# VERIFICATION OF STREAMING DESIGNS BY COMBINING SYMBOLIC SIMULATION AND EQUIVALENCE CHECKING

*Tim Todman, Wayne Luk*

Department of Computing
Imperial College London
180 Queen's Gate
London
email: timothy.todman@imperial.ac.uk, w.luk@imperial.ac.uk

## ABSTRACT

As design complexity grows, verification becomes a bottleneck in design development and implementation. This paper describes a novel approach for verifying reconfigurable streaming designs, based on symbolic simulation and equivalence checking. Compared with numerical simulation, symbolic simulation provides a more informative way of showing a design behaved as expected; equivalence checking enables automatic checking of equivalence of symbolic expressions. Our approach has been implemented for designs targeting Maxeler technologies, using an easy-to-use symbolic simulator and the Yices equivalence checker, together with other facilities such as an output combiner to support an automated verification flow. Several benchmarks including, including one-dimensional convolution and finite difference computation, are used to evaluate the proposed approach.

## 1. INTRODUCTION

In recent years, Field-Programmable Gate Arrays (FPGAs) have continued to increase in size, and in some applications have started to replace fixed-function devices like application-specific integrated circuits. Researchers have also used FPGAs in reconfigurable computing applications, where they have proved competitive with both general purpose processors and graphics processing units. This increase in use of FPGAs has led to research into different approaches to programming them. The approaches include:

- Traditional hardware description languages (HDLs) like VHDL and Verilog, which have good vendor and third party support, but lead to large, complex designs;
- C-like approaches, which try to allow software-style programming of FPGAs. Commercial approaches include Catapult-C, AutoESL, and Handel-C;
- Streaming approaches, which have advantages including the natural expression of many traditional reconfigurable computing applications, and naturally pipelined designs. Academic approaches include ROCCC [1], while commercial approaches include Maxeler's MaxCompiler [2], which uses a custom Java subset overloaded with hardware-specific types and operations to compactly describe streaming kernels.

Streaming approaches can improve productivity, but the key challenge remains *verification*: how to ensure that optimized designs preserve the original design's behaviour.

Traditional approaches to verification simulate the reference and optimized designs with a set of test inputs, comparing the outputs. This approach works well, but the test inputs must cover all aspects of the design's behaviour, and can be very large for designs with many possible inputs. For example, it would be infeasible to simulate all input pairs of a 64-bit multiplier. There is always a danger that the test inputs do not cover all the cases, or that the output is only coincidentally correct.

Rather than relying on numerical or logical simulation, our approach combines *symbolic simulation* with *equivalence checking*. Symbolic simulation applies symbols rather than numbers or logic values to the design, the outputs being functions of these symbolic inputs. For example, symbolically simulating an adder with inputs $a$ and $b$ might result in $a + b$. However, for larger designs it is harder to distinguish different but equivalent outputs ($b + a$ instead of $a + b$) from incorrect ones. The equivalence checker tests whether or not the outputs of transformed designs are equivalent to those of the reference design.

Our approach has the advantages that:

- We simulate and validate word-level designs, assuming that arithmetic operations have already been validated by other techniques, with the advantage that any mismatches between source and target are found at word level, rather than at bit level, which can be hard to read for application designers;

- Symbolic simulation means users need not worry about covering full input and output ranges, as with numerical simulation;
- The equivalence checker automatically compares symbolic input and output to ensure the optimized design preserves the behaviour. Unlike numerical simulation, it can reject false negatives due to different but equivalent output that would differ using simple-minded numerical comparison;
- The user can still use numerical simulation to (a) compare with other simulators, such as those from FPGA vendor tools, those from higher-level tools such as Maxeler's MaxCompiler simulator, or results of software implementing the same design, (b) simulate only part of design symbolically to eliminate data-dependent values, allowing application to larger problems, or to isolate problematic corner cases.

This work makes the following contributions:

- We develop an approach for verification of streaming designs by combining symbolic simulation with equivalence checking;
- We implement our approach for Maxeler designs using our symbolic simulator and Yices as an equivalence checker, using a compile scheme to translate the symbolic simulator output into universally quantified expressions for Yices;
- We evaluate our approach on several benchmarks including 1D convolution, and reverse time migration, which is based on finite difference computation.

The paper is structured as follows. The next section presents related work. Section 3 gives an overview of our approach, showing how we verify streaming designs on reconfigurable hardware. Section 4 implements our approach for Maxeler kernel designs, while section 5 gives results and evaluates our approach on several benchmarks. Finally, section 6 concludes and gives ideas for future work.

## 2. RELATED WORK

Partly thanks to some well-publicised bugs in widely used hardware [3], formal verification of hardware designs has received both academic and industrial attention. Industrial tools include the Formality [4] equivalence checker, which works with existing hardware flows to ensure the equivalence of register-transfer level designs with optimized and synthesized netlists.

Academic approaches include the work of Singh and Lilleroth [5], who verify the equivalence of FPGA cores using a model checker, and give some ideas for run-time verification by running the model checker at run-time, which is necessarily restricted to small designs such as adders. Susanto and Melham [6] verify run-time reconfigurable optimizations such as partial evaluation using a theorem prover.

Other researchers have considered verification of properties of discrete event systems (such as freedom from deadlock) by model checking [7], verifying programs running on FPGA-based soft processors [8], verifying declarative parameterized hardware designs with placement information using higher-order logic [9] and verifying that hardware requested at run time implements a particular function using the concept of proof-carrying code [10, 11].

Our approach resembles work on design validation of imaging operations using symbolic simulation and equivalence checking [12]. This work embeds a subset of a C-like language for FPGA design into a theorem prover, using symbolic simulation and an equivalence checker to verify the correctness of transformed designs. Unlike that work, we verify optimizations of streaming designs, with our implementation using Maxeler's MaxCompiler. This means that we must take care to preserve the order of inputs to and outputs from the design. Because Maxeler designs are effectively implemented as Java programs, they allow for metaprogramming using the full power of that programming language. Unlike previous work on verifying other design inputs, we must allow for this in our verification.

## 3. OVERVIEW OF APPROACH

We now give a high-level overview of our approach, showing how an optimized design is verified by comparing it with a reference design using symbolic simulation and equivalence checking. Whilst our approach is implemented for Maxeler designs using a particular symbolic simulator and equivalence checker, the same approach could apply to other design inputs such as C-like languages, and to other symbolic simulators and equivalence checkers.

The core of our approach is to compile a high-level design description to a form that can be processed by a symbolic simulator. The symbolic simulator applies symbolic inputs to the design, resulting in symbolic outputs. Finally, we use a checker to ensure symbolic output is equivalent to original.

Fig. 1 shows our approach, where a reference design (the *source*) is used to compare with an optimized, transformed design (the *target*), in four phases:

1. *Design optimization*: users apply various optimizations manually or automatically [13] to transform a source design to a target, optimized design;
2. *Compilation for simulation*: a compiler compiles both source and target designs into input for a symbolic simulator;

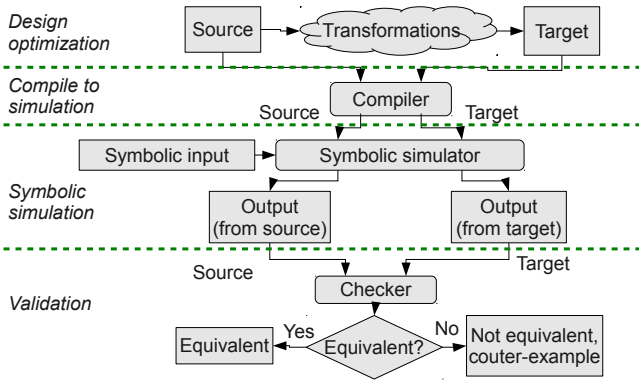


**Fig. 1**. Abstract verification design flow.

3. *Symbolic simulation*: a symbolic simulator applies the same set of symbolic inputs to both source and target designs, yielding symbolic outputs for both;

4. *Validation*: a checker compares the outputs of source and target designs, resulting in either success (source and target designs match), or failure, with a counterexample showing why the designs are not equivalent.

We assume designers optimize their design by starting with a straightforward, but perhaps less optimal design, and optimizing by applying successive transformations to optimize various aspects of the design; the space of transformed designs forms a tree rooted at the straightforward design.

The user only needs to verify the final design against the straightforward design as a reference. If the designs are not equivalent, the user can track down which transformation caused the error by validating intermediate designs, perhaps by manually using an algorithm such as binary search, although we do not expect the transformation space to be large.

Finally, validation can result in a counterexample if the designs are not equivalent; this can be used by the designer to debug the optimization sequence, showing which part of the optimized design is not equivalent to the source.

Note that our abstract approach is highly general and can apply to any sort of input design or any optimization that can be expressed as a transformation. While we apply our approach to streaming reconfigurable hardware designs in this paper, the same approach could be used to validate software designs, hardware designs in other languages, and so on. It could also be used to validate the compilation of one language into another, for example comparing a source design in a software language such as C with the target design being the output of a compiler, for example a VHDL design if we are validating a C to VHDL compiler. If the source and target designs are in different design descriptions, the approach must use different compilers to compile to symbolic simulation, but the abstract approach is otherwise unchanged. This could also be extended to have multiple design descriptions in the source and target, for example having control logic defined in a tool such as Catapult-C, while the high-throughput datapath is described in a tool such as MaxCompiler or a conventional HDL like Verilog.

Our approach is also not limited to one validation technique. While we implement and evaluate our approach using an equivalence checker, other techniques such as model checkers could also be used.

This section introduces our abstract approach to validating optimized designs. The next section shows how we apply this general, abstract approach to the specific problem of validating streaming reconfigurable designs.

## 4. IMPLEMENTATION: MAXELER TARGETS

We implement our approach for Maxeler MaxCompiler kernel designs that support streaming computation. These are effectively dataflow graphs which the user constructs programmatically using Maxeler's custom extensions to the Java programming language.

Our implementation comprises:

- a *compiler* from a subset of Maxeler's MaxJ kernel description language to symbolic simulator input. The compiler includes an interpreter for a subset of Java to evaluate (unroll) any compile-time loops and conditionals used for metaprogramming. The compiler then evaluates the design to build a dataflow graph (DFG) and compiles the DFG into symbolic simulator input.

- a *symbolic simulator* based on the Rebecca system [14], which can simulate bit-level and word-level designs, and combine symbolic, numeric and logical inputs and outputs;

- an *output combiner*, which compiles symbolic outputs from source and target designs into a single design in the Yices format, using universal quantified variables to represent symbolic values;

- the Yices [15] SMT (Satisfiability Modulo Theories) solver [15], used as an *equivalence checker*, to check whether symbolic expressions, which may be structurally different, are semantically equivalent.

All tools except Yices are written or modified by us.

Because Maxeler's MaxCompiler allows use of the full power of the Java language to metaprogram designs (particularly loops and iteration statements), we choose to interpret any metaprogramming features in the designs before compiling to the symbolic simulation. While it would be more general to try to validate the designs without interpreting the metaprogramming features, this requires a more general logic system and more user intervention [9].

For symbolic simulation, we extend the Rebecca symbolic simulator [16], rather than implement our own simulator or use another implementation. This simulator has several advantages: (a) it supports both word-level and bit-level designs, meaning that our approach could extend to bit-level designs in future; (b) there are already several tools which compile other design inputs into the same simulator, examples including the Ruby language and the Pebble hardware description language, which is effectively a structural subset of VHDL, meaning our approach could extend to include hardware cores written in other design inputs in future.

Previous work has used the ACL2 theorem prover for simulation [12]. Whilst powerful, ACL2 is a large, complex system which would be difficult to integrate with our other tools. In contrast, our simulator is simple but modular, and can be customised to support (i) new processing elements, as long as their behaviour can be described numerically and symbolically; (ii) new simulator functions, such as converting between bit-level and word-level representations; (iii) new target description formats such as VHDL, as well as formats required by other tools such as Yices.

Moreover, the simulator can itself include optimizations (such as simple code simplification) which can be validated by an independent checker; the simplified output can ease the equivalence checking in the next step, potentially allowing larger designs to be verified. Since the same internal representation generates the netlist for multiple targets, a single design description can be used for both verification and synthesis, reducing design effort and ensuring that what is synthesized is what was verified.

We use the Yices SMT solver [15] to verify the equivalence of source and target designs. We choose Yices for several reasons. First, because its concept of function theory allows us to implement array and memory updates as updates to functions; this sometimes allows us to compare the outputs of original and transformed designs without needing to reorder them, which can save time and eliminate possible bugs in the reordering.

Second, Yices allows checking of designs at *word-level*, rather than bit-level, and supports linear arithmetic on integers and reals. We make the tradeoff: rather than checking every bit of the design, we assume that the operator implementations are correct (perhaps verified by other approaches, such as Singh and Lilleroth [5]), and check the arithmetic at word level. This potentially allows much larger designs to be checked but relies on correct operator implementations. Since such operator implementations are very heavily tested, and can be verified by other means, we feel this is a reasonable compromise.

*Example:* we illustrate comparing unrolled and rolling-sum versions of a one-dimensional moving average kernel. The following is a basic MaxCompiler kernel implementation (for clarity, we only show the kernel core without surrounding declarations; line numbers are not part of the code):

```
1:HWVar x = io.input("x", scalarType);
2:HWVar sum = constant.var(scalarType, 0);
3:for (int i = 0; i < W; i++)
4:  sum += stream.offset(x, -i);
5:io.output("z", sum / W, scalarType);
```

Respectively: line 1 declares a stream input named x; the type scalarType is parameterizable, in our example it is single-precision floating point, lines 3-4 implement the summation, summing the current stream input with previous ones using stream offsets and line 5 outputs the sum divided by the window size $W$ to an output stream named z.

Note that the loop in lines 3-4 runs at compile time, effectively unrolling the summation and resulting in $W$ adders in total. We compare this design with one implemented as a rolling sum, which saves $W - 2$ additions by saving partial sum results between outputs:

```
1:HWVar inp1=io.input("x",scalarType);
2:HWVar temp=scalarType.newInstance(this);
3:HWVar sum=temp+inp1;
4:temp <== stream.offset(sum,-1);
5:HWVar temp2=sum-stream.offset(sum,-(W+1));
6:io.output("z",temp2 / W,scalarType);
```

where respectively: line 1 is as before, line 2 creates an unconnected graph node which will be used for feedback, line 3 adds the current input to the rolling sum, line 4 connects the sum result back to the unconnected node, line 5 subtracts the input from $W + 1$ cycles previously, line 6 is as before.

Our compiler translates this into the following simulator input, for window width $W = 3$:

```
1: D 0 .2 .3
2: D 0 .1 .2
3: add <.3,.2>   .4
4: add <.4,.1>   .5
5: div <.5,3>    .7
6:Directions - in ~ out
7:Wiring - .1 ~ .7
```

where nodes are numbered, so `.1` is node 1, and each line comprises the name of a simulator primitive and its input and output nodes, for example `D 0 .2 .3` is a register with input node 2, output node 3. Respectively: lines 1-2 implement a register chain to implement the stream offsets, lines 3-4 implement the unrolled adder tree, line 5 divides by constant 3, and lines 6-7 give the external connections, input `x` and output `z` compiled into nodes 1 and 7 respectively.

To simulate symbolically, we generate symbols to apply to the inputs. We let the user decide how many cycles to simulate; for Maxeler designs controlled by counters, the design should be simulated for the total number of counter states. Simulation results are as follows:

```
0 - a_0 ~ ((0 + a_0) / 3)
1 - a_1 ~ (((0 + a_0) + a_1) / 3)
2 - a_2 ~ (((a_0 + a_1) + a_2) / 3)
```

This shows three cycles of symbolic output for symbolic input stream `a_0`, `a_1`, `a_2`.

Finally, our output combining tool compiles the source and target symbolic simulation outputs into input for the equivalence checker. For our simple example, this looks like (window width $W = 3$):

```
1: (echo "1: ")
2: (assert (forall (a_0::int)
   (=(/(+ 0 a_0)3) (/(-(+ 0 a_0)0)3))))
3: (check)
```

where each simulation clock cycle gives rise to three statements: an echo statement, which prints the clock cycle, an assert statement, which declares symbolic variables used in the statement and compares the output expressions, and a check statement, which checks the previous statement for equivalence. Note that the language uses Lisp-style s-expressions, so each expression is enclosed by brackets and contains the operator and its operands in sequence, thus $a_0 + a_1$ becomes $(+a_0 a_1)$.

Our compile scheme to translate symbolic expressions into Yices input works by translating: 1. symbolic values to universally-quantified variables; 2. uninitialized values into uniquely-named new variables, which may help the user to locate the source of any mismatch; 3. corresponding symbolic outputs into a Yices statement asserting their equality.

The assert statement uses a universal quantifier *forall* to compare output expressions from the source and target designs, which are the first and second operands of the '=' operator, respectively. Without the universal quantifier, the SMT solver will try to guess values for symbols, which can lead to different expressions being wrongly found to be equivalent. Sample output is:

```
1: sat
2: sat
```

For each cycle, the equivalence checker output can be: (a) *sat*, meaning the source and target designs match, (b) *unknown*, meaning the checker could not tell if the designs matched, and (c) *unsat*, meaning the outputs definitely do not match. If the output is *sat* for all cycles, the source and target designs have been successfully verified as equivalent. The *unknown* case may indicate that the problem is too large for the equivalence checker, meaning the user should try to prove corresponding parts of the design equivalent instead. In the *unsat* case, the user can examine the symbolic expressions to aid in debugging the design.

Because different designs may have different startup latencies (cycles to remove uninitialized data from the pipeline), or run at different rates, the output combiner can be parameterized to skip (a) a set number of offset (startup) cycles, and (b) every $n$ cycles, where $n >= 1$.

## 5. RESULTS AND EVALUATION

We evaluate our approach by applying it to several benchmarks, including one dimensional convolution and finite difference computation. Transformations applied include:

- loop unrolling: the simplest way to deal with loop-carried dependences in streaming designs, leading to a large design area;
- rolling sums: reduce area and latency by saving partial sum results;
- multi-cycle feedback: a simple way to deal with loop-carried dependences

*Experimental setup*: we use Maxeler MaxCompiler 2011.3.1 and Xilinx ISE 13.3 to implement our designs as Maxeler kernels; verification uses our symbolic simulator compiler and output combiner, Rebecca 0.1 and Yices 1.0.34. Maxeler kernels require that all loop-carried dependences be resolved by the user, so we choose the simplest design which resolves the dependences as a reference: multi-cycle feedback.

*1D Convolution*: table 1 summarizes the designs verified. We see that the rolling sum design has the highest clock speed but a low throughput compared to the unrolled version, while the multicycle has lowest area but lowest throughput.

*Reverse time migration (RTM)*: we implement a simplified core of the RTM algorithm, which iterates over a discrete grid, summing corresponding neighbours of a grid element along each axis. We verify that for a 1D implementation, both unrolled (fig. 2) and rolling sum versions are symbolically equivalent. Because the rolling sum needs to be primed, we offset that version by the number of cycles to prime. The result shows that the unrolled and rolling sum versions are verified as equivalent.

## 6. CONCLUSION

This paper describes a novel approach to verifying the equivalence of streaming designs on reconfigurable hardware by combining symbolic simulation and equivalence checking.

Future work includes extending our system to include bit-level designs, which would eliminate any errors in translating from word level to bit level, or allow the user to verify part of their design at word level, part at bit level, concentrating the greater computational demands of bit-level verification on problematic parts of their design. Also, our approach only compares two designs, but if designs are optimized by transformation, there are potentially many designs that can be

| Design | Verification time (s) | Code size (lines) | Area (% LUTs) | Area (% FFs) | Speed (MHz) | Latency (cycles) | Throughput (results/cycle) |
|---|---|---|---|---|---|---|---|
| Unroll | $< 1$ | 32 | 4.07 | 2.53 | 69.5 | $WL$ | 1 |
| Rolling Sum | $< 1$ | 41 | 2.14 | 1.26 | 77.0 | $L$ | $1/L$ |
| Multicycle | $< 1$ | 48 | 2.00 | 1.19 | 75.3 | $WL$ | $1/(WL)$ |
| Loop tile | $< 1$ | 71 | 2.27 | 1.33 | 55.9 | $L$ | 1 |

**Table 1**. Results for verifying 1D convolution targeting Xilinx Virtex-6 SX475T FPGA: $W$ is the window width, $L$ is the latency of one adder, speed and area results for $W = 16$, single-precision floating point data.

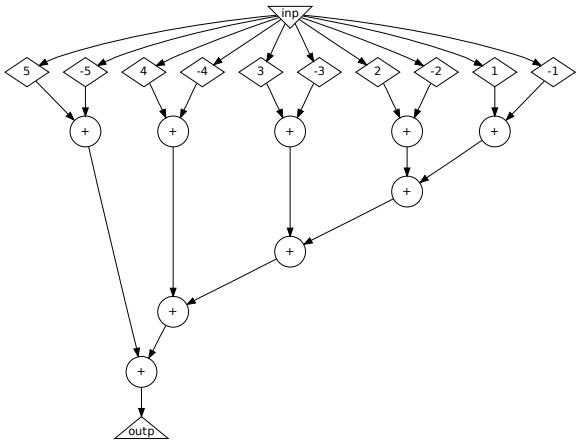


**Fig. 2**. Dataflow graph from reverse time migration.

compared. Verifying multiple, related, designs at once could lead to faster verification than verifying each in turn versus the reference design, as some components of the transformed designs will be identical. Another direction of research is to integrate the proposed approach with complementary previous work, such as those based on designs with explicit layout directives [9].

*Acknowledgement:* The research leading to these results has received funding from European Union Seventh Framework Programme under grant agreement number 287804, 248976 and 257906. The support by UK EPSRC, the HiPEAC NoE, the Maxeler University Program, and Xilinx is gratefully acknowledged.

## 7. REFERENCES

[1] J. Villarreal, A. Park, W. Najjar, and R. Halstead, "Designing modular hardware accelerators in C with ROCCC 2.0," in *FCCM*, May 2010, pp. 127 –134.

[2] "MaxCompiler White Paper," Maxeler Technologies, Tech. Rep., February 2011. [Online]. Available: http://www.maxeler.com/content/briefings/MaxelerWhitePaperMaxCompiler.pdf

[3] T. Coe, "Inside the Pentium FDIV bug," *Dr. Dobb's Journal*, no. 229, pp. 129–135 and 148, April 1995.

[4] "Formality: equivalence checking for DC Ultra," Tech. Rep., 2012. [Online]. Available: http://www.synopsys.com/tools/verification/formalequivalence/pages/formality.aspx

[5] S. Singh and C. J. Lillieroth, "Formal verification of reconfigurable cores," in *FCCM*. IEEE Computer Society, 1999, pp. 25–32.

[6] K. W. Susanto and T. F. Melham, "Formally analyzed dynamic synthesis of hardware," *The Journal of Supercomputing*, vol. 19, no. 1, pp. 7–22, 2001.

[7] F. Madlener, J. Weingart, and S. A. Huss, "Verification of dynamically reconfigurable embedded systems by model transformation rules," in *CODES+ISSS*. ACM, 2010, pp. 33–40.

[8] K. W. Susanto and W. Luk, "Automating formal verification of customized soft-processors," in *FPT*. IEEE, 2011, pp. 1–8.

[9] O. Pell, "Verification of FPGA layout generators in higher-order logic," *J. Autom. Reasoning*, vol. 37, no. 1-2, pp. 117–152, 2006.

[10] S. Drzevitzky, U. Kastens, and M. Platzner, "Proof-carrying hardware: Towards runtime verification of reconfigurable modules," in *ReConFig*. IEEE Computer Society, 2009, pp. 189–194.

[11] S. Drzevitzky, "Proof-carrying hardware: Runtime formal verification for secure dynamic reconfiguration," in *FPL*. IEEE, 2010, pp. 255–258.

[12] K. W. Susanto, T. Todman, J. G. F. Coutinho, and W. Luk, "Design validation by symbolic simulation and equivalence checking: A case study in memory optimization for image manipulation," in *SOFSEM*, ser. Lecture Notes in Computer Science, vol. 5404. Springer, 2009, pp. 509–520.

[13] T. Todman, Q. Liu, W. Luk, and G. A. Constantinides, "Customizable composition and parameterization of hardware design transformations," in *DSD*. IEEE, 2010, pp. 595–602.

[14] S. R. Guo and W. Luk, "An integrated system for developing regular array designs," *Journal of Systems Architecture*, vol. 47, no. 3-4, pp. 315–337, April 2001.

[15] B. Dutertre and L. de Moura, "The YICES SMT Solver," Computer Science Laboratory, SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025 - USA, Tech. Rep., 2006.

[16] W. Luk, "A declarative approach to incremental custom computing," in *FCCM*, April 1995, pp. 164 –172.