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 EC claimed equivalence incorrectly), 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 handcrafted tests that EC developers use as standard.**

I. Introduction

C-to-RTL formal equivalence checkers (ECs) such as Synopsys DPV [10], Cadence Jasper C2RTL [13] and Siemens SLEC [22] are valuable components of the hardware designer's toolbox. They are used to prove that an RTL implementation matches a high-level specification, usually written in C, C++, or SystemC. Where simulation-based approaches explicitly traverse the state space (and hence can only check a subset of inputs and a limited number of cycles), ECs provide an *unbounded* proof: valid for any inputs, and for any number of cycles. This exhaustive coverage means that ECs are deeply trusted, even to the extent of being used to authorise design sign-off. Indeed, the Synopsys marketing blog claims that:

> HECTOR delivers 100% confidence that the RTL design implementation conforms to the C/C++ reference algorithm, thereby significantly speeding up signoff [14]

while Siemens claim that their SLEC tool:

> enables designers to have the ultimate confidence to move to high-level synthesis [...] dramatically reducing or eliminating the need for design teams to perform simulation/verification of RTL. [21]

The guarantees that ECs provide are, of course, contingent on the ECs themselves being correct. In this work, we seek to help users and developers of ECs achieve higher confidence by identifying bugs that have escaped manual testing. This is important because bugs in ECs can have serious consequences, partly because they can be difficult to spot. Users will usually assume that an unexpected result from the EC is due to their own code, rather than a bug in the EC itself. Moreover, a bug in an EC could, like compiler bugs, be exploited to allow malicious code inserted into a design to slip through verification [2].

In order to evaluate, and hopefully improve, the reliability of ECs, we turn to randomized testing, also known as *fuzzing*. Fuzzing has 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 gives a result other than "equivalent".

We demonstrate the effectiveness of Equifuzz by uncovering 16 distinct bugs in three major commercial ECs, 12 of which have been confirmed by the vendors: 9 unsoundness bugs (where the EC claims equivalence incorrectly) and 7 incompleteness bugs (where the EC fails to find an existing equivalence).

Since SystemC is a superset of C++, we also compare the bug-finding ability of Equifuzz against two state-of-the-art C fuzzers (Csmith [25] and YARPGen [18]) by counting how many bugs each tool is able to find within 24 hours. Additionally, because Equifuzz allows for precise control over the size of generated programs, we investigate the effectiveness of different program sizes in catching bugs. These experiments demonstrate that even though Equifuzz generates smaller and simpler programs than state-of-the-art tools, it is much more capable at finding bugs in SystemC-to-RTL ECs.

We believe Equifuzz can complement the handcrafted test suites that EC developers currently use. Those suites can exhaustively check language features *individually* – and indeed we found no bugs involving only a single feature – while Equifuzz can (non-exhaustively) check interactions *between* features.

Our contributions

1) We present the first fuzzer for SystemC programs.
2) We present the first fuzzer oriented towards testing ECs.
3) We present a novel program generation algorithm that works by applying a random sequence of small transformations which maintain syntactic wellformedness at each step (§IV), speeding up test-case reduction (§V).
4) We present a technique for rephrasing false-negative EC bugs as false-positives, thus increasing their perceived severity (§III-B).

5) We present a total of 16 bugs in three major commercial ECs, among them 9 unsoundness bugs (§VI-A).
6) We investigate how Equifuzz's bug-finding ability changes with the size of the SystemC programs it randomly generates, and we also compare its bug-finding ability against two state-of-the-art C fuzzers (§VI-B).

II. Problem Statement

We are looking to create random, valid inputs for an EC, in order to trigger buggy behaviours. In general, an EC checks equivalence between programs in two languages, say L_1 and L_2. In the context of this paper, L_1 is SystemC and L_2 is RTL. An EC can be viewed as a function of the following type:

$$\langle P_1(\vec{x}),\ P_2(\vec{x})\rangle \rightarrow \{\text{True},\ \text{False},\ \text{Error}\}$$

where $P_i(\vec{x})$ is a program in language L_i, which takes $\vec{x}$ as inputs. The EC will have one of the following results:

- "True", if it can show that for all inputs $\vec{x}$, $P_1(\vec{x})$ and $P_2(\vec{x})$ produce *the same* output.
- "False", if it can find an input $\vec{x}$ for which $P_1(\vec{x})$ and $P_2(\vec{x})$ produce *different* outputs. In this case, $\vec{x}$ is presented to the user as a *counterexample*.
- "Error", if one of the programs is invalid in some way (e.g. it exhibits a syntax or type error).

The problem, then, is to generate pairs of programs $P_1(\vec{x})$ and $P_2(\vec{x})$, as well as, for each pair, an *expected result* $E \in \{\text{True},\ \text{False},\ \text{Error}\}$, according to whether the programs really are equivalent (or even valid). We can then use these program pairs with their expected results to look for violations of this expectation. These violations can be divided into the following categories:

- **False positive:** The EC says that two programs are equivalent, when in reality they are not. This could be because they exhibit different behaviours, or because at least one is invalid (e.g. contains a syntax error) and thus should be rejected.
- **False negative:** The EC says that two programs are not equivalent, when in reality they are (they are both valid and have the same behaviour for all inputs).
- **Valid input rejected:** The EC gives an error message, even though both input programs are valid.

Of those, a false positive is the most critical, as it could mean that an incorrect design is signed-off. We also refer to this kind of bug as an "unsoundness". A false negative or valid input being rejected, if not fixed, could result in lost engineering time for the verification engineer, who will have to re-write the design in a way that works around the bug. As we will demonstrate in §III-B, however, a false negative can often be rephrased to expose an associated — but much more serious — false positive. For this reason, the discovery of false negatives is valuable in practice.

It is also possible for the EC to fail to produce any result at all, usually because of reaching memory or time limits. Such results are expected, as the problem of equivalence checking is, in general, undecidable. We did not encounter any such issues in our investigation. If we had, we would not have considered them as bugs.

III. Fuzzer Design

Given this general problem statement, we constrain it to a more limited problem in order to arrive at a concrete implementation.

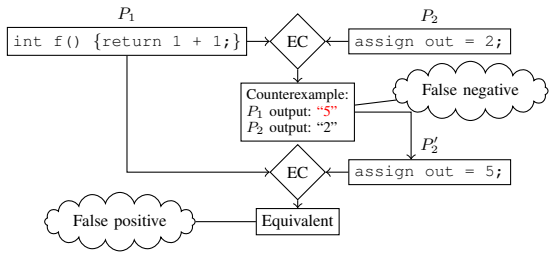

Fig. 1: Converting a false negative (failing to prove 1+1=2) into a false positive (proving 1+1=5).

```
sc_dt::sc_fixed<16,8> dut() {
    sc_dt::sc_int<1> x0 = sc_dt::sc_int<1>(-1);
    sc_dt::sc_fixed<16,8> x1 =
        sc_dt::sc_fixed<16,8>(x0.range(0, 0));
    return x1;
}
```

(a) SystemC code triggering false negative and false positive bugs.

```
module top
 ( output [15:0] out );
 assign out = 16'h0100
endmodule
```

(b) Verilog demonstrating false negative. EC incorrectly produces negative result

```
module top
 ( output [15:0] out );
 assign out = 16'hff00
endmodule
```

(c) Verilog demonstrating false positive. EC incorrectly produces positive result

Fig. 2: Example of false positive bug discovered from false negative.

A. Constraining the problem

The "direct" approach to solving the problem identified in §II would be to generate arbitrary $\langle P_1(\vec{x}),\ P_2(\vec{x})\rangle$ pairs, decide whether they are equivalent, and use that to test the EC. This approach has multiple issues. First, we run into the oracle problem [1]: deciding whether two arbitrary programs are equivalent amounts to building another EC, likely to contain more bugs than the ECs under test. Even overcoming that (say, using differential testing [20]) this approach would be unlikely to find any bugs. Two *entirely arbitrary* programs will almost certainly be non-equivalent in uninteresting ways.

Constraining the problem can both make it tractable and increase the chance of finding bugs. First, we focus our attention on *valid* programs; that is, SystemC programs that can be compiled and run without error. This ensures that we have an expected output to check against. Furthermore, valid programs are likely to exercise more of the equivalence checker than invalid programs. Second, we focus on *input-free* programs: programs which take no inputs and can therefore be expected to produce a constant result. This might appear to reduce the chances of triggering real-world bugs. However, input-free programs have proven to be useful in the domain of compiler testing: Csmith [25] and Verismith [8] both used exclusively input-free programs to find multiple miscompilation bugs in C compilers and FPGA synthesis tools, respectively.

Given these two restrictions, we can produce program pairs as follows: generate an input-free program P_1 (in SystemC). This program will have some constant output N. Then, P_2 can be a program in another language supported by the EC, e.g. Verilog, that outputs N directly, without any computation. The expected result E for the equivalence of P_1 and P_2 will therefore be "True", and any other result can be considered a probable bug.

B. Converting false negatives to false positives

When looking for EC bugs we want to maximise the proportion of false-positive (unsoundness) results. The design we have described,

however, can only directly find false-negative or valid-input-rejected bugs. This is because we generate program pairs that *are* equivalent. We demonstrate a way to recover a possible false-positive bug from a false-negative witnessed by an input-free program.

ECs produce a *counterexample* with negative results. For input-free (i.e. constant) programs, this counterexample contains the value that the EC *believes* that program evaluates to. If we know that the EC is incorrect, we can create a new program which returns the value that the EC expects. This can be used to trigger a false-positive bug in the EC. Figure 1 illustrates this process with an artificial example.

We provide a real example of this "conversion" process in Figure 2, using a bug found by Equifuzz. Equifuzz initially reports (after reduction, see §V) the pair of the SystemC in Figure 2a and the Verilog in Figure 2b as triggering a false negative bug. The code in Figure 2a produces a result of `0x0100` when running using the reference SystemC implementation [11], and yet, the EC claims that it is *not* equivalent to Figure 2b which is hard-coded to produce this value. This is a false negative bug. Examining the EC log, we find that it produced a counter-example, claiming that the SystemC code has a result of `0xFF00`. We can use this to generate a second Verilog module (Figure 2c), and run the EC a second time, comparing the original SystemC against this second Verilog module. Since this has the result that the EC (incorrectly) expects, it produces a positive result. Since this disagrees with the true value of the SystemC code, we have a false positive bug. More details on this specific bug can be found in §VI-A.

It is certainly *possible* to encounter a bug that cannot be converted to a false positive using this process but we did not during our testing. We were successful in converting every false negative result found during our testing into a false positive.

The converted bugs have the same underlying cause as the originals. However, they are presented in their most severe form, which improves the likelihood of a quick fix when reported the the tool developers. This is analogous to how a bug that has been automatically reduced is more likely to be fixed than a bug reported with a big program as generated by the fuzzer.

C. Undefined behaviour

Given that SystemC is a superset of C++, we must contend with the existence of undefined behaviour (UB) in the programs we generate for testing. The usual approach in compiler fuzzing, as popularised by Csmith [25], is to construct the generator such that it avoids programs that could trigger UB. The reason for this, in the context of compiler testing, is that code which triggers UB has no value in finding compiler bugs: the compiler is free to generate code that behaves in an arbitrary fashion. Therefore, no behaviour exhibited by the program (on input that triggers UB) could be considered a bug.

In the context of equivalence checking (and perhaps formal tooling more generally) the story is not quite as simple. Since the program can have any behaviours at all, it cannot be considered equivalent to *any* RTL designs! We take the stance that a positive result is a false-positive bug, and that any non-positive result is correct, whether that be a negative answer (possibly with a counterexample showing the input that triggers the UB) or simply an error.

Equifuzz is designed with this attitude towards UB in mind. It will occasionally generate programs containing UB. Ideally, we would detect when a generated program contains UB, and then expect a non-positive result from the EC. We attempted to perform this detection using the "undefined behaviour sanitizer" (UBSan) in the Clang compiler. Unfortunately, this approach ran into issues, with the official SystemC implementation from Accellera triggering UB

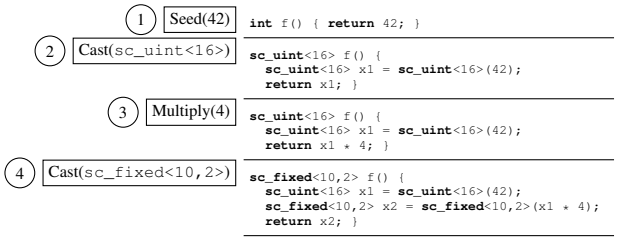

Fig. 3: Example of generation process

for valid SystemC code. We reported this issue to Accellera[1], who confirmed that the issue was present in the latest stable version of the library (SystemC 2.3), but that it had been fixed in the upcoming SystemC 3.0 release. Our current workaround has been to use the result given by the Accellera SystemC implementation even for programs with UB. This way, we can still find cases where the EC-under-test disagrees with Accellera's SystemC. After finding the discrepancy we can check whether the result constitutes a bug manually, often using UBSan to check if UB *could* be present.

It would also be possible to test ECs using *only* programs that contain UB, e.g. by using UBFuzz [16]. We leave this as future work.

IV. Fuzzer implementation

At a high level, the fuzzer follows the following process.

1) Generate an input-free SystemC program P using the method that will be specified below.
2) Compile and run it using GCC and the Accellera SystemC Class Library [11] to get its output N.
3) Use the EC to check that P is equivalent to the constant N.

The most complex part of this process is step 1. We need to generate programs that cover a wide range of SystemC features, while remaining valid SystemC. It is also important to make them amenable to reduction (as will be discussed in §V).

To get the expected result of the generated SystemC code we use the Accellera SystemC Class Library. This is a software implementation of the C++ classes mandated by the SystemC Standard [9].

We should note that any bugs that show up could possibly be bugs in either GCC or the Accellera SystemC implementation. Speaking precisely, the fuzzer is set up to find *discrepancies* between the EC-under-test and the combination of GCC and Accellera SystemC.

Once the fuzzer uncovers a possible bug it is up to a human to verify that this is, indeed, a bug in the EC, and not in another component of the system (including the fuzzer itself). This process is easy, thanks to the automated test-case reduction we have implemented (described in §V) which means that the test cases presented by Equifuzz are already minimised (usually to fewer than 5 lines of code).

With the above goals in mind, Equifuzz generates programs in *steps*. Starting from a simple *seed* program (e.g. a random integer constant), we iteratively apply *transformations* that grow the program. As we expand the generated program, we model it in two parts: the *head expression*, which will become the final return statement, and a list of statements that will become the function's body. Transformations modify the head expression and add statements to the list. They cannot modify the statements given by previous transformations. We also keep track of some additional information about the head expression, such as its type and bit-width. This is used to decide whether transformations can be applied. An example of this process is given in Figure 3: we begin with a seed expression `42`,

[1]Link to bug report: https://github.com/accellera-official/systemc/issues/61

which is transformed by adding a cast to a 16-bit unsigned integer, a multiplication by 4, and finally a second cast to a 10-bit fixed-point number with 2 fractional bits. This process can be repeated an arbitrary number of times to produce programs of a *precise* desired size that can be used as test cases.

We have currently implemented the following transformations. These can easily be extended to cover more of the SystemC language: Cast with assignment (e.g. `sc_int<8> y = sc_int<8>(x)`), functional cast (e.g. `sc_int<8>(x)`), range operator (e.g `x.range(10, 2)`), arithmetic (e.g. `x + 5`), unary operators (e.g. `- x`), ternary operator (e.g. `x ? 1 : 2`), bit select operator (e.g. `x[1]`), and reduction operators (e.g. `x.or_reduce()`). The cast operations can use any of the following types: `sc_(u)int`, `sc_big(u)int`, `sc_(u)fixed`, `int`, `unsigned`, `float`, `double`, `bool`.

It is worth remarking that transformation-based approaches to program generation have previously been used in a few other contexts. Donaldson et al. use it to test SPIR-V compilers [4], but where they seek a pair of minimally different programs that give distinct results after compilation, we seek a single minimal program that confuses the EC-under-test. Le et al. generate test programs by repeatedly *removing* instructions deemed unreachable from an initial program [15], whereas we build our test programs additively. Perhaps the closest program generation algorithm to ours is in HyperPUT [5], which aims to generate challenging test-cases for bug-hunting tools by repeatedly wrapping a buggy instruction in control-flow structures. However, our approach focuses on data-flow transformations (i.e. changing the value returned by the program) rather than control flow (i.e. changing which statements are executed). Moreover, where HyperPUT tests whether a bug-hunting tool can reach a specific, deeply nested line of code, we are testing whether an EC can reason correctly about the generated SystemC program as a whole.

Finally, having generated an experiment $\langle P, N \rangle$ as described above, our experiment runner generates a Verilog program that produces N, and a TCL script that tells the EC to compare that against the SystemC program P. These are all sent over SSH to the server that runs the EC, and the result is sent back. If the result is not as expected, the experiment is marked as a possible bug, and the reduction process described in the following section is initiated. We have also developed a web-based user interface for the fuzzer, which displays the test cases that are currently running or have been run. This allows quickly sifting through a large number of test cases, as well as simplifying debugging of the fuzzer.

V. Reducer Design and Implementation

After a bug-triggering example is found, it needs to be *reduced*. This makes it easier to understand (and therefore easier for the EC developers to fix) and also allows us to spot duplicate bugs. There are various techniques to achieve this in the literature. Sun et al. [23] described a general method of program reduction which uses language grammars to exclusively consider syntactically valid programs during reduction. Our reduction algorithm is based on work from Donaldson et al. [4] on test case reduction for transformation-based fuzzers. It operates on programs represented as the sequences of transformations that generated them. It attempts to remove transformations in batches, while still preserving the bug. We remove a sequence of n transformations from the sequence, generate the program for that sequence, and use it to test the EC (i.e. run the program with the reference SystemC implementation, and compare the program against the result value using the EC, expecting a positive result). If the response is incorrect (i.e. negative), then the bug can be triggered without these transformations so they can be removed, and we iterate with the reduced sequence. If the response is correct, then the transformations are necessary to trigger the bug, so we keep the same sequence, and repeat with a reduced n, until there is no value of $n \geq 1$ for which transformations can be removed.

Tool	False positive: invalid input	False positive: valid input	False negative	Valid input rejected	**Total**
EC 1	1	3	0	2	**6**
EC 2	2	1	1	2	**6**
EC 3	0	2	0	2	**4**

TABLE I: Summary of bugs found

VI. Experimental Results

We used Equifuzz to test three major commercial formal ECs and discovered false-positive, false-negative, and valid-input-rejected bugs in all. Results are listed in Table I. The names of the tools have been censored due to licensing restrictions. The bugs in EC 1 and EC 2 have have been reported to and confirmed by the tool vendors. The bugs in EC 3 have been reported to the tool vendor, but have not been confirmed by the time of publication.

We have classified the bugs found into four categories:

- **False positive: Invalid input** An invalid program was deemed equivalent to some value by the EC. Invalid here means either not allowed by the SystemC reference manual or triggering UB.
- **False positive: Valid input** A valid program was deemed equivalent to an incorrect value by the EC. These cases were discovered as false negatives and converted using the process described in §III-B.
- **False negative** A valid program, compared to its true result, produced an inequivalent result by the EC. Here, we only list bugs that *could not* be converted into false positives.
- **Valid input rejected** A valid program, compared to its true result, produced an error by the EC.

We used Equifuzz to test the three ECs continuously through the nine months of its development. During that time, new versions of the ECs being tested were released with bugs that we reported in previous versions fixed, while we continuously added new features to Equifuzz. We searched for bugs opportunistically, running the fuzzer for some period of time (usually overnight or during a weekend) after implementing a new feature. If this revealed new bugs, there would usually be a collection of possible bugs reported by Equifuzz, which we would then de-deduplicate to uncover the true number of new bugs discovered. This was aided by the reduction described in §V.

We also had to address the issue of easier-to-trigger bugs re-appearing, increasing the volume of possible bugs to look through, and obscuring harder-to-trigger bugs. The transformation-based approach to test-case generation allowed us to work around this. We were able to add restrictions on the transformations that could be applied, preventing us from triggering known bugs, but not disabling entire features, which could still have bugs that could be tested.

All bugs were initially found using programs generated by sequences of 30 transformations. After reduction, all were reduced to sequences of two or three transformations. Notably, we found no single-transformation bugs. This means that there was no single feature that was problematic in any of the ECs tested. All bugs stemmed from interactions *between* language features.

The speed of the bug-finding process was entirely limited by the speed of the EC being tested, and not by Equifuzz. Generating a program takes well under a second for the 30-transformation programs we used (and also for much larger sizes that were attempted).

```
sc_dt::sc_uint<8> dut() {
  sc_dt::sc_fixed<10,8> x0 =
    sc_dt::sc_fixed<10,8>(-1);
  sc_dt::sc_uint<8> x1 = sc_dt::sc_uint<8>(x0);
  return x1;
}
```

True result	0xFF
Result accepted by EC 1	0xFC

Fig. 4: Example of false positive bug found in EC 1

```
double dut() {
    int x0 = 1;
    return double(bool(x0));
}
```

True result	0x3FF0000000000000
Result accepted by EC 2	0x0000000000000001

Fig. 5: Example of false positive bug found in EC 2

Compiling and running the generated program usually took a couple of seconds, meaning that the overall program-generation process was in the order of 3-5 seconds. Running the ECs we tested took about ten seconds for EC 2 and about one minute for EC 1. Furthermore, Equifuzz will generate programs while waiting for the EC to finish and keep a queue of test programs ready. This all means that the EC-under-test is going through test programs at the fastest rate allowed by the hardware and number of licenses.

A. Bug examples

We present a set of bugs as found by Equifuzz in the ECs we tested. Figure 4 lists the code to trigger one of the false-negative bugs found in EC 1. The problematic operation here was the cast from `sc_fixed` to `sc_uint`. According to the SystemC specification (which the reference implementation followed correctly), this cast should truncate the fractional part of the fixed point number, then use the integer part as an `sc_uint` value. Instead, EC 1 assumes that the operation should use the *entire* `sc_fixed` value, and re-interpret it as an `sc_uint`.

Figure 5 shows an example of a bug found in EC 2. The code returns a double-precision floating point value of 1. EC 2 deems this program equivalent to RTL producing an *integer* value of 1. According the documentation of EC 2, it should, in the default configuration which we were using, perform a bit-by-bit comparison. Instead, it appears to be checking that the *values* of the SystemC and RTL results are equivalent. It *should*, therefore, find this program equivalent to RTL producing a floating-point value of 1 (`0x3FF0000000000000`). This bug does not make use of any SystemC-specific code. Of the bugs reported here, this is the only one which could, in principle, be found by Csmith or YARPGen. It also demonstrates that Equifuzz is also capable of finding bugs in the pure C fragment of SystemC.

In Table I, for EC 1, we list an unsoundness bug due to an invalid input being deemed equivalent to some value. This refers to a bug that was discovered as an incompleteness, but could be converted into a false positive through manual investigation. The original, discovered by Equifuzz and listed on the left in Figure 6, should produce a result of 157952. EC 1 produces an error when attempting to check this code, making this an incompleteness bug. After manual testing, attempting to get the EC to accept this code, we found a related bug, listed on the right of Figure 6, triggered by calling a (non-existent) method `.to_int()` on the result of a multiplication between an `sc_int` and an integer constant.

```
int f() {
  sc_int<63> x = 1234;
  return x * 128; }
```

```
int f() {
  sc_int<63> x = 1234;
  return (x * 128).to_int(); }
```

Fig. 6: Left: Incompleteness bug, as found by Equifuzz. Right: False positive bug, found by manual investigation.

This code is illegal according to the SystemC standard. The multiplication `x * 128` is meant to have type `int`, since, according to the standard, `x` should be implicitly cast to a native C++ `int`. Native C++ `int`s do not have a `to_int()` method (or, indeed, any method). However, the EC accepts it, and gives a positive result when comparing it to the constant value 157952, which we consider a bug (as discussed in §III-C).

Finally, Figure 2 also demonstrates a bug found by Equifuzz, caused by the interaction between the `sc_int::range()` method, and casting to `sc_fixed`.

B. Fuzzer evaluation

Given that Equifuzz is the first fuzzer targetting SystemC, we picked two *C* fuzzers as comparison points: Csmith [25] and YARPGen [18]. As SystemC is a C++ library, programs generated by these fuzzers will be accepted by the tools that expect SystemC as input.

We ran a set of controlled experiments comparing three different configurations of Equifuzz against Csmith and YARPGen. For Equifuzz, we compare three different sizes to investigate whether that has any impact on bug-finding ability. For Csmith and YARPGen there was no way to configure a desired overall program size, so we used their default settings. Each experiment ran for 24 hours, on an Intel Xeon E5-2630 server, using a single CPU core (due to license restrictions of the EC). We initially used a timeout of five minutes for the EC run, however that generated a large number of timeouts for the Csmith- and YARPGen-generated programs, so we also include an experiment with a 30 minute timeout.

This evaluation was conducted in a different way from the fuzzing campaign discussed previously. We ran Equifuzz as "default", placing no restrictions in the generated programs. This meant that bugs would be triggered multiple times, "wasting" test runs on bugs that had been already found. When hunting for bugs, the transformation-based structure of Equifuzz gave us the option to restrict the generated programs in very targeted ways (e.g. "no xor_reduce operations on expressions of type `sc_bigint`"), working around specific bugs without affecting fuzzing effectiveness. Because this is an ad-hoc process, and to have a fair comparison with Csmith and YARPGen, we did not use this capability in our evaluation.

Based on the Table II results, we make the following observations:

1) The number of false negatives found by Equifuzz in 24 hours is (for 3 and 30 transformations) orders of magnitude larger than that of by Csmith and YARPGen.
2) Programs generated by Csmith and (especially) YARPGen required more time to be checked by the ECs. This meant that a longer timeout was needed to get meaningful results, but also that fewer experiments could run in the same timeframe.
3) When Equifuzz generates larger programs, its testing rate actually *increases* (from 6367 runs in 24 hours to 8892). This is counter-intuitive because one would expect larger programs to take longer to check, and hence the testing rate to decrease. However, almost all of the larger programs are causing the EC to error out, and this tends to happen quickly. Almost all of the errors are caused by the `sc_bigint::xor_reduce()` method, which the EC claims to support but actually causes an error. The larger the generated program, the higher the chance that it contains at least one occurrence of this feature.

Fuzzer	Total runs	True positives	False negatives	Timeouts	Other errors	Average non-timeout time
Equifuzz (3 transformations)	6367	6035	176	0	156	14s
Equifuzz (30 transformations)	8654	4132	789	0	3733	10s
Equifuzz (300 transformations)	8892	16	17	0	8859	9.7s
YARPGen (5' timeout)	1029	789	1	239	0	48s
YARPGen (30' timeout)	193	148	0	44	1	63s
Csmith (5' timeout)	2008	1808	13	66	121	33s
Csmith (30' timeout)	1461	1347	7	14	93	42s

TABLE II: Fuzzer evaluation results. All experiments ran for 24 hours, targetting EC 2 from Table I

4) Many of the YARPGen- and Csmith-generated programs take more than 30 minutes to check. This could be because these programs are much larger than those generated by Equifuzz.

VII. Conclusion

We have introduced Equifuzz, a fuzzer for formal equivalence checkers using SystemC. Equifuzz generates input-free SystemC programs in a step-by-step manner, which allows for straightforward test-case reduction. We have demonstrated its effectiveness by using it to uncover 16 bugs in major commercial ECs, including 7 confirmed unsoundness bugs (and 2 pending confirmation) that could have led to incorrect designs being signed off.

In the future, there are improvements we believe could be made to Equifuzz to expand the classes of bugs that it can detect, and generally improve its utility to the users and developers of ECs. Primarily, we would like to generate designs involving deeper control flow as well as designs that operate on inputs. This could expand the classes of bugs that Equifuzz can find.

Longer term, we are exploring the possibility of avoiding EC bugs altogether by proving the implementation of the EC bug-free using a proof assistant. This approach has been found to be feasible for other EDA tools, such as Vericert (high-level synthesis, verified in Coq) [7] and Lutsig (logic synthesis, verified in HOL4) [19], but has not yet been attempted for ECs (except for very simple combinational circuits of basic gates [12]).

VIII. Acknowledgements

We thank Sam Coward, Theo Drane, Alfred Koelbl, Richard Langridge, Tom Melham, Manish Pandey, Cesar Rodriguez, Sean Safarpour, and Andreas Tiemeyer.

We acknowledge financial support from the UK EPSRC via a Programme Grant (EP/R006865/1) and a PhD scholarship.

References

[1] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. "The Oracle Problem in Software Testing: A Survey". In: *IEEE Trans. on Softw. Eng.* 41.5 (May 2015).

[2] Scott Bauer, Pascal Cuoq, and John Regehr. "Deniable Backdoors using Compiler Bugs". In: *PoC GTFO* 9 (2015).

[3] Alastair F. Donaldson and Andrei Lascu. "Metamorphic testing for (graphics) compilers". In: *MET*. May 2016.

[4] Alastair F. Donaldson, Paul Thomson, Vasyl Teliman, Stefano Milizia, André Perez Maselco, and Antoni Karpiński. "Test-case reduction and deduplication almost for free with transformation-based compiler testing". In: *PLDI*. 2021.

[5] Riccardo Felici, Laura Pozzi, and Carlo A. Furia. "HyperPUT: generating synthetic faulty programs to challenge bug-finding tools". In: *EMSE: Empirical Softw. Eng.* 29.2 (2024).

[6] Yann Herklotz, Zewei Du, Nadesh Ramanathan, and John Wickerson. "An Empirical Study of the Reliability of High-Level Synthesis Tools". In: *FCCM*. 2021.

[7] Yann Herklotz, James D. Pollard, Nadesh Ramanathan, and John Wickerson. "Formal verification of high-level synthesis". In: *Proc. ACM Program. Lang.* 5.OOPSLA (2021).

[8] Yann Herklotz and John Wickerson. "Finding and Understanding Bugs in FPGA Synthesis Tools". In: *FPGA*. Feb. 2020.

[9] "IEEE Standard for Standard SystemC Language Reference Manual". In: *IEEE Std 1666-2011* (2012).

[10] Synopsys Inc. *VC Formal Datapath Validation*. URL: https://www.synopsys.com/verification/static-and-formal-verification/vc-formal/vc-formal-datapath-validation.html (visited on 11/16/2023).

[11] Accellera Systems Initiative. *SystemC Class Library*. Dec. 2, 2022. URL: https://github.com/accellera-official/systemc (visited on 11/19/2023).

[12] Wilayat Khan, Farrukh Aslam Khan, Abdelouahid Derhab, and Adi Alhudhaif. "CoCEC: An Automatic Combinational Circuit Equivalence Checker Based on the Interactive Theorem Prover". In: *Complex.* 2021 (2021).

[13] Vinod Khera. *Jasper C2RTL App for Datapath Verification*. July 12, 2022. URL: https://community.cadence.com/cadence_blogs_8/b/fv/posts/jasper-c2rtl-app-for-datapath-verification (visited on 11/16/2023).

[14] Alfred Koelbl, Kiran Vittal, and Pratik Mahajan. *Verifying Complex Datapath Designs with HECTOR*. Feb. 23, 2021. URL: https://www.synopsys.com/blogs/chip-design/verifying-complex-datapath-designs-with-hector.html (visited on 10/13/2023).

[15] Vu Le, Mehrdad Afshari, and Zhendong Su. "Compiler validation via equivalence modulo inputs". In: *PLDI*. 2014.

[16] Shaohua Li and Zhendong Su. "UBFuzz: Finding Bugs in Sanitizer Implementations". In: *ASPLOS*. 2024.

[17] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. "Many-core compiler fuzzing". In: *PLDI*. 2015.

[18] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. "Random testing for C and C++ compilers with YARPGen". In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020).

[19] Andreas Lööw. "Lutsig: a verified Verilog compiler for verified circuit development". In: *CPP*. 2021.

[20] William M McKeeman. "Differential testing for software". In: *Digital Technical Journal* 10.1 (1998).

[21] Mentor Graphics Corporation. *Mentor Ushers in New Era of C++ Verification Signoff with New Catapult Tools and Solutions*. June 6, 2017. URL: https://www.prnewswire.com/news-releases/mentor-ushers-in-new-era-of-c-verification-signoff-with-new-catapult-tools-and-solutions-300469227.html (visited on 10/13/2023).

[22] Siemens. *C++/SystemC/RTL Formal*. URL: https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/hls-verification/slec/ (visited on 11/16/2023).

[23] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. "Perses: Syntax-Guided Program Reduction". In: *ICSE*. 2018.

[24] Dominik Winterer, Chengyu Zhang, and Zhendong Su. "Validating SMT solvers via semantic fusion". In: *PLDI*. 2020.

[25] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. "Finding and understanding bugs in C compilers". In: PLDI. 2011.

[26] Chengyu Zhang, Ting Su, Yichen Yan, Fuyuan Zhang, Geguang Pu, and Zhendong Su. "Finding and understanding bugs in software model checkers". In: *FSE*. 2019.