Who checks the checkers? Automatically finding bugs in C-to-RTL Formal Equivalence Checkers

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

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

Emiliano Morini
Intel
USA
emiliano.morini@intel.com

Laura Pozzi
University of Lugano
Switzerland
laura.pozzi@usi.ch

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

ABSTRACT

C-to-RTL formal equivalence checkers (ECs) allow hardware implementations to be compared against software specifications. Thanks to their complete state space coverage, they are often trusted to authorise design sign-off, so ridding them of bugs is a top priority. We have developed Equifuzz, a fuzzer for ECs that take SystemC as input. Equifuzz has uncovered 7 *unsoundness* bugs in commercial ECs (where the EC claimed equivalence incorrectly), and 5 *incompleteness* bugs (where the EC failed to find an existing equivalence). We believe that Equifuzz can be a valuable addition to the testing infrastructure used by EC developers, with its randomly generated designs complementing existing suites of handcrafted tests.

ACM Reference Format:
Michalis Pardalos, Alastair F. Donaldson, Emiliano Morini, Laura Pozzi, and John Wickerson. 2024. Who checks the checkers? Automatically finding bugs in C-to-RTL Formal Equivalence Checkers . In *Proceedings of Design Automation Conference (DAC'24).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

C-to-RTL formal equivalence checkers (ECs) such as Synopsys DPV [11], Cadence Jasper C2RTL [14] and Siemens SLEC [23] are valuable tools in the hardware designer's toolbox. They are used to prove that an RTL implementation matches a higher-level specification, usually written in C/C++/SystemC. Where simulation-based approaches explicitly traverse the state space (and hence only exhaustively check a subset of inputs or a limited number of cycles), formal equivalence checkers provide an *unbounded* proof: valid for any and all inputs to the design, and for any number of cycles. This exhaustive coverage means that ECs are deeply trusted, even to the extent of authorising 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 [15]

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]

In this paper, we seek to help users and developers of ECs achieve higher confidence by identifying bugs that might have escaped manual testing. This is important because bugs in ECs can have serious consequences, partly because they can be very difficult to spot. A user checking a design with an EC will do so assuming that the design matches the specification. If the design is incorrect and the EC has a bug that hides that incorrectness, then that will not become apparent until much later, when the bug in the design has had its impact. Moreover, a bug in an EC could be exploited to allow code maliciously inserted into a design to slip through verification [2].

In order to evaluate, and hopefully improve, the reliability of ECs, we turn to random testing, also known as fuzzing. This is a technique that has found great success in uncovering bugs in many different tools, from compilers [26] to graphics drivers [4]. We have developed *Equifuzz*: a fuzzer for ECs that compare an 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 repeatedly generating random SystemC programs. These are compared (using the EC-under-test) against trivial RTL designs that they are known to be equivalent to, recording a potential bug if the EC gives a result other than "equivalent". We demonstrate its effectiveness by using it to uncover 12 distinct bugs in two commercial ECs, including 7 unsoundness bugs. Our results show that a fuzzer like Equifuzz can make a valuable addition to the extensive testing that formal tools like ECs already go through, by catching edge-case bugs that had gone unnoticed.

DAC'24, June 23–27, Moscone West, San Fransisco
© 2024 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM. . . $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1.1 Related Work

Fuzzing is a well-known technique, which has been applied on many different types of targets. Compilers are a usual target, due to the possible widespread impact of bugs in them, with a large variety of techniques presented in the literature [3]. Fuzzing has been used to find bugs in state of the art C [26, 16], graphics shader [4] and OpenCL [17] compilers. It has also been effective in finding bugs in tools in the same space as ECs: In the realm of EDA, bugs have been found in FPGA synthesis [7] and high-level synthesis tools [8]. More generally, purpose-made fuzzers have also found bugs in verification tools such as SMT solvers [25] and software model checkers [27].

Another possible way to avoid bugs in ECs altogether is to prove the implementation bug-free using a proof assistant. Khan et al. [13] have done this using Coq, though their checker only handles simple combinational circuits built from basic Boolean gates. Other EDA tools that have been proven bug-free using such deductive methods include Vericert (high-level synthesis, verified in Coq) [9] and Lutsig (logic synthesis, verified in HOL4) [19].

2 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 two languages L_1 and L_2. In the context of this paper, L_1 is SystemC and L_2 is RTL, but we present our approach more generally so that it could be applied to other languages. An EC can be viewed as a function of the following type:

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

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 $P_2(\vec{x})$ (i.e. P_1 and P_2, given the same input, $\vec{x}$) have *the same* output.
- "False", if it can show that $P_1(\vec{x})$ and $P_2(\vec{x})$ have *different* outputs for *some* inputs, i.e. that there exists an $\vec{x}$ for which P_1 and P_2 compute different results. In this case, it will also give a *counterexample* showcasing an example input $\vec{x}$ that causes a disagreement in output.
- "Error", if the input is invalid in some way (e.g. it exhibits a syntax or type error).

The problem, then, is to generate pairs of programs $\langle P_1(\vec{x}),\ P_2(\vec{x})\rangle$, together with an *expected result* $E \in \{\text{True}, \text{False}, \text{Error}\}$ for each pair, 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 either because the two programs exhibit different behaviours (*valid programs wrongly deemed inequivalent*), or because one of the two is invalid in some way (e.g. contains a syntax error) and thus should be rejected (*invalid programs deemed equivalent*).

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 class of bug, as it could mean that an incorrect design is signed-off. We also refer to this kind of bug as "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 their design in a way that works around the bug. As we will demonstrate in Subsection 3.3, however, a false negative can be exploited to create a false positive, so false negatives also need to be addressed seriously. It is also possible for the EC to fail to produce a result, usually because of reaching memory or time limits. Such results are expected, as there are limits to the kinds of programs that an EC could be expected to cope with. We did not encounter any such issues in our investigation, and, even if we had, we would not have reported them as bugs.

3 FUZZER DESIGN

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

3.1 Constraining the problem

The "direct" approach to this problem would be to generate arbitrary $\langle P_1(\vec{x}),\ P_2(\vec{x})\rangle$, decide whether they are equivalent, and use that to test the EC. This approach has multiple problems. First, we run into the oracle problem [1]: Deciding whether two arbitrary programs are equivalent amounts to building another equivalence checker (which would likely contain many more bugs than the commercial ECs we are testing). 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 chances of finding bugs. First of all, we can focus our attention on *valid* programs: That is, SystemC programs that can be compiled and run without error. There are multiple reasons for this. First, an invalid program will simply not test as much of the EC as we would like. For example, a program that does not even parse as valid C code will be rejected before most of the equivalence checker's code has even had a chance to run. Such a program is, therefore, less likely to trigger a bug in the EC. Secondly, we need a program that can run to completion in order to have an expected output to check against.

We further decide to focus on *input-free* programs. That is, programs that take no inputs and can be expected to produce a constant result. This might appear to dramatically reduce the chances of triggering real-world bugs. However, input-free programs have proven to be useful in the domain of compiler testing: CSmith [26] and Verismith [7] both used exclusively input-free programs to find multiple miscompilation bugs in C compilers and FPGA synthesis tools, respectively.

Given these two restrictions (i.e. generating only valid, input-free programs), we can generate program pairs as follows: Generate an input-free program P_1 (in SystemC), which simply performs a computation on constants. 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 performing any computation. The expected result E for the equivalence of P_1 and P_2 will therefore be True. Any non-equivalent result from an EC (i.e. negative or error) can be considered a probable bug.

3.2 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 the Csmith tool [26], is to construct the generator such that it avoids programs that could trigger UB. This is because, when talking about compiler testing, code that triggers UB has no value with regards to finding bugs: The compiler is free to generate code that behaves in an arbitrary fashion when executed on inputs that trigger UB, therefore, no behaviour exhibited by the program on those inputs could be considered a bug.

In the context of equivalence checking (and perhaps formal tooling more generally) the story is not quite as simple. What should a C-to-RTL equivalence checker do when given a C program that triggers undefined behaviour? Since the program can have any behaviours at all, it is both equivalent and non-equivalent to any and all 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 counter-example showing the input that triggers the UB) or simply an error.

We have taken this approach to UB into account with Equifuzz. Equifuzz will occasionally generate programs containing undefined behaviour. Given that we are testing ECs, these programs are not "useless", as they would be if we were testing compilers. The ideal approach here would be to 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) feature of the Clang compiler [18]. Unfortunately, this approach ran into issues with the Accellera SystemC implementation triggering undefined behaviour for valid SystemC code. We reported this issue to the developers[1], who confirmed that the issue was present in the latest stable version of the library (SystemC 2.3), but that it had already been fixed in the upcoming SystemC 3.0 release. Our alternative approach 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.

3.3 Converting false negatives to false positives

When looking for EC bugs we want to, ideally, 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. Fortunately for the class of programs we are working with (input-free), there is a way to recover a (possible) false-positive bug from a false-negative. Recall that ECs produce a *counterexample* when they give a negative result. 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, then we can create a new program which directly returns the value that the EC expects. We can then use this to trigger a false-positive bug in the EC. Figure 1 illustrates this process.

This process was successful in converting every false negative result found during our testing (where the EC provided a counterexample) into a false positive. 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. Such a bug would manifest as a program that should be equivalent to value x, but when compared to it using the EC produces a negative result with a counterexample claiming it is actually equivalent to value y, but then when compared to y produces another negative result, with another counterexample claiming it is equal to a third value z.

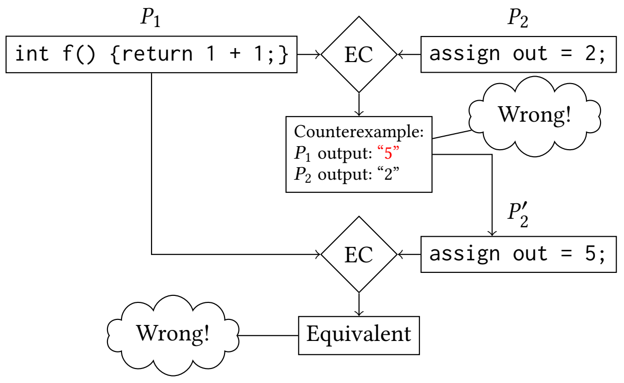

Figure 1: Converting a false negative (failing to prove 1+1=2) into a false positive (proving 1+1=5).

4 FUZZER IMPLEMENTATION

At a high level the fuzzer follows the following process to generate inputs for the EC.

(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 [12] to get its output N.
(3) Use the EC to check that P is equivalent to the constant N. If the EC finds them non-equivalent or cannot decide, then there is a possible bug.

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 Section 5).

In order 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 [10].

We should note here: Any bugs that show up could possibly be bugs in either GCC or the Accellera SystemC implementation. Speaking precisely, what the described system is set up to find

[1]Link to issue tracker removed for blind review

Seed(42)

```
int f() {
  return 42;
}
```

Apply: Cast(sc_uint<16>)

```
sc_uint<16> f() {
  sc_uint<16> x1 = sc_uint<16>(42);
  return x1;
}
```

Apply: Multiply(4)

```
sc_uint<16> f() {
  sc_uint<16> x1 = sc_uint<16>(42);
  return x1 * 4;
}
```

Apply: Cast(sc_fixed<10,2>)

```
sc_fixed<10,2> f() {
  sc_uint<16> x1 = sc_uint<16>(42);
  sc_fixed<10,2> x2 =
      sc_fixed<10,2>(x1 * 4);
  return x2;
}
```

Figure 2: Example of generation process

is discrepancies between the EC under test and the combination of GCC and Accellera SystemC. Once the fuzzer uncovers such a discrepancy (i.e. "possible bug") it is up to a human to verify that this is, indeed, a bug in the EC, and not any of the other component of the system (including the fuzzer itself). This means examining the bug-triggering program against the SystemC reference manual to verify that it should, indeed, have the behaviour that the fuzzer expected it to have (rather than the one that the EC expected). This process is generally quite easy, thanks to the automated test case reduction we have implemented (we give more details in Section 5) which means that the test cases presented by Equifuzz are minimised (usually under 5 lines of code). Also, in our experience, the EC under test has always been to blame for the discrepancy.

With the above goals in mind, we follow an approach inspired by HyperPUT [6]. Equifuzz generates programs in *steps*. Starting from a simple *seed* program (e.g. a random integer constant), we iteratively apply *transformations*. 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 body of the function. 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 extra 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 2: We begin with a seed expression 42, which is transformed by adding a cast, a multiplication and finally a second cast.

4.1 Experiment runner

Having a way to generate test cases, Equifuzz also needed the surrounding structure to run them against the ECs being tested. This process was formed out of practical constraints around the servers that were needed to run the ECs-under-test

This runner follows the following process:

(1) Generate an experiment as described in Section 4.
(2) Generate a TCL script to drive the EC we are testing against, and a Verilog design containing the expected result value.
(3) Connect to a server that can run the EC over SSH, copy the generated SystemC code, the Verilog program for comparison and the TCL script
(4) Run the EC to get a result.
(5) If the result is not as expected, mark the code as a possible bug and begin reducing the test case, using the same process to run the reduced test cases.

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.

5 REDUCER DESIGN AND IMPLEMENTATION

After a bug-triggering example is found, we need to *reduce* it. 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. In recent work, PERSES [24] described a general method of program reduction which uses language grammars to exclusively consider synactically valid programs during reduction. Our reduction algorithm is based on work from Donaldson et al. [5] 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 try again with a reduced N. This is repeated until there is no value of $N \geq 1$ for which transformations can be removed.

6 EXPERIMENTAL RESULTS

We have run Equifuzz against two commercial formal equivalence checkers and discovered false positive, false negative, and valid input rejected bugs in both. All bugs have been reported to and confirmed by the tool vendors. 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.

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

Table 1: Summary of bugs found

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 Subsection 3.3.

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.

All results are listed in Table 1.

We used Equifuzz to test the two ECs continuously through the 9 months of its development. During that time, new versions of the Equivalence checkers being tested were released with bugs found 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) when we had implemented a new feature. If the new feature had allowed for a new bug or bugs to be uncovered it would usually mean there would 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 made much easier thanks to the reduction described in Section 5. We also had to address the issue of easier to trigger bugs constantly re-appearing, increasing the volume of possible bugs to look through, and possibly obscuring harder to trigger bugs. We worked around this by introducing restrictions to the programs Equifuzz was allowed to generate in order to avoid known bugs.

All bugs were initially found using programs generated by sequences of about 30 transformations. After reduction, all were reduced to sequences of either two or three transformations. Notably, there were no examples of 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 is 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). Compiling and running the generated program to get an expected result 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 between 20 seconds and a full minute. 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.

```
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

Figure 3: Example of bug found in EC 1

6.1 Bug examples

As an example of the kinds of bugs that Equifuzz can find, Figure 3 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, and then use the integer part as a `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`.

In Table 1, we list a one 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 bug, as discovered by Equifuzz, was triggered by the code in Figure 4. This code is valid SystemC, and produces a result of 157952. However, after some manual testing, attempting to get the EC to accept this code, we found a related bug, triggered by the code in Figure 5.

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 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 Subsection 3.2).

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

Figure 4: Code to trigger incompleteness bug, as found by Equifuzz. Result is 157952**. EC 1 produces an error when comparing this against any RTL.**

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

Figure 5: Code to trigger false positive bug. Should not compile according to SystemC standard, but compares equal to RTL producing the value 157952 **under EC 1**

7 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 12 bugs in commercial ECs, including 7 unsoundness bugs that could have led to incorrect designs being signed off.

In the future, there are certain improvements we believe could be made to equifuzz in order to expand the classes of bugs that it can detect, and generally improve its utility to the users and developers of ECs:

(1) We would like to evaluate how different sizes for the generated programs affect the rate at which bugs are found, and, indeed, whether certain bugs are discovered at all. There is a trade-off here between having more operations that could possibly trigger bugs while also having the possibility that operations later in the program "hide" bugs triggered by earlier operations.
(2) Currently, we can only generate combinational designs. ECs should contain code to perform k-induction [22] or a similar technique. We would need to generate sequential code to test that.
(3) It is possible that using entirely input-free programs "obscures" parts of the EC from the fuzzer. If the EC performs any kind of simplification on the programs it is given, then our programs will very likely get folded down to a single constant, meaning we would only properly test code in the EC that runs before this simplification.

REFERENCES

[1] Earl T. Barr et al. "The Oracle Problem in Software Testing: A Survey". In: *IEEE Transactions on Software Engineering* 41.5 (May 2015), pp. 507–525.

[2] Scott Bauer, Pascal Cuoq, and John Regehr. "Deniable Backdoors using Compiler Bugs". In: *PoC GTFO* 9 (2015). URL: https://www.alchemistowl.org/pocorgtfo/pocorgtfo08.pdf.

[3] Junjie Chen et al. "A Survey of Compiler Testing". In: *ACM Computing Surveys* 53.1 (Feb. 2020), pp. 1–36. ISSN: 1557-7341.

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

[5] Alastair F. Donaldson et al. "Test-case reduction and deduplication almost for free with transformation-based compiler testing". In: *PLDI '2021*. ACM, June 2021.

[6] Riccardo Felici, Laura Pozzi, and Carlo A. Furia. "HyperPUT: Generating Synthetic Faulty Programs to Challenge Bug-Finding Tools". In: (Sept. 2022). arXiv: 2209.06615 [cs.SE].

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

[8] Yann Herklotz et al. "An Empirical Study of the Reliability of High-Level Synthesis Tools". In: *29th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2021, Orlando, FL, USA, May 9-12, 2021*. IEEE, 2021, pp. 219–223.

[9] Yann Herklotz et al. "Formal verification of high-level synthesis". In: *Proc. ACM Program. Lang.* 5.OOPSLA (2021), pp. 1–30.

[10] "IEEE Standard for Standard SystemC Language Reference Manual". In: ().

[11] 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).

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

[13] Wilayat Khan et al. "CoCEC: An Automatic Combinational Circuit Equivalence Checker Based on the Interactive Theorem Prover". In: *Complex.* 2021 (2021), 5525539:1–5525539:12.

[14] 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).

[15] 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).

[16] Vu Le, Mehrdad Afshari, and Zhendong Su. "Compiler validation via equivalence modulo inputs". In: *ACM SIGPLAN Notices* 49.6 (June 2014), pp. 216–226. ISSN: 1558-1160.

[17] Christopher Lidbury et al. "Many-core compiler fuzzing". In: *ACM SIGPLAN Notices* 50.6 (June 2015), pp. 65–76. ISSN: 1558-1160.

[18] LLVM/Clang. *UndefinedBehaviorSanitizer*. 2023. URL: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html (visited on 11/15/2023).

[19] Andreas Lööw. "Lutsig: a verified Verilog compiler for verified circuit development". In: *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*. Ed. by Catalin Hritcu and Andrei Popescu. ACM, 2021, pp. 46–60.

[20] William M McKeeman. "Differential testing for software". In: *Digital Technical Journal* 10.1 (1998), pp. 100–107.

[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] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. "Checking Safety Properties Using Induction and a SAT-Solver". In: *FMCAD '2000*. Springer Berlin Heidelberg, 2000, pp. 127–144.

[23] Siemens. *C++/SystemC/RTL Formal*. (Visited on 11/16/2023).

[24] Chengnian Sun et al. "Perses: Syntax-Guided Program Reduction". In: *ICSE '2018*. ACM, May 2018.

[25] Dominik Winterer, Chengyu Zhang, and Zhendong Su. "Validating SMT solvers via semantic fusion". In: *PLDI '2020*. Ed. by Alastair F. Donaldson and Emina Torlak. ACM, 2020, pp. 718–730.

[26] Xuejun Yang et al. "Finding and understanding bugs in C compilers". In: PLDI '11. San Jose, California, USA: Association for Computing Machinery, June 2011, pp. 283–294. ISBN: 9781450306638.

[27] Chengyu Zhang et al. "Finding and understanding bugs in software model checkers". In: *ESEC/SIGSOFT FSE 2019*. Ed. by Marlon Dumas et al. ACM, 2019, pp. 763–773.