Automating Datapath Verification and Bug Correction via Equality Saturation

Emiliano Morini*, Samuel Coward*, Theo Drane*, Rafael Barbalho*, George A. Constantinides[$]
*Intel Numerical and System Level Design Group, Intel, USA
$ Electrical and Electronic Engineering, Imperial College London, UK
Email: samuel.coward@intel.com

Abstract - **In this paper we present a word-level RTL rewriting framework based on equality saturation which facilitates exploration of equivalent designs. By applying rewrites to a single data structure containing the given specification and implementation, we can automatically bridge the gap between the two designs. In the case where the given designs are equivalent, the framework can be leveraged to automatically decompose the proof, greatly simplifying the equivalence checking problem. If there is a bug in the implementation, the framework can be used to automatically propose a minimal fix, without substantially modifying the optimized implementation. Our experiments have demonstrated valuable results, converting inconclusive equivalence checking problems to conclusive ones, and speeding up the run-time of converging cases by up to 6x. For cases when a bug is identified in the implementation, the framework produces a variant of the design containing a candidate fix.**

I. Introduction

The performance and correctness of arithmetic datapath circuits like adders and multipliers are particularly important. Unfortunately, these two requirements are often in conflict. Designers optimizing to improve power, performance and area (PPA) resort to increasingly complex implementations, making the verification ever more difficult. Moreover, due to the size of the input space, exhaustive simulation is infeasible and Formal Verification (FV) is the only option to prove the correctness of these designs.

One of the most successful FV approaches to verify datapath circuit designs is based on Transaction Equivalence Checking (EC), where the design under test is proven to be equivalent to a golden reference design. For many simple designs, EC often converges out of the box, thanks to the commercial tools developed by the EDA companies. One of the solver components of these tools is a rewrite engine. In [1], for example, the application of rewrites is driven mostly by heuristics which can fail to prove the objective.

In this work, we present an RTL rewriting framework that provides an orchestration layer on top of a formal equivalence checking tool. In this framework, the rewrites are close to the user, operating directly on the RTL rather than relying (only) on the solvers' API, allowing us to tailor our high-level rewriting rules according to the problem we are tackling. Based on this work, we have developed two applications.

- **Roverify** is a tool that can assist equivalence proof convergence by finding a sequence of rewrites that can close the gap, decomposing the equivalence checking problem into a sequence of simpler checks.
- **Roverifix** is a tool that takes a specification and a buggy implementation and automatically generates a fix, whilst maintaining the implementation's optimizations.

The first part of the paper summarizes what we presented in [2], including the Roverify results, while the Roverifix application is completely new.

II. Background

A. Equivalence Checking

One of the most successful approaches to verify datapath components is based on Equivalence Checking (EC), a FV methodology which compares the design under test (*implementation*) against a golden reference model (*specification*). Implementation and specification can have difference latencies, and the comparison is performed on a generic transaction, a computation which takes some input values and produces output results, which might have different length in each of the two designs. The output of the comparison can be pass, when a property is proven, fail, when the property is not true (a counterexample is generated), or inconclusive, when the tool does not manage to either prove or disprove a property.

A trusted reference is fundamental for EC. One standard verification flow used in the semiconductor industry is the following: starting from a component specification, a developer writes a high-level reference implementation (in C++ or in "vanilla" RTL) without any interaction with the designer who writes the RTL implementation, providing diversity and independence between the two, which are then formally tested for equivalence. Another common option is to use a trusted version of the same design in RTL as reference, usually a version from previous projects or based on a third-party library. This is usually described as C2RTL, RTL2RTL or C2C EC, depending on the language used to write implementation and specification.

In real-life, inconclusive EC results are very common and they require advanced techniques to achieve full convergence. This is often a manual task which can be very time consuming, as it is necessary to understand the differences between the implementation and specification. When the differences between the two designs are identified, one common approach is to introduce intermediate models and prove equivalence between those and the original designs, as shown in Figure 1. If all the intermediate equivalence steps are proven, the equivalence between specification and implementation holds. The validity of the waterfall can be proven by formulating an assume guarantee lemma, which proves the equivalence of the specification and implementation, assuming each intermediate equivalence holds. The approach we describe here automatically generates these waterfalls, simplifying the task for FV engineers.

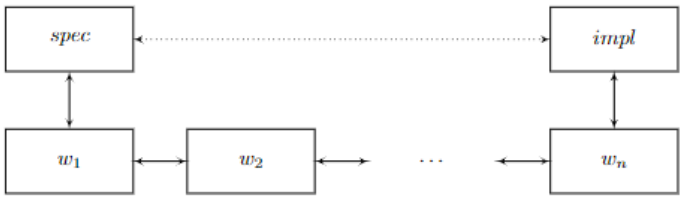

Figure 1: Overview of the waterfall approach used by FV engineers. The dashed line between $\boldsymbol{spec}$ and $\boldsymbol{impl}$ represents an inconclusive verification. Full equivalence is achieved introducing $\boldsymbol{n}$ intermediate designs $\boldsymbol{w_i}$ and proving the equivalences of all the pairs $(\boldsymbol{spec}, \boldsymbol{w_1}), (\boldsymbol{w_1}, \boldsymbol{w_2}), \ldots, (\boldsymbol{w_n}, \boldsymbol{impl})$.

One of the key motivations for this work derives from an overview of the technology behind industry leading tools. In [1], the tool orchestrates a suite of techniques and solvers to prove the equivalence of input designs. One of these techniques is a set of rewrite engines. The authors state that the rewrite engines are driven by heuristics. Heuristic driven search is often brittle and may not explore the required space. The techniques presented in Section III describe a rewrite orchestration approach that does not suffer from these limitations. Here we solely focus on EC, but the rewriting techniques described may be equally applicable to formal property verification of datapath properties.

B. E-graphs

E-graphs cluster equivalent expressions into e(quivalence)-classes, enabling a compact representation of alternative but functionally identical implementations. In the e-graph, nodes represent variables, constants or operators that point to children e-classes. This captures the intuition that we may choose how to implement a given sub-expression at any point in the design. Due to these nested choices, an e-graph can represent exponentially many implementations in the number of nodes.

An e-graph is grown via constructive application of local equivalence preserving rewrites, $l \rightarrow r$, where the right-hand side of the rewrite is added to the e-class containing l, without removing l as would be done in a traditional rewrite engine. As a result, the e-graph avoids the phase-ordering problem, where the order of application impacts the results. This approach to growing an e-graph is known as equality saturation [4]. A simple e-graph rewriting example is shown in Figure 2, where the dashed boxes represent e-class boundaries, and the arrows connect nodes to their child e-classes. The e-graph data structure has been used in the formal methods community for many years [5] and can be found in modern SMT solvers such as Z3 [6]. We build on the extensible egg e-graph library [4] to represent datapath circuit designs in RTL and take advantage of the e-graphs' ability to explore equivalent designs efficiently.

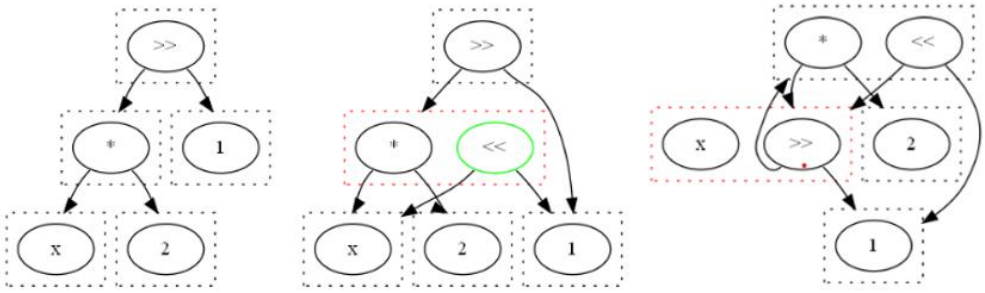

Figure 2: Simple e-graph rewriting over the integers. The dashed boxes represent e-classes of expressions, where we have highlighted the modified e-class in red at each stage. Green nodes represent newly added nodes.

III. METHODOLOGY

In this section we will first introduce the core e-graph rewriting and analysis technology, implemented in Rust, that provides the foundations for the Roverify verification assistant and Roverifix bug fixing tool. We will then describe how we modify the e-graph initialization and extraction process to build a custom tool for each application. The current implementation of the framework handles only System Verilog, so we will assume that all the EC problems discussed below are RTL2RTL comparisons. Sections III and IV summarize work initially presented at the 23rd Conference on Formal Methods in Computer Aided Design [2]. Section V builds upon this framework with a novel approach to determining a minimal bug fix via RTL rewriting.

A method to construct an e-graph representation of RTL was described in [3], encoding all signal bitwidth and signage definitions. We use a nested S-expression intermediate language called VeriLang, as in Common Lisp:

```
term::=(operator [term] [term] … [term])
```

For example, the unsigned multiplication of two 8-bit inputs a and b, stored in a 16-bit result is expressed as:

```
(* 16 unsigned 8 unsigned a 8 unsigned b)
```

Based on this intermediate language the e-graph can represent the functional behavior of combinational RTL designs. We process input System Verilog using the open-source Slang parser [9], combined with a custom translation to the intermediate language. We parse both RTL designs and generate expressions, S and I, in the intermediate language, for the specification and implementation respectively. We pass the VeriLang expressions to egg [4], which initializes an e-graph containing a single node per e-class. The e-graph initialization shall differ for each application, so we defer additional discussion to Sections IV and V. An e-graph containing two designs is shown in Figure **3**.

```
module spec(A,B,M,N,O);
  input  [15:0] A, B;
  input  [3:0]  M, N;
  output [62:0] O;
  wire   [30:0] D;
  wire   [30:0] E;

  assign D = A << M;
  assign E = B << N;
  assign O = D * E;
endmodule
```

```
module impl(A,B,M,N,O);
  input  [15:0] A, B;
  input  [3:0]  M, N;
  output [62:0] O;
  wire   [31:0] C;
  wire   [4:0]  P;

  assign C = A * B;
  assign P = M + N;
  assign O = C << P;
endmodule
```

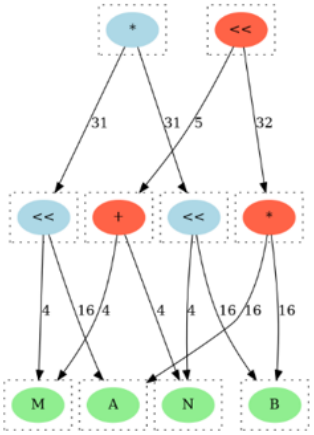

Figure 3: Initial e-graph representing a specification (blue) and implementation (red). Shared nodes are colored green. Edge labels denote bitwidths. All e-classes (dashed boxes) initially contain a single node.

Rewrites are the way in which the e-graph adds new equivalent designs. In [3] we defined a set of bitwidth dependent RTL rewrites for area optimization. The verification rewrite set includes the optimization capabilities but adds additional verification specific rewrites to essentially reverse optimizations.

Table 1 gives examples of the verification rewrites which are guided by our experience using commercial EC tools. These rewrites are hard to consider as optimizations. The space of "unoptimizations" is not well defined, so rewrite selection is non-obvious, and mostly driven by experience and test cases. The intention is that users should be able to easily add additional rewrites that are applicable to their designs.

To maximize rewrite application, we ensure that the left-hand side of a rewrite matches any possible type combinations. We then filter these candidates to valid rewriting opportunities via conditional rewriting. The conditions are necessary and sufficient as a function of the rewrite parameters. The sufficiency of the condition ensures that only valid, equivalence preserving, rewrites are applied. The necessity of the condition guarantees that no rewriting opportunities are missed. Missed rewriting opportunities lead to brittleness and over-dependence on metrics which can be the difference between a proven equivalence check and an inconclusive result.

Table 1: An example set of bitwidth dependent datapath verification rewrites. All rewrites are conditionally applied to ensure correctness. Bitwidth and signage information of operators and operands is omitted here for concision.

Name	Left-Hand Side	Right-Hand Side
Unmerge Left Shift	$a \ll (b + c)$	$(a \ll b) \ll c$
Mult Left Shift	$a \times (b \ll c)$	$(a \times b) \ll c$
Shift to Mult	$a \ll const$	$a \times 2^{const}$
Mult to Add	$a \times 2$	$a + a$

Bitwidth definitions can impact the functional behavior of RTL, for example the addition of two 8-bit values stored in an 8-bit and a 9-bit result differ in general but may be equivalent in a particular context. To reduce all signals to their minimal bitwidth we use interval analysis and reduction rewrites (described in [7]), taking advantage of egg's e-class analysis feature. Some commercial ECs also utilize a form of interval analysis [1], but e-graph analyses benefit from more precise abstractions [8].

We have described the underlying framework comprised of an intermediate language, front and back ends for translation to and from System Verilog and a set of rewrites and analyses. On top of this framework, we develop a pair of applications, Roverify and Roverifix, that differ in how we initialize the e-graph and how we extract designs from it. In both cases we apply rewrites to the e-graph, adding new nodes to the e-graph and growing the e-classes. We vary the number of e-graph rewriting iterations to control the e-graph growth throughout this work. Constructive rewrite application adds the overhead of maintaining many equivalent representations of the two designs in the e-graph, but greatly simplifies the problem of determining a correct rewrite application order.

IV. ROVERIFY

The first problem we solve, is the following. Given two Verilog designs, a Specification (Spec), and an Implementation (Impl) that we believe to be equivalent. Generate a set of hints to assist the EC tool in proof convergence. Figure 4 describes the overall flow of Roverify. Both the Spec and Impl are translated to VeriLang (S and I respectively) and passed to egg [4] which initializes an e-graph containing both designs. An example of the initialized e-graph is shown in Figure **3**. Roverify then applies the rewrites and analysis as described above, growing the e-graph. The objective is to apply sequences of rewrites to discover equivalent intermediate signals in the e-graph. Intuitively we apply rewrites to bridge the gap between the two designs. The e-graph rewriting progress is shown in Figure **5**.

Once Roverify halts rewriting, it determines whether it was able to discover any shared intermediate signals between designs equivalent to the Spec and designs equivalent to the Impl. From the rewritten e-graph, Roverify extracts two designs, $S^* \cong S$ and $I^* \cong I$, such that S^* and I^* share the maximal number of intermediate signals. Roverify then generates a new equivalence checking problem, converting S^* and I^* to Verilog, which is passed to the EC tool. This new problem should be simpler to solve as the two designs in question share more common intermediate signals. To guarantee that S^* and I^* are indeed equivalent to S and I, respectively, Roverify traces the rewrite sequences from S to S^* and I to I^*, as shown in Figure **4**. Each rewrite step is checked using the EC tool, generating Verilog before and after the rewrite and checking the results. These rewrites are small transformations, which are generally trivial for the EC tool to prove.

If Roverify can find a complete sequence of rewrites between the two designs, then the root e-classes of the Spec and Impl will merge, such that all nodes become "shared", as in Figure **5**. If this scenario is reached, rewriting halts and Roverify extracts identical S^* and I^*, making this equivalence check trivial.

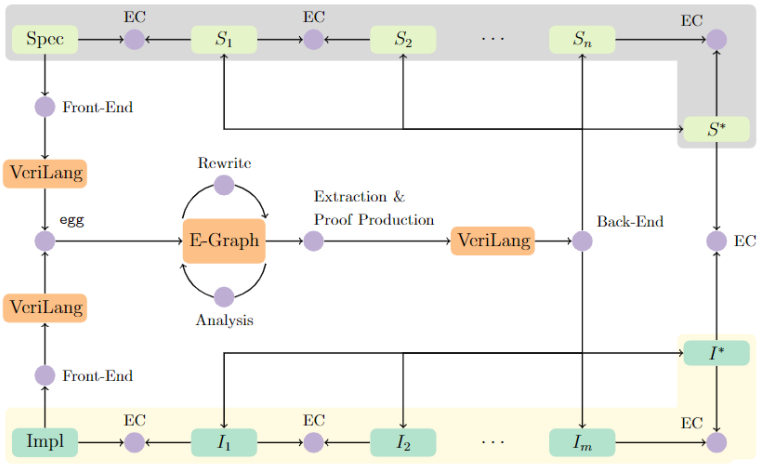

Figure 4: Flow diagram describing the automated equivalence checking assistant Roverify. Lighter green nodes denote designs known to be equivalent to the specification, darker green nodes denote designs known to be equivalent to the implementation.

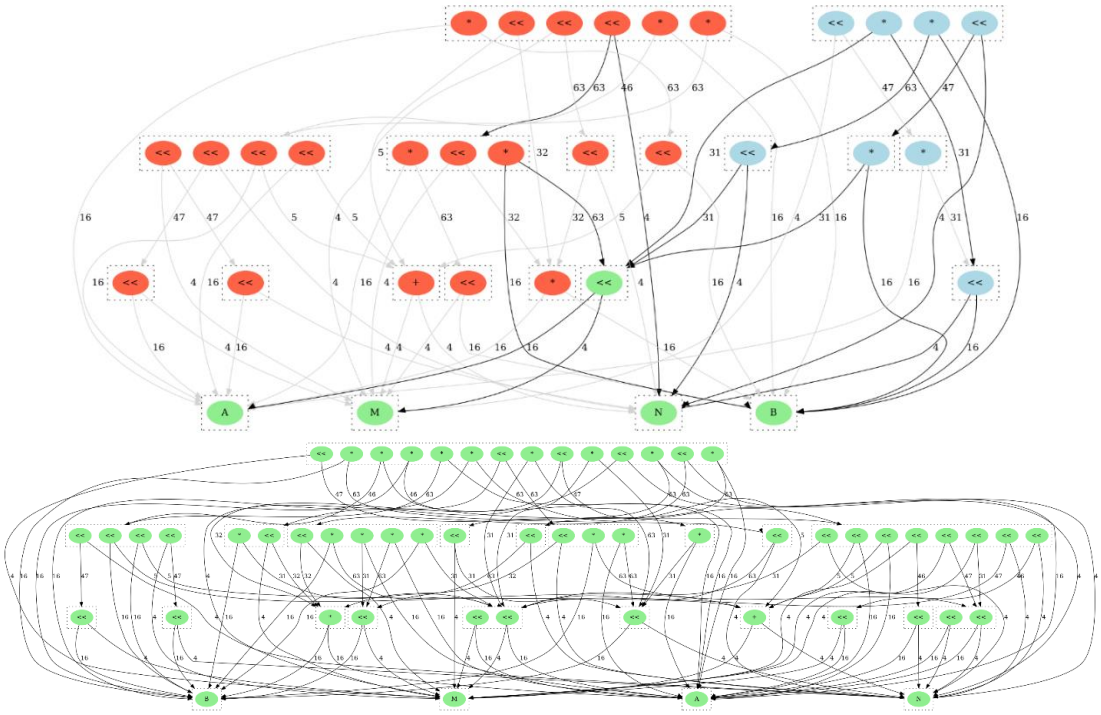

Figure 5. Stages of e-graph growth starting from the initial e-graph in Figure 3.

V. ROVERIFIX

In contrast to Roverify, which assists existing tools with proof convergence. Roverifix goes beyond existing tools and can automatically provide bug fixes for broken implementations. More precisely, given a Spec and Impl that are proven not equivalent, find the minimal fix or fixes to the implementation such that the two designs are equal. We target a minimal fix because an implementation typically contains optimizations that it is desirable for the fixed design to retain. This is similar to the intent behind corrections applied for an engineering change order (ECO).

An important motivation for this tool is that understanding how to fix a datapath component could be non-trivial, even when a counterexample is available. Verilog and System Verilog have some interesting subtleties which might

seems confusing at first sight, as shown in Figure 6, the result of very simple operations is not what many engineers would guess [10], hence fixing the implementation might take longer than expected.

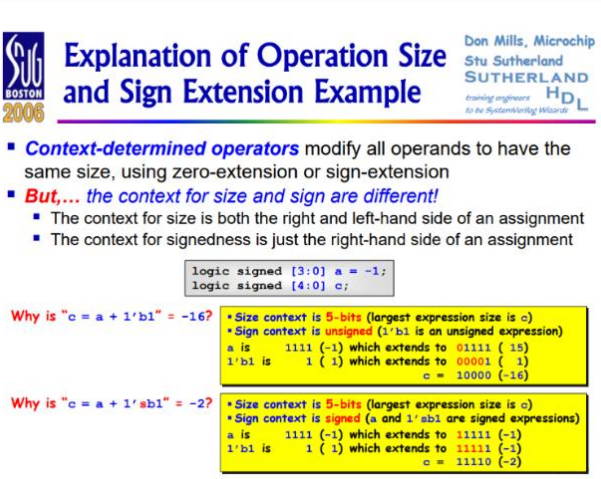

Figure 6: Examples of Verilog operators gotchas [10].

Another very common problem is “datapath leakage”, that occurs when an internal operand is not wide enough to store the exact result of an operation, as illustrated in Figure 7.

```
1  module spec(A,B,C,out);
2  input  logic [7:0] A, B, C;
3  output logic [9:0] out;
4  wire  [8:0] add_right;
5
6  assign add_right = B + C;
7  assign out       = A + add_right;
8
9  endmodule
```

```
1  module impl(A,B,C,out);
2  input  logic [7:0] A, B, C;
3  output logic [9:0] out;
4  wire  [7:0] add_8bit;
5
6  assign add_8bit  = A + B; // carry-out discarded
7  assign out       = add_8bit + C;
8
9  endmodule
```

Figure 7: Specification and implementation, which differ in functionality due to a bug in the implementation.

The bug in impl is caused by an incorrect bitwidth definition of the add_8bit signal, leading to the carry-out of the addition being discarded. Roverifix identifies and corrects this bug by first rewriting the spec using associativity of addition. Then propagates custom FIX operators (described next) through both designs to identify a minimal fix, in this case increasing the bitwidth of add_8bit by 1.

Figure 8 describes the general Roverifix flow. Roverifix initializes a first e-graph only with VeriLang derived from the Spec, as we wish to explore the space of designs equivalent to the Spec. This first e-graph is rewritten using the same set of rewrites as Roverify, constructing a set of candidate fixes. From this set of candidates, Roverifix extracts the design that is syntactically closest to the Impl. To achieve this Roverifix adds the following VeriLang expression to the e-graph, FIX(S,I). This FIX node encodes a correction, namely replacing I by S, is a correction, albeit a large modification. A second phase of rewriting is applied to this e-graph using only a new set of rewrites, described in

Table 2. These rewrites push FIX nodes down the expression trees of Spec and Impl, where the objective is to share as much as possible. The deeper we can force the FIX nodes the smaller the corresponding correction.

Table 2: FIX propagation and removal rewrites. We propagate FIX nodes over any VeriLang operator and remove FIX nodes when the suggested FIX is to replace something with itself.

Name	Left-Hand Side	Right-Hand Side
Fix over Operator	`FIX(a op b, c op d)`	`FIX(a,c) op FIX(b,d)`
Fix Same	`FIX(a,a)`	`a`

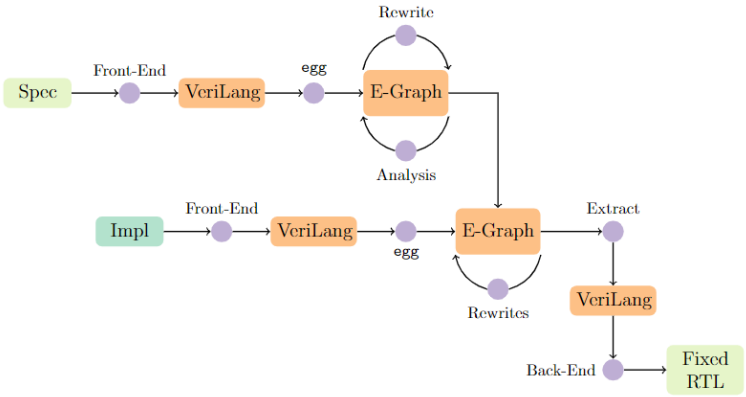

Figure 8: flow diagram describing the automated bug fixing assistant, Roverifix. The generated Fixed RTL is functionally equivalent to the specification but is the minimal change from the implementation. We highlight correct RTL in light green and buggy RTL in a darker green.

Once the FIX nodes have been propagated through the second stage e-graph, an extraction phase is run to determine the minimal FIX nodes required to correct the design. The extracted VeriLang is converted to Verilog, taking the left argument of each FIX node, as this corresponds to a design that is equivalent to the specification. Roverifix can also report back to the user precisely where the fixes were applied. The extracted VeriLang for the example above would contain a single FIX operator, FIX(9,8), corresponding to the need to increment the bitwidth of the add_8bit signal as the sole correction. If there is no bug and via the initial rewriting, we discover that $S \cong I$, then we remove all FIX nodes from the e-graph via the "Fix Same" rewrite.

VI. Results

In all the following results we use an up-to-date version of an industry standard EC tool running on SLES 12 on Intel Xeon W-2155 CPUs.

A. *Roverify*

Open-source benchmarks are taken from [11]. We implement original and optimized RTL for these designs. We include two instances of a kernel from the H.264 VBSME, corresponding to summation trees of size four and eight, $\sum_i |a_i - b_i|$. The Denorm Mult and box filter are Intel provided benchmarks. The box filter is a reconfigurable square filter, sampling four pixels at a time. For each benchmark, we run Roverify until either, it discovers a complete path between specification and implementation, or it deploys five iterations of rewriting. The e-graph rewrites different parts of the design in parallel. We did not see any increase in the EC tool compilation times. The runtimes are reported from when the solvers start running. The baseline, deploys all tool solve scripts in parallel, taking the fastest reported result. We run all Roverify generated proofs in parallel and report the maximum time taken to solve a single sub-problem. In practice, the multi-processor environment introduced runtime overhead unrelated to solving the proof.

Table 3 demonstrates the benefit of Roverify, resolving an inconclusive proof and reducing the total verification time. For the ADPCM Decoder, the EC tool already efficiently proves the correctness of the two designs, making the Roverify runtime overhead overall detrimental. It is worth noting that the intermediate proofs do help reduce the solve time. In the Denorm Mult example, when passed the original EC problem with no assistance, the EC tool did not return a result within 24 hours. In this case, Roverify is able to convert an inconclusive proof into one solved in less than a second. The assistant generated up to 115 intermediate proofs and 42 on average.

Table 3: EC tool performance with and without intermediate proofs generated by the assistant. We report the baseline EC tool performance when solving the original EC problem. We also report the runtime of Roverify and the runtime of the EC tool when solving the problem with the intermediate proofs. The sum provides a total verification time for the assisted proof. The last column shows the speedup ratio achieved using Roverify. Runtimes are in seconds.

Benchmark	DPV without Assistance	Roverify	DPV with Assistance	Assisted Total	Speedup (without/with)
ADPCM Decoder	**0.68**	0.38	0.49	0.87	0.78
H-264 VBSME-4	7.93	7.04	0.71	**7.75**	1.02
H-264 VBSME-8	93.13	14.30	0.20	**14.50**	6.42
FIR Filter	5.50	3.49	0.79	**4.28**	1.29
Box Filter	79.56	16.10	1.61	**17.71**	4.49
Denorm Mult	-	0.14	0.10	**0.24**	-

In all other benchmarks, Roverify is a net positive. The introduction of intermediate proofs reduces the EC tool solve time by up to 465x (when we ignore the Roverify runtime). Including Roverify, total verification time is reduced

by up to 6x. For each intermediate proof Roverify can tailor the most optimal solver orchestration script, which greatly helps performance. Such an advantage can only be gained because Roverify understands the rewrites applied. We do not run multiple solvers in parallel.

B. Roverifix

We have conducted a more limited evaluation of Roverifix, looking at a few examples. We first apply Roverifix to the example shown in Figure 7. Roverifix identifies the correction needed in under 0.12 seconds reporting back the FIX, updating the bitwidth of a single signal. To demonstrate real-world value, we applied Roverifix to an internally developed floating point norm component calculating $\frac{1}{\sqrt{1+x^2}}$. Simulation discovered that the component incorrectly processed input NaNs. We passed a correct specification and the buggy implementation to Roverifix, which correctly identified a FIX inserting a mux to correctly handle NaNs. On this example Roverifix runs in 0.19 seconds.

There is no limit on how many bugs Roverifix is able to correct in a single run, however there is also no limit on the size of a single correction. Namely, if Roverifix is unable to propagate FIX nodes through the e-graph then it will propose a potentially large correction, simply replace the implementation with the specification. We measure the size of the correction by the increase in circuit area of the corrected design over the implementation. Roverifix minimizes this distance to avoid proposing the direct replacement of the implementation with the specification.

VI. Conclusions

In this paper, we have presented an RTL rewriting and analysis framework, leveraging the e-graph data structure to mitigate the phase ordering challenges. On top of this framework, we developed a pair of applications. First, Roverify, an assistant for datapath equivalence checking tools. Roverify decomposes the datapath equivalence checking problem into a sequence of independent checks, which are easier for the EC tool to solve. Second, Roverifix, a bug fixing tool that can automatically generate a minimal correction for a buggy implementation. We demonstrated that both tools have a meaningful impact on their respective challenges via several benchmarks. The underlying framework provides a solid foundation for a range of applications, as RTL rewriting is an essential component of many EDA tools. The applications themselves can reduce the burden on both FV engineers and RTL designers, helping them to prove equivalence faster and fix bugs with minimal manual intervention. More broadly, the paper demonstrates the value of manual or automated rewriting in assisting FV tools to achieve convergence and remove bugs.

We are currently exploring how to make the tools more widely available and hope to integrate this technology into a complete datapath equivalence checking tool, since the existing interface between the tools is a barrier to greater performance. A particularly exciting direction would be to integrate SAT/SMT solvers into Roverify to bridge gaps beyond the limits imposed by our rewrite sets. For Roverifix, we could explore how equivalence checking counterexamples can guide the search for a correction. We have also yet to address how best to fix operator replacement bugs, where for example a greater than or equal was used in place of a greater than. A more general improvement would be adding support for C/C++ to consider specifications defined in higher-level languages.

References

[1] A. Koelbl, R. Jacoby, H. Jain, and C. Pixley, “Solver technology for system-level to RTL equivalence checking,” in Proceedings -Design, Automation and Test in Europe, DATE, 2009.

[2] Coward, S., Morini, E., Tan, B., Drane, T., & Constantinides, G. A. (2023, October). Datapath Verification via Word-Level E-Graph Rewriting. In 2023 Formal Methods in Computer-Aided Design (FMCAD) (pp. 92-100). IEEE.

[3] Coward, S., Constantinides, G. A., & Drane, T. (2022, September). Automatic datapath optimization using e-graphs. In 2022 IEEE 29th Symposium on Computer Arithmetic (ARITH) (pp. 43-50). IEEE.

[4] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, “Egg: Fast and extensible equality saturation,” in Proceedings of the ACM on Principles of Programming Languages, vol. 5, no. POPL, 2021.

[5] C. G. Nelson, “Techniques for program verification,” Ph.D. dissertation, Stanford University, 1980.

[6] L. De Moura and N. Bjørner, “Z3: An efficient SMT Solver,” in Lecture Notes in Computer Science including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 4963 LNCS. Springer, 2008.

[7] Coward, S., Constantinides, G. A., & Drane, T. (2023). Automating Constraint-Aware Datapath Optimization using E-Graphs. In 2023 60th ACM/IEEE Design Automation Conference (DAC) IEEE.

[8] Coward, S., Constantinides, G. A., & Drane, T. (2023, June). Combining E-Graphs with Abstract Interpretation. In Proceedings of the 12th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (pp. 1-7).

[9] M. Popoloski, “Slang,” 2023. [Online]. Available: https://github.com/MikePopoloski/slang

[10] S. Sutherland, D. Mills, “Standard Gotchas Subtleties in the Verilog and SystemVerilog Standards That Every Engineer Should Know”, SNUG Boston 2006.

[11] A. K. Verma, P. Brisk, and P. Ienne, “Data-flow transformations to maximize the use of carry-save representation in arithmetic circuits,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 27, no. 10, pp. 1761–1774, 2008.