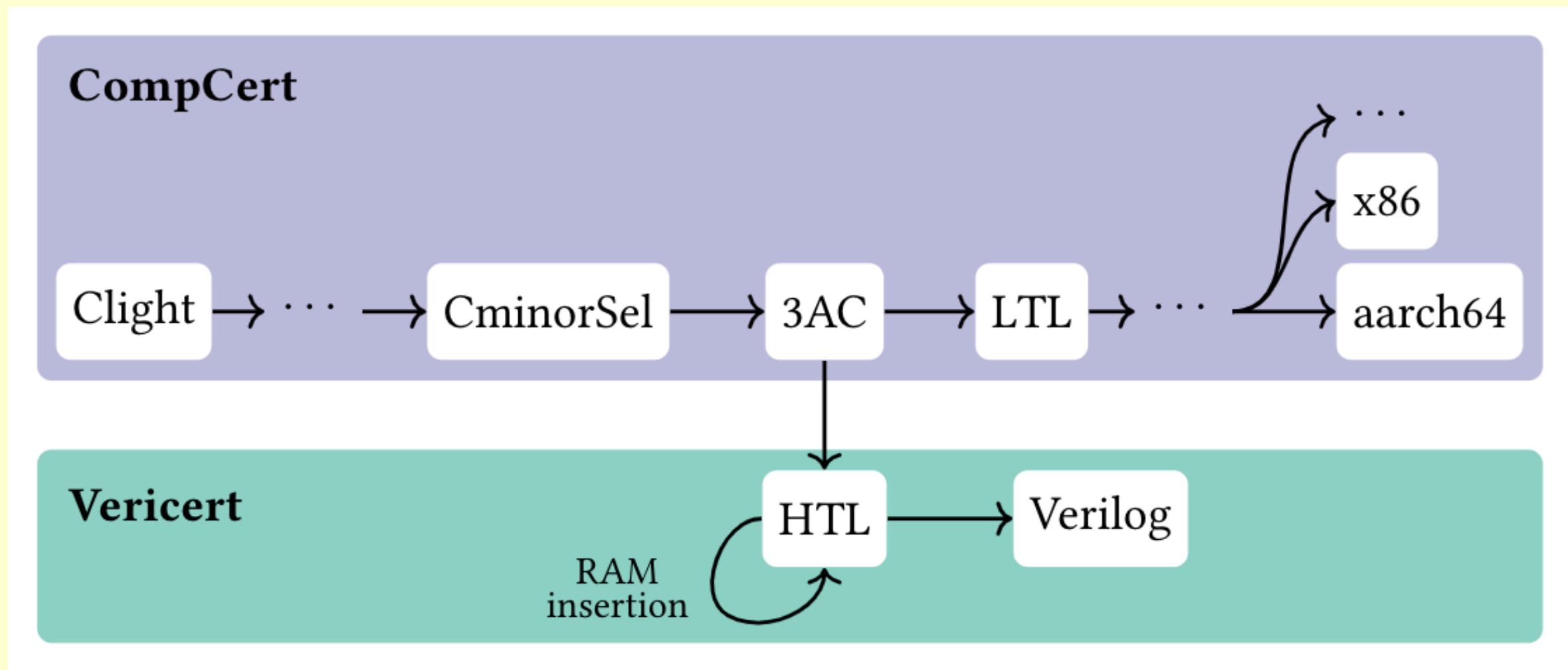


**Can we have more
reliable HLS tools?**

Vericert



Vericert



Correctness

- We prove the following theorem:

$$\forall C, V, B, \quad \text{HLS}(C) = \text{OK}(V) \wedge \text{Safe}(C) \implies (V \Downarrow B \implies C \Downarrow B).$$

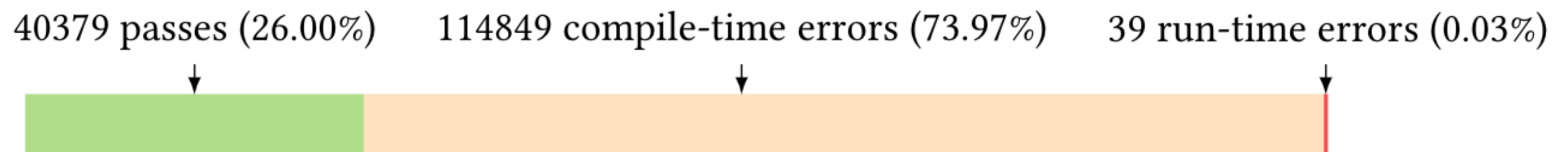
- Roughly:
 - 1.5 person-years of effort,
 - 3k lines of implementation,
 - 8k lines of Coq proof.



Correctness

Beware of bugs in the above code; I have only proved it correct, not tried it.

- From executing 155267 randomly generated C programs:

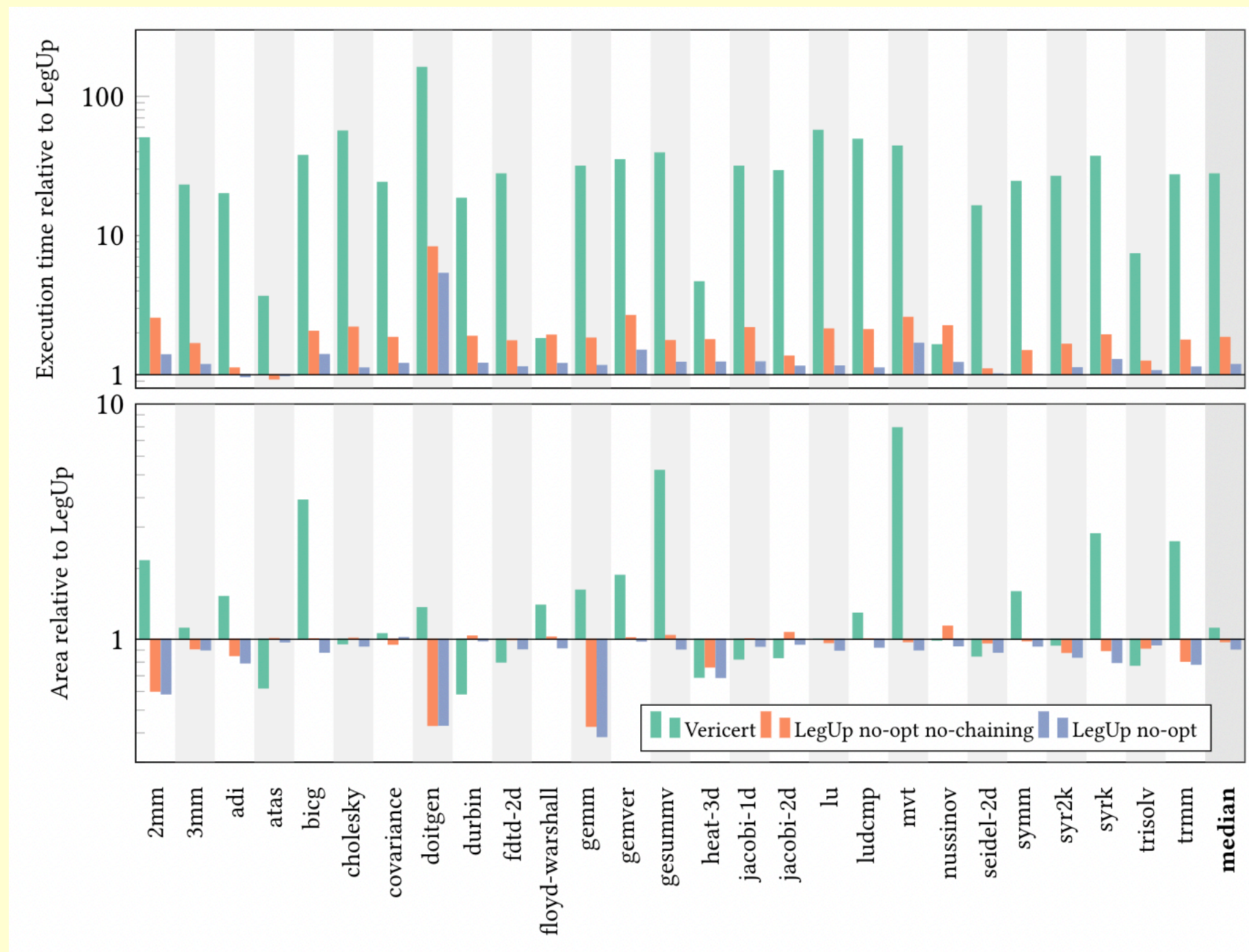


- After fixing the bug in Vericert's pretty-printer, we found 0 run-time errors.

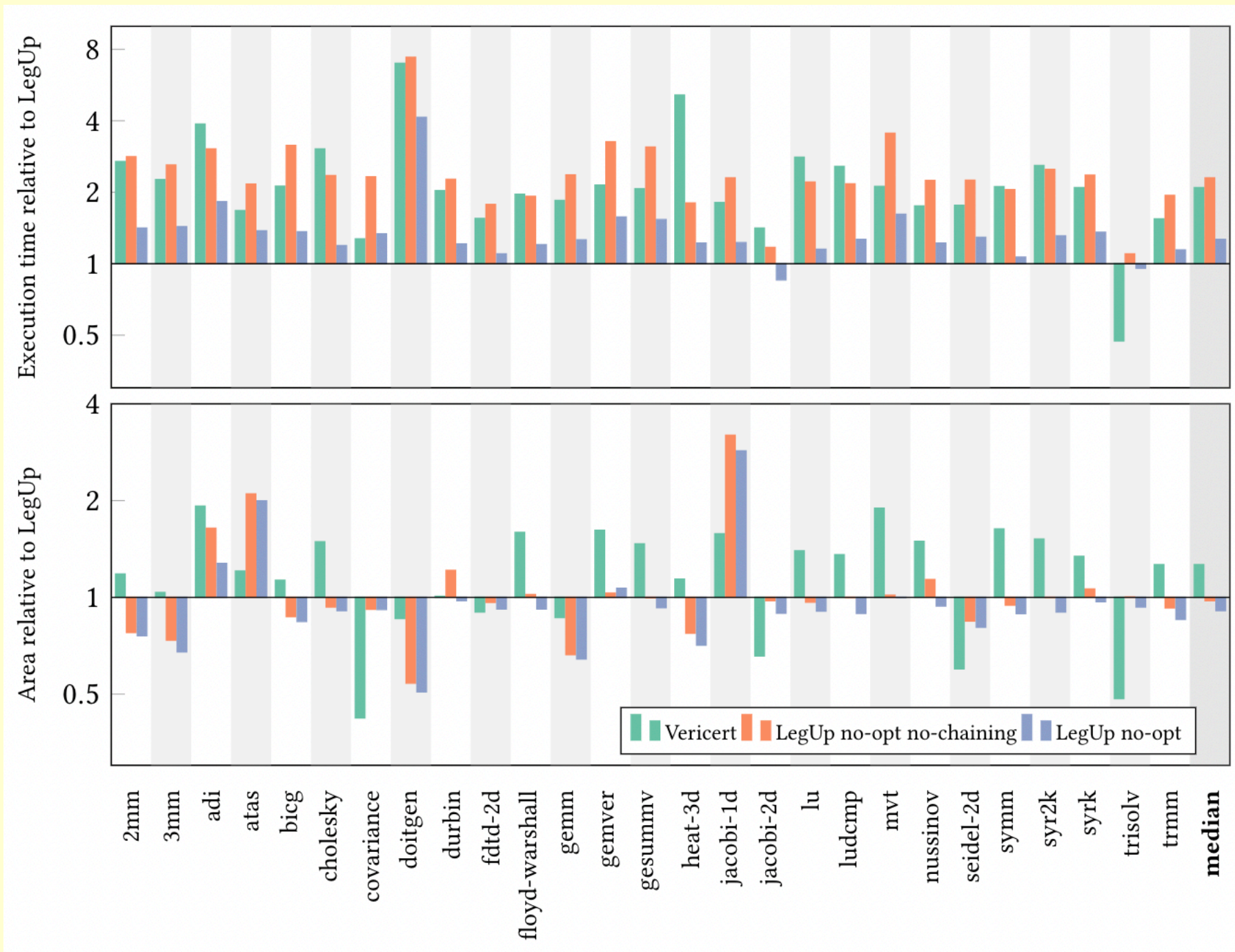
Performance

- Measured performance using the PolyBench/C benchmark.
- 27 of the 30 programs in the benchmark are applicable (the others use floats).
- We synthesised the designs for a Xilinx FPGA and measured their area and running time.
- We compared against an open-source HLS tool called LegUp.

Performance (1st attempt)



Performance



Further reading

- Vericert is fully open-source and available on Github:



<https://github.com/ymherklotz/vericert>

Further reading

OOPSLA '21

Formal Verification of High-Level Synthesis

YANN HERKLOTZ, Imperial College London, UK
JAMES D. POLLARD, Imperial College London, UK
MADESH RAMANATHAN, Imperial College London, UK
Imperial College London, UK

of software into hardware, is rapidly
promises

PLDI '24



Hyperblock Scheduling for Verified High-Level Synthesis

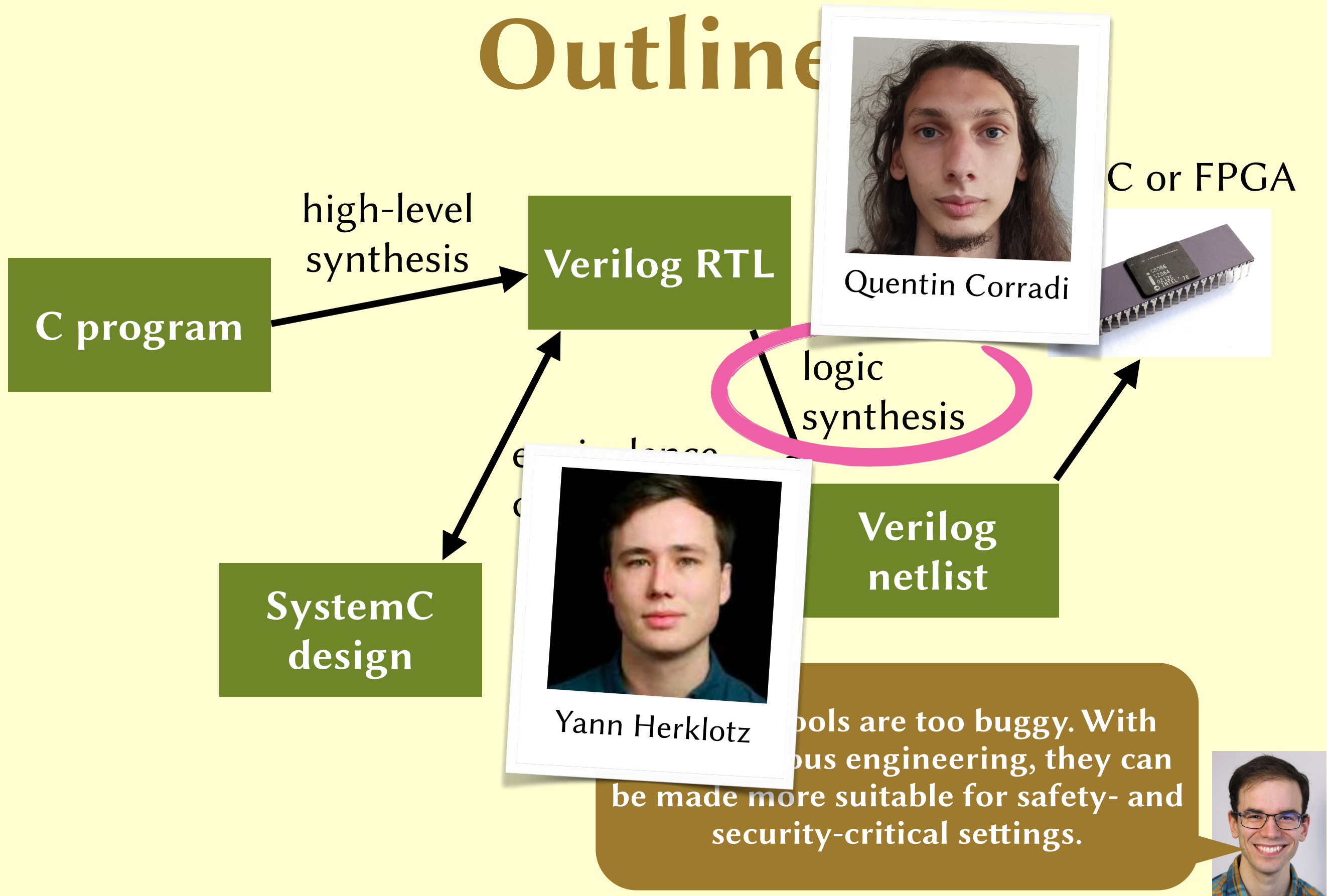
YANN HERKLOTZ, Imperial College London, United Kingdom
JOHN WICKERSON, Imperial College London, United Kingdom

High-level synthesis (HLS) is the automatic compilation of software programs into custom hardware designs. With programmable hardware devices (such as FPGAs) now widespread, HLS is increasingly used. However, existing HLS tools are too unreliable for safety-critical applications, and this paper addresses this.

Where next?

- Operation scheduling
- Resource sharing
- Constant propagation, loop pipelining, ...
- Support more of the C language as input

Outline



Logic synthesis

- We generated about 100,000 random Verilog designs.

```

1 module top #(parameter param0 = 5'h9e23848124)
2 (y, clk, wire0, wire1, wire2, wire3);
3 // *** Declarations ***
4 output wire [(5'h31):(1'h0)] y;
5 input wire [(1'h0):(1'h0)] clk;
6 input wire [(3'h6):(1'h0)] wire0;
7 input wire [(4'ha):(1'h0)] wire1;
8 input wire signed [(4'ha):(1'h0)] wire2;
9 input wire [(4'hb):(1'h0)] wire3;
10 reg [(3'h2):(1'h0)] reg20 = (1'h0);
11 reg [(3'h5):(1'h0)] reg19 = (1'h0);
12 reg [(3'h4):(1'h0)] reg18 = (1'h0);
13 reg [(2'h2):(1'h0)] reg17 = (1'h0);
14 reg [(4'ha):(1'h0)] reg16 = (1'h0);
15 reg signed [(4'h9):(1'h0)] reg15 = (1'h0);
16 wire [(3'h6):(1'h0)] wire5;
17 wire [(2'h3):(1'h0)] wire4;
18 // *** Assign output ***
19 assign y =
20 {reg20,reg19,reg18,reg17,reg16,reg15,wire5,wire4};
21 // *** Random module items ***
22 assign wire4 = (((~wire1) ? (((15'h9ecc51592fdeb04)
23 ? reg17[(5'h2):(2'h2)] : (reg18 ? wire2 : wire0))
24 ? $unsigned((-2'ha73a956341f45c0) << reg18)) :
25 wire1[(4'ha):(3'h7)]) - reg18) :
26 reg15[(4'h9):(3'h7)]) >>> $unsigned($signed((
27 reg16[(4'ha):(3'h7)] ? ((wire1 && reg16) &&
28 {reg15, reg15, wire3}) : (reg18 ? (~&wire3) :
29 (-39'ha7a1419cd4ea34a))))));
30 assign wire5 = $signed(((wire2 ? (
31 (-8'h5e411249da4f335) ? (4'hb2fa97daae9ff) :
32 wire1) : (wire4 ? wire2 : wire1)) ?
33 $signed(wire3) : ({(7'hbac46141008d14)} >>>
34 (&wire0))));
35 always @(posedge clk) begin
36   for (reg15 = (1'h0); (reg15 < (2'h2)); reg15 =
37     (reg15 + (1'h1))) begin
38     if (((wire3 == (~reg16 + wire1))) >=
39       {$signed(wire0[(2'h2):(1'h0)]}))
40       reg16 <= ($unsigned($unsigned(wire1)) <
41         wire3[(1'h1):(1'h1)]);
42     else reg16 <= $unsigned(reg17[(2'h2):(2'h0)]);
43     reg17 <= wire3[(1'h0):(1'h0)];
44   end
45   reg18 <= $signed(({wire0} ^ wire3));
46 end
47 always @(posedge clk) begin
48   if (wire3[(4'h9):(3'h6)])
49     reg19 = $signed($unsigned(wire1)) <<
50     $unsigned({wire1});
51   reg20 <= ({((~|wire3), $unsigned(reg19)) ?
52     reg16 : reg15[(2'h2):(1'h1)]),
53     (~&((wire0 ? wire3 : reg17) ^ reg18))
54     || ((~&(wire3[(4'hb):(4'h9)] ? wire4 : (+wire5))));
55 end
56 endmodule

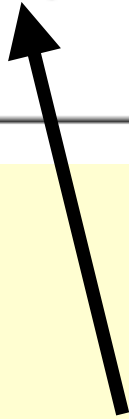
```

Logic synthesis - results

Tool	Total test cases	Failing test cases	Distinct failing test cases	Bug reports
Yosys 0.8	26400	7164 (27.1%)	≥ 1	0
Yosys 3333e00	51000	7224 (14.2%)	≥ 4	3
Yosys 70d0f38 (crash)	11	1 (9.09%)	≥ 1	1
Yosys 0.9	26400	611 (2.31%)	≥ 1	1
Vivado 18.2	47992	1134 (2.36%)	≥ 5	3
Vivado 18.2 (crash)	47992	566 (1.18%)	5	2
XST 14.7	47992	539 (1.12%)	≥ 2	0
Quartus Prime 19.2	80300	0 (0%)	0	0
Quartus Prime Lite 19.1	43	17 (39.5%)	1	0
Quartus Prime Lite 19.1 (No \$signed)	137	0 (0%)	0	0

Logic synthesis - example

```
1 module top (y, clk, w1);  
2     output y;  
3     input clk;  
4     input signed [1:0] w1;  
5     reg r1 = 1'b0;  
6     assign y = r1;  
7     always @(posedge clk)  
8         if ({-1'b1 == w1}) r1 <= 1'b1;  
9 endmodule
```



Vivado incorrectly expands `1'b1` to `2'b11` (should be `2'b01`)

Further reading

FPGA '20

Finding and Understanding Bugs in FPGA Synthesis Tools

Yann Herklotz
yann.herklotz15@imperial.ac.uk
Imperial College London
London, UK

John Wickerson
j.wickerson@imperial.ac.uk
Imperial College London
London, UK

ABSTRACT

All software ultimately relies on hardware functioning correctly. Hardware correctness is becoming increasingly important due to the growing use of custom accelerators using FPGAs to speed up applications on servers. Furthermore, the increasing complexity of hardware also leads to ever more reliance on automation, meaning that the correctness of synthesis tools is vital for the reliability of the hardware.

This paper aims to improve the quality of FPGA synthesis tools by introducing a method to test them automatically using randomly generated, correct Verilog, and checking that the synthesised netlist is always equivalent to the original design. The main contributions of this work are twofold: firstly a method for generating random behavioural Verilog free of undefined values, and secondly a Verilog test case reducer used to locate the cause of the bug that was found. These are implemented in a tool called Verismith. This paper also provides a qualitative and quantitative analysis of the bugs found in Yosys, Vivado, XST and Quartus Prime. Every synthesis tool except Quartus Prime was found to introduce discrepancies between the synthesised netlist and the original design. In addition to that, Vivado and a development

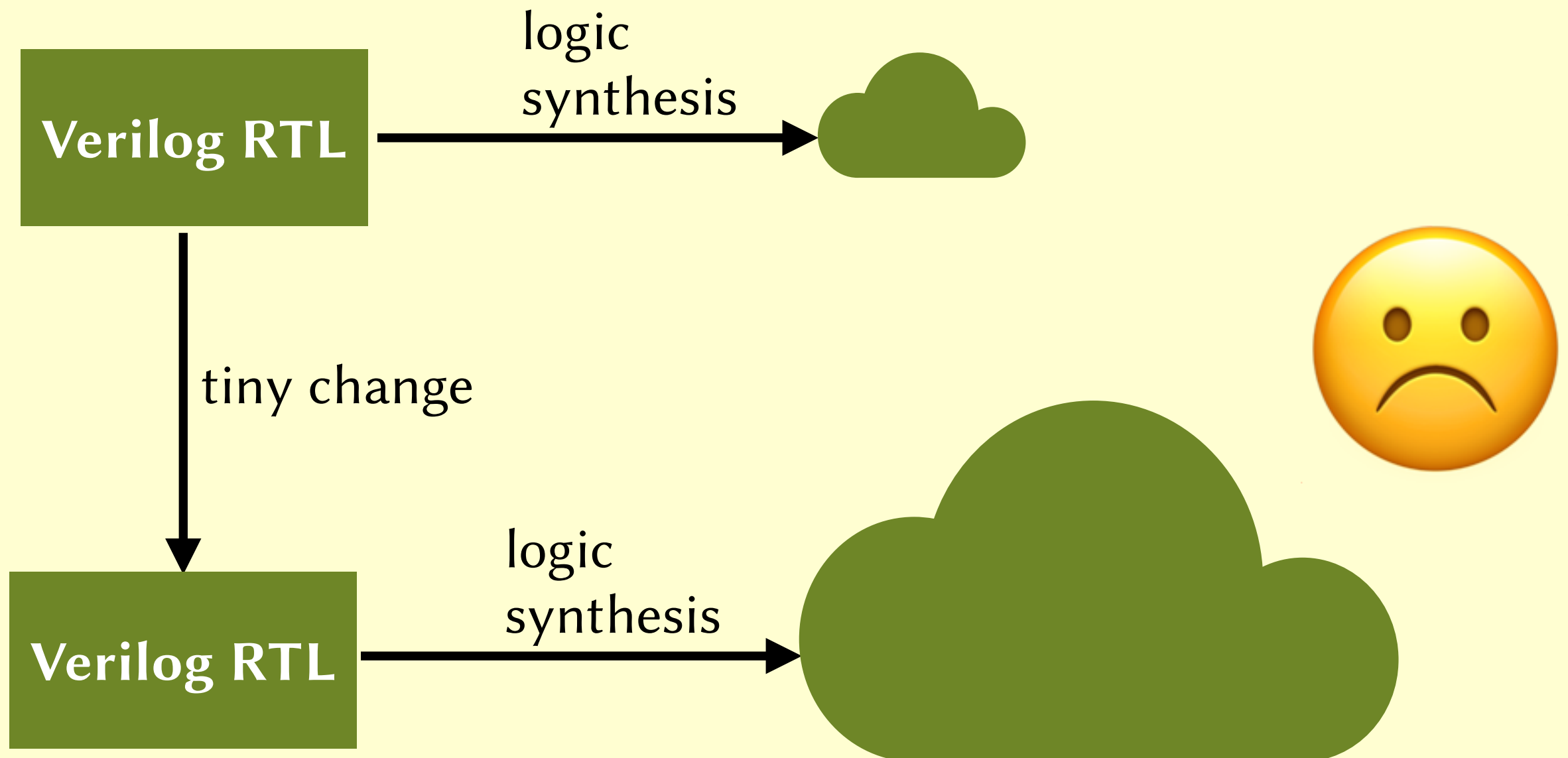
```
1 module top (y, clk, w1);  
2   output y;  
3   input clk;  
4   input signed [1:0] w1;  
5   reg r1 = 1'b0;  
6   assign y = r1;  
7   always @(posedge clk)  
8     if ({-1'b1 == w1}) r1 <= 1'b1;  
9 endmodule
```

Figure 1: Vivado bug found automatically by Verismith. Vivado incorrectly expands `-1'b1` to `-2'b11` instead of `-2'b01`. The bug was reported and confirmed by Xilinx.¹

1 INTRODUCTION

Almost all digital computation performed in the world today relies, in one way or another, on a logic synthesis tool. Computation specified in RTL passes through a logic synthesis tool before being implemented on an FPGA or an ASIC. Even designs that are expressed in higher-level languages eventually get synthesised down to a hardware description language that is executed in software is carried out on

Stability



Stability bug

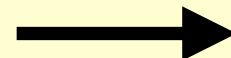
```
95  wire foo;  
96  some;  
97  other;  
98  stuff;  
99  here;  
100 wire baz;
```



```
95  wire foo;  
96  some;  
97  other; stuff;  
98  here;  
99  wire baz;
```



```
wire w95_foo;  
wire w100_baz;
```

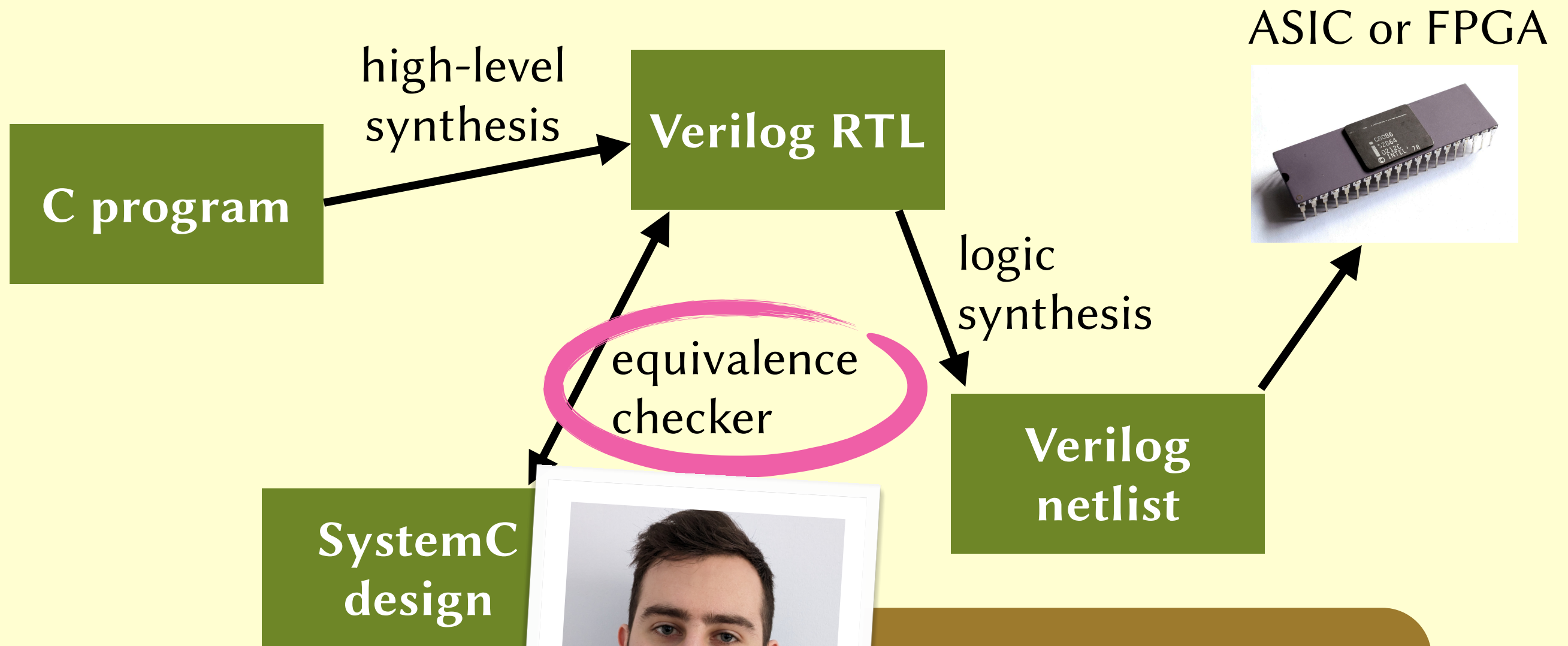


```
wire w95_foo;  
wire w99_baz;
```


**Can we have more
reliable logic synthesis
tools?**

ecosystem for software development, we find tools for the following development methodology: (i) prove a correctness theorem about your program at the source level, (ii) use a verified compiler to transform your program to machine code, and, lastly, (iii) transport the source-level program correctness theorem down to the generated machine code by composing the source-level program correctness theorem with the compiler correctness theorem. When carried out inside an ITP, the development methodology is capable of producing artifacts with remarkably small trusted computing bases (TCBs) [20]. For example, the verified CakeML compiler [35] and its accompanying formal methods tools hosts such a development methodology inside the ITP HOL4 [32]. To put trust in the compiler, we need to trust the ITP.

Outline

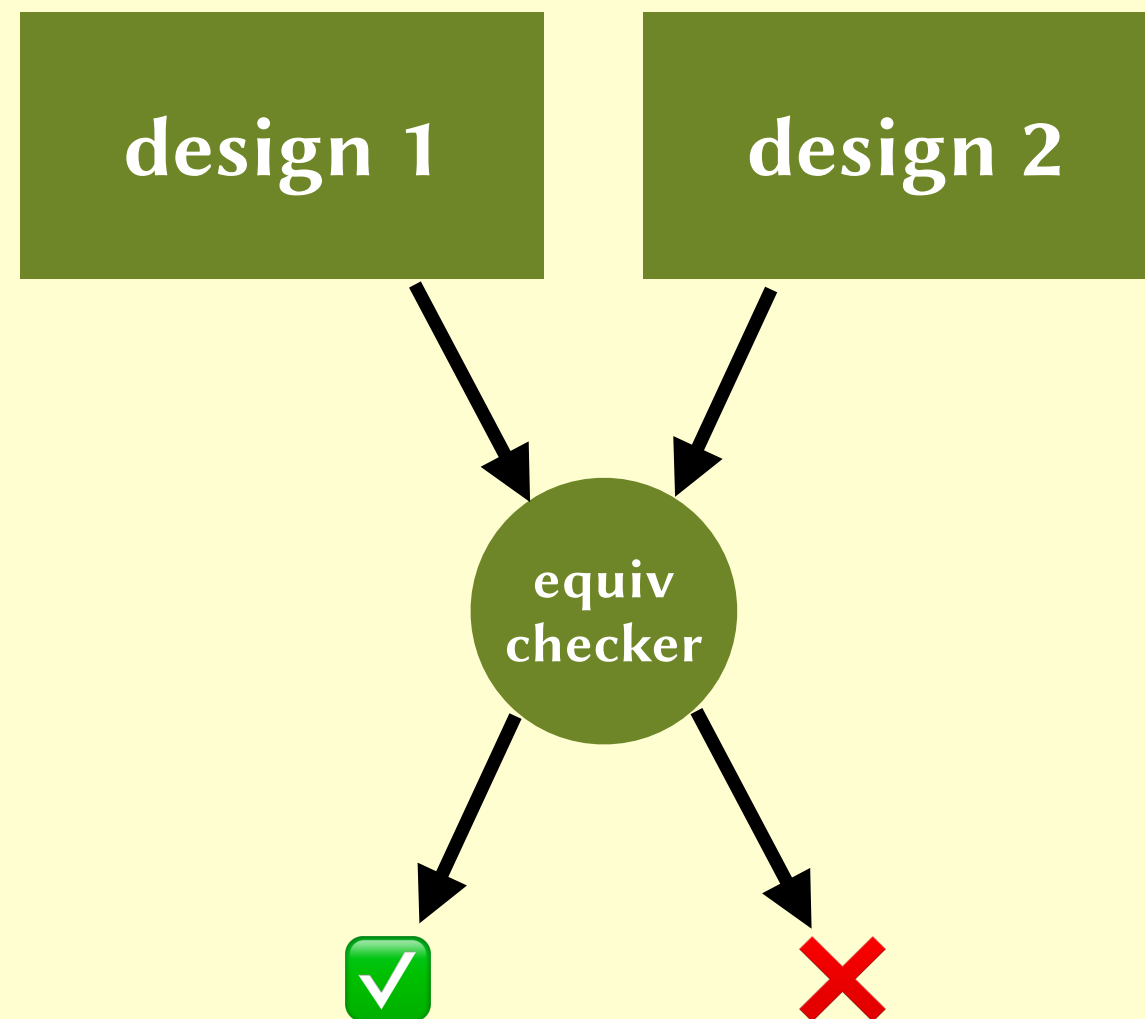


Michalis Pardalos

These tools are too buggy. With rigorous engineering, they can be more suitable for safety- and security-critical settings.



Testing equivalence checkers



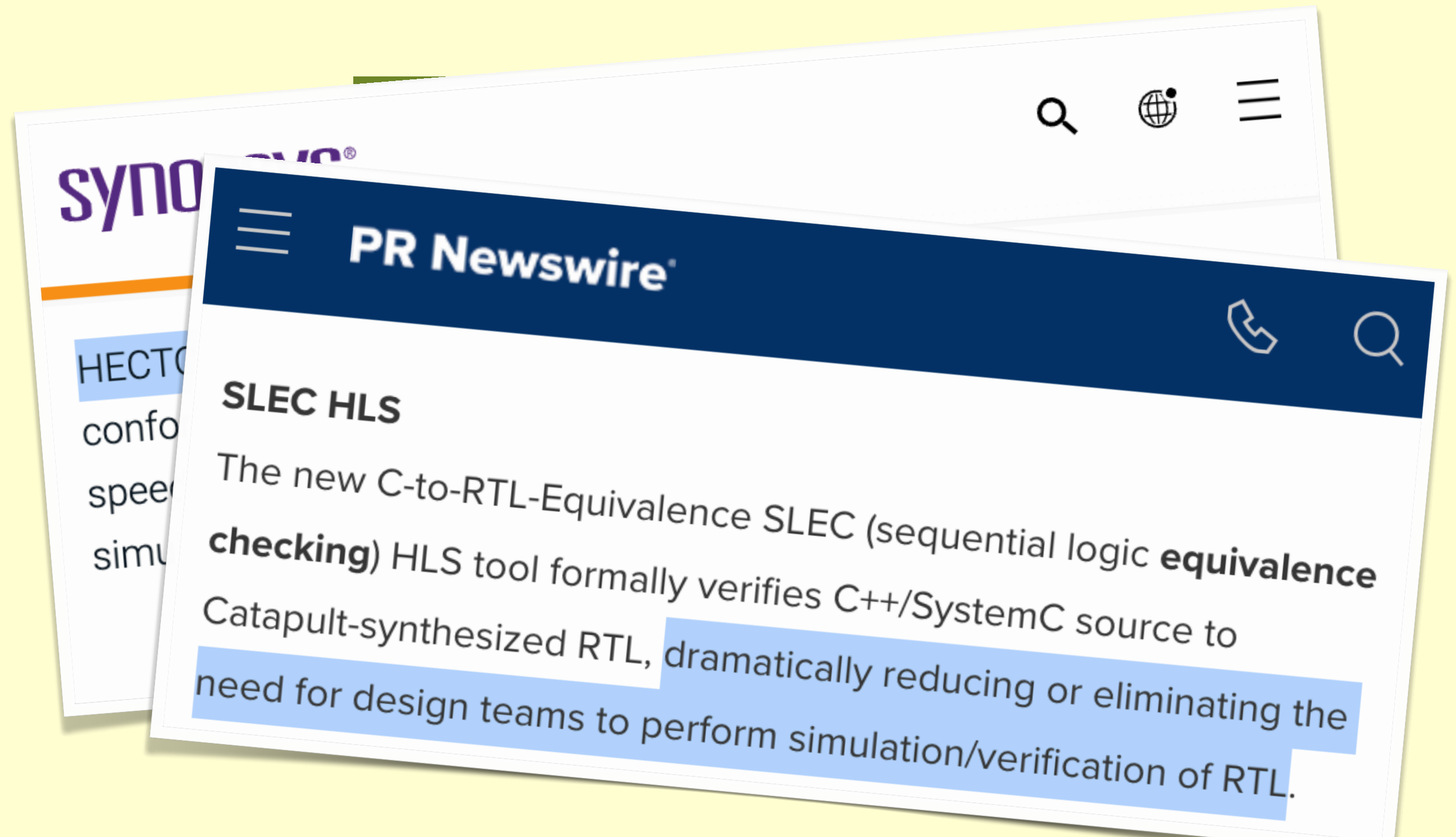
Testing equivalence checkers

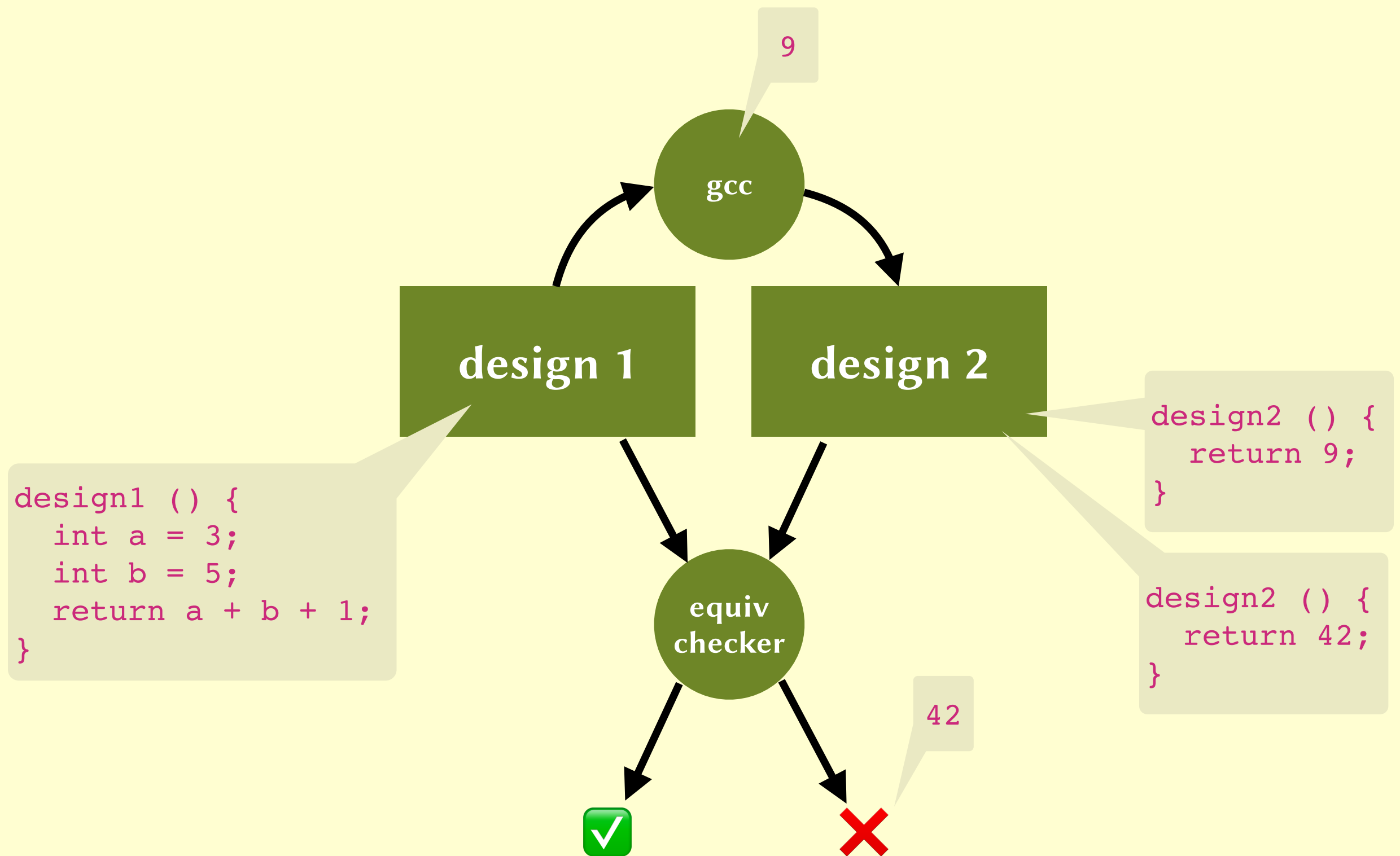


synopsys®

HECTOR delivers 100% confidence that the RTL design implementation conforms to the C/C++ reference algorithm, thereby significantly speeding up signoff for datapath components as compared to simulation-based techniques.

Testing equivalence checkers





Testing equivalence checkers

```
sc_fixed<8,3> a = 1.5;           //a == 001.10000
sc_int<8> b = sc_int<8>(a);      //b == 00000001 (✓)
                                //b == 00110000 (✗)
```

Further reading

DVCon '24

Who checks the checkers? Automatically finding bugs in C-to-RTL formal equivalence checkers

Michalis Pardalos
Imperial College London
michail.pardalos17@imperial.ac.uk

Alastair F. Donaldson
Imperial College London
alastair.donaldson@imperial.ac.uk

Emiliano Morini
Intel
emiliano.morini@intel.com

Laura Pozzi
Università della Svizzera italiana (USI) Lugano
laura.pozzi@usi.ch

John Wickerson
Imperial College London
j.wickerson@imperial.ac.uk

Abstract—C-to-RTL (register-transfer level) formal equivalence checkers (ECs) allow hardware implementations to be compared against software specifications. Thanks to their complete state-space coverage, ECs are trusted to authorise design sign-off. Therefore, ridding ECs of bugs is a top priority. In pursuit of this goal, we have developed Equifuzz, a technique and tool for randomized testing (fuzzing) of SystemC-to-RTL ECs. Equifuzz uses knowledge of SystemC semantics to generate rich designs that are known to be equivalent to trivial RTL designs. It has uncovered 7 *unsoundness* bugs in major commercial ECs (where the claimed equivalence is incorrect), and 5 *incompleteness* bugs (where the EC failed to prove equivalence between equivalent designs), all of which have been confirmed by the tool vendors. The fact that Equifuzz has been able to find serious bugs in extensively tested, major commercial ECs demonstrates that fuzzing is a valuable complement to the hand-crafted tests that EC developers use as standard.

I. INTRODUCTION

Formal equivalence checkers (ECs) such as Synopsys

found great success at uncovering bugs in many different tools, such as state-of-the-art C compilers [25, 15], graphics shader compilers [3] and OpenCL compilers [17]. Although fuzzing has not, to our knowledge, been applied to ECs before, it has been effective at finding bugs in other tools from the EDA realm, such as FPGA synthesis tools [8] and high-level synthesis tools [6]. Fuzzing has also been used to find bugs in verification tools other than ECs, such as SMT solvers [24] and software model checkers [26].

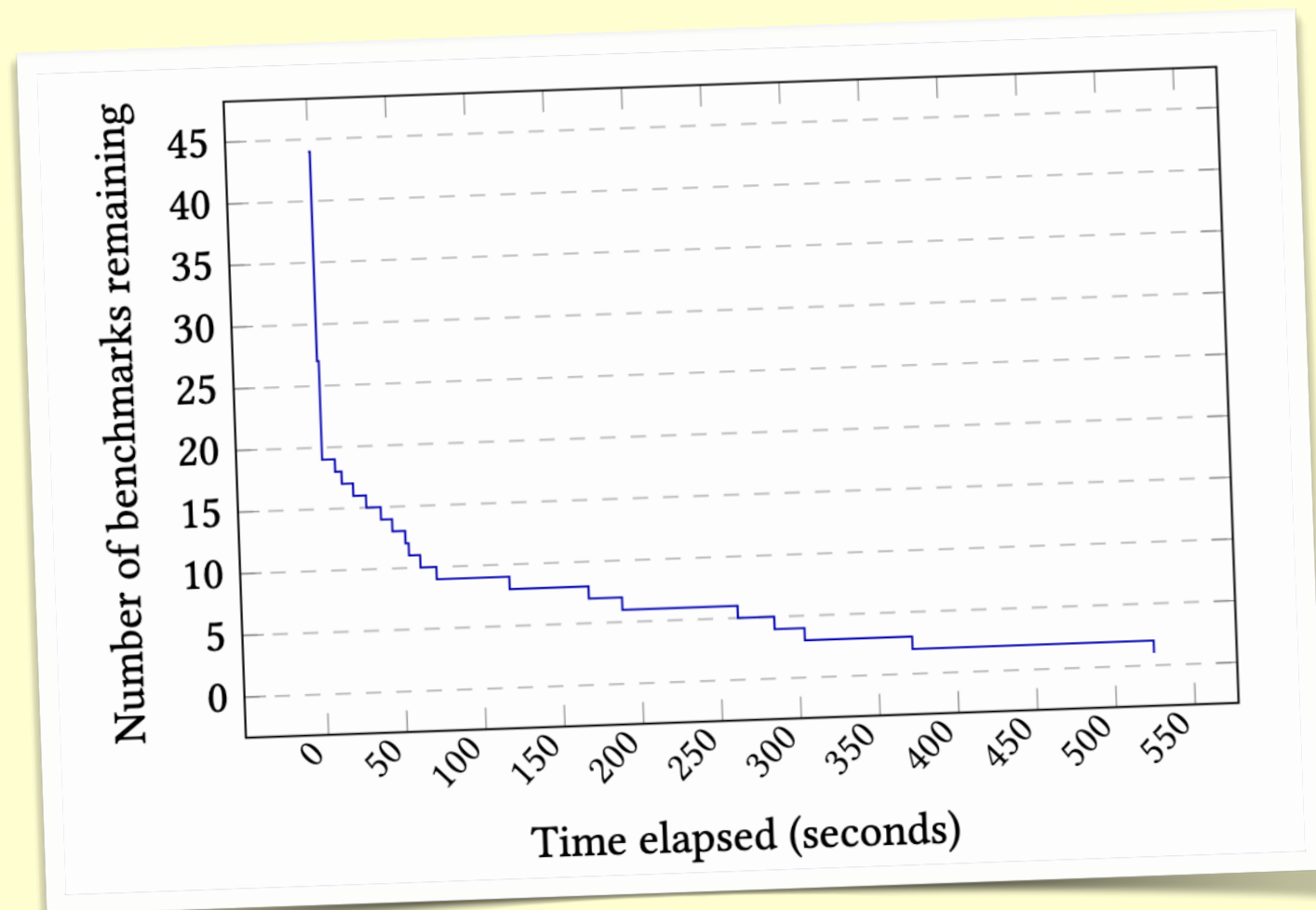
We have developed *Equifuzz*: a technique and tool for randomized testing of ECs that compare RTL implementations against SystemC specifications. We focus on SystemC because it is accepted by the three major commercial ECs, and often used by their industrial users. Equifuzz works by generating random SystemC programs. These are then compared (using the EC-under-test) against trivial RTL designs that are known to be equivalent. We record a potential bug if the EC returns a result other than “equivalent”.



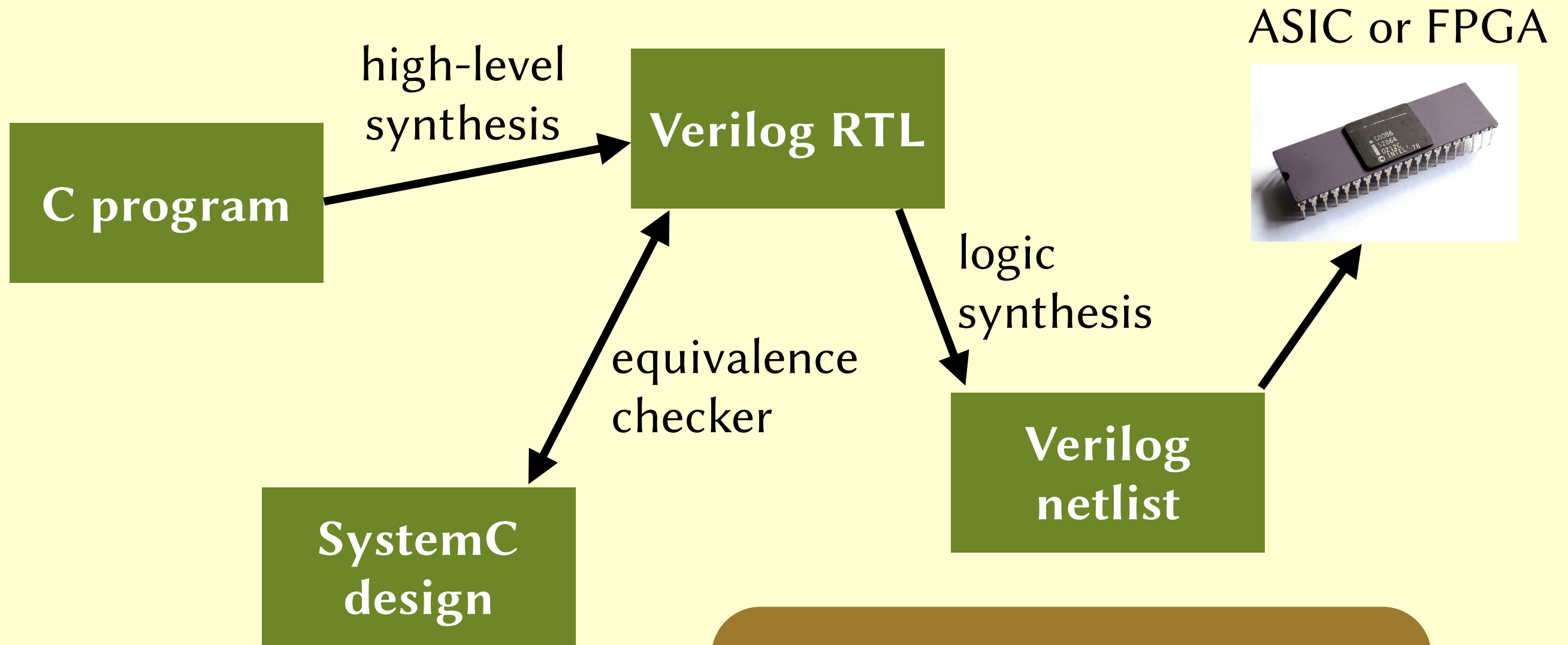
**Can we have more
reliable equivalence
checkers?**

Vera

- An equivalence checker built and proven-correct in Coq.
- Reduces the equivalence problem to an SMT query and uses SMTCoq to solve it.
- Preliminary results on the EPFL benchmarks are encouraging (44 out of 60 verify within a few minutes).
- A subtlety: undefined behaviour.



Outline



All these tools are too buggy. With more rigorous engineering, they can be made more suitable for safety- and security-critical settings.

