Applying Automated Program Repair to Dataflow Programming Languages

Yu Huang
University of Michigan
yhhy@umich.edu

Hammad Ahmad
University of Michigan
hammada@umich.edu

Stephanie Forrest
Arizona State University
steph@asu.edu

Westley Weimer
University of Michigan
weimerw@umich.edu

***Abstract*—Dataflow programming languages are used in a variety of settings, and defects in their programs can have serious consequences. However, prior work in automated program repair (APR) emphasizes control flow over dataflow languages. We identify three impediments to the use of APR in dataflow programming—parallelism, state, and evaluation—and highlight opportunities for overcoming them.**

***Index Terms*—Automated program repair, dataflow programming languages, parallelism**

I. Introduction

Unlike control flow programming languages (e.g., imperative languages, such as C/C++, Java, or Python), where a program is modeled as a series of operations occurring in a specific order, *dataflow* programming languages (also called datastream languages), model a program as a directed graph of the data flowing between operations. Originally designed to exploit parallelism, modern dataflow languages have since been applied to many different application domains. For example, *Verilog* (absorbed into the SystemVerilog Standard in 2009) and *VHDL* are hardware description languages (HDLs) in the dataflow language paradigm that are widely used in both academia and industry for circuit design. Another example is *Lustre*, an earlier dataflow language for programming reactive systems, which is now used for critical control software in aircraft and nuclear power plants. Meanwhile, graphical and visual data flow languages, such as *LabView* and *Simulink*, have been developed to support system design and simulation. These languages and tools are particularly exploited in non-IT industries and have been adopted by engineers without professional computer science or programming backgrounds. Dataflow programming also plays an increasingly important role with the growth of machine learning and artificial intelligence: the popular *TensorFlow* library for neural networks is based on dataflow programming.

Although dataflow programming has been deployed to support a global market worth billions of dollars (e.g., the global integrated circuits market alone was $412.3 billion in 2019 [1]), and defects in dataflow-based software systems can cause severe consequences (e.g., the Pentium FDIV bug [2]), automated and evolutionary approaches to support this category of software are understudied. By contrast, a significant amount of research has shown how genetic improvement (GI) based techniques, such as automated program repair (APR), can repair bugs [3] in real-world imperative programs, scaling to millions of lines of code and thousands of test cases. Fortunately, dataflow programming shares some common characteristics with control flow programming. For instance, testing and debugging are critical processes for both paradigms. Sequential statements can appear in both dataflow (e.g., Verilog) and control flow programming, suggesting the possibility of repurposing GI techniques developed for control flow programs for dataflow programs. However, the special structure of dataflow languages also raises a number of novel challenges and opportunities.

```
reg a [1:0] = 2'b01;
reg b [1:0] = 2'b10;
reg c [3:0] = 4'b0000;
reg d [3:0] = 4'b0000;

always@(posedge clock) begin  //executes on clock
c <= a + b;         // statements in parallel
d <= c + a;         // after the clock rising c = 4'0011
end                 // and d = 4'0001;
```

Fig. 1. Non-blocking assignments in the Verilog dataflow language are denoted with <= and are executed in parallel.

In this paper, we propose to apply the insights developed for control flow GI to the problems of testing, repairing and improving dataflow programs. We anticipate that GI techniques can reduce the maintenance costs of, and improve the quality and efficiency of the design and maintenance process for, dataflow programming. We highlight the similarities and differences between imperative languages and several commonly-used dataflow languages, and outline potential benefits, challenges, and solutions arising from our comparison.

II. Challenges and Opportunities

We identify and discuss three key challenges for automatically repairing bugs in dataflow programs: parallelism, state, and evaluation. We conclude with initial solution directions.

A. Parallelism

In control flow programming, instructions and branches are typically executed sequentially. Many approaches to pinpointing the region of the program implicated in a defect (fault localization [4]), are predicated on contrasting the instructions executed on conforming executions with those executed on failing ones. GI methods search the space of edits and patches near that region to produce program variants, and then evaluate each variant with respect to a test suite until one is found that meets all requirements. However, in dataflow programming, statements are often executed in parallel. Examples of such parallelism are concurrent statements in Verilog (e.g., non-blocking assignments, as shown in Figure 1 [5]), and nodes

in TensorFlow representing operations (together with edges representing the dependencies of nodes) that can be executed simultaneously. Traditional fault localization is much less precise in the face of massive parallelism and thus less applicable to dataflow languages (e.g., informally, if all parts of a circuit described in Verilog execute at all times, comparisons between executed lines are unrevealing).

B. State

In control flow languages, imperative formalisms are used to describe program states and explicit transitions between states. However, dataflow programming may abstract state or transitions. For example, in Lustre, programs can be described using state variables and the strongest invariant property of each state variable. These asserted invariants then primarily direct the compiled order of operations, rather than being checked for correctness per se [6, Sec. II-B]. Traditional program repair approaches to leverage `assert` statements may not apply directly. Another example of the difference in describing program states can be seen in the programming model used in LabView, where execution is based on data availability [7]. In these settings, traditional imperative analyses (e.g., constraint solving [8]) may not be directly relevant, complicating the adaptation of concepts from current APR techniques.

C. Evaluation

Test suites are commonly used to assess control flow programs, and APR methods often use them to assess candidate repairs. Testing is also important for dataflow programming, but the testing process, or model, is often quite different from control flow programming. For instance, Verilog applies a testbench-based process, more historically related to hardware, where the device under test (DUT) is tested in a separate environment. In this testing environment, multiple inputs are typically combined sequentially to check the functionality of the DUT, which generates sequential outputs. In this case, HDL engineers visualize the timeseries signals in waveforms to manually check the correctness of the design. This process can be both time-consuming and error-prone. To apply APR techniques to dataflow languages, we must adapt either the algorithms or the testing process (i.e., the fitness function).

Another challenge for applying APR to dataflow is efficiency. Many dataflow programs include scientific computations (e.g., physical calculations in Verilog, mathematics in TensorFlow, etc.) that may strain search budgets.

D. Discussion

Although parallelism complicates fault localization, it may provide the opportunity to reduce the search space by exploiting certain characteristics of control flow programming. For example, there is potential to leverage the flow of information of faulty data or signals and narrow down candidate buggy statements or components based on data dependencies (e.g., "netlist" in Verilog). This feature of dataflow languages also lends itself to other techniques, such as retrograde analysis [9], for bug localization.

Earlier research on using GI for parallelization, albeit not directly focusing on dataflow languages, has shown some promise (e.g., GB-GP-Parallelisation reduces the execution time of software on a GPU by 2.60% on average [10]), suggesting that exploration of GI-based techniques for dataflow is warranted. Further, although still expensive to evaluate, many dataflow programs (e.g., HDLs, LabView) do not face an infinite input space. Instead, the design requirements of such systems often imply a limited set of inputs, which may admit techniques like model checking [11] to reduce the testing and state space problems. Supporting tools exist to apply imperative-style analyses to dataflow languages (e.g., *Pyverilog* for Verilog descriptions). Open source dataflow programs are also available in both general repositories (e.g., GitHub) and domain-specific forums (e.g., *OpenCores* for hardware designs).

These advances in bug localization and program analysis suggest that new testing or model checking processes can be combined with existing APR methods to bring GI to the world of dataflow languages. If successful, these new methods can improve development efficiency and effectiveness in the many other industries driven by dataflow programming.

III. Conclusion

Dataflow programming languages are widely deployed in many industries and domains, but we currently lack GI methods to support and improve their design efficiency and effectiveness. In this paper, we highlight the potential of applying program repair to dataflow programming by presenting and discussing the similarities and differences between control flow and dataflow languages, the challenges of this domain, and possible solutions or directions for forward progress.

References

[1] "ICs market: Integrated circuits industry growth," retrieved Jan 11, 2021 from https://www.prnewswire.com/news-releases/ics-market–integrated-circuits-industry-growth-by-most-key-players-intel-toshiba-broadcom-and-qualcomm-301029636.html.

[2] "Pentium FDIV bug," retrieved Jan 11, 2021 from https://en.wikipedia.org/wiki/Pentium_FDIV_bug.

[3] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: a survey," in *International Conference on Software Engineering*, 2018.

[4] J. A. Jones and M. J. Harrold, "Empirical evaluation of the Tarantula automatic fault-localization technique," in *Automated software engineering*, 2005, pp. 273–282.

[5] "Verilog Assignments," retrieved Jan 11, 2021 from https://documentation-rp-test.readthedocs.io/en/latest/tutorfpga05.html.

[6] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.

[7] R. Bitter, T. Mohiuddin, and M. Nawrocki, *LabVIEW: Advanced programming techniques*. CRC press, 2017.

[8] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *International conference on software engineering (ICSE)*, 2016, pp. 691–701.

[9] P. Ohmann, D. B. Brown, B. Liblit, and T. Reps, "Recovering execution data from incomplete observations," in *Proceedings of the 13th International Workshop on Dynamic Analysis*, 2015, pp. 19–24.

[10] B. R. Bruce and J. Petke, "Towards automatic generation and insertion of OpenACC directives," *Research Notes*, vol. 18, no. 04, p. 04, 2018.

[11] J. Edmund M. Clarke, O. Grumberg, D. Kroening, D. Peled, and H. Veith, *Model Checking, Second Edition*. MIT Press.