Increasing Verilog's Generative Power

Cherif Salama
Ain Shams University, Cairo, Egypt
Email: Cherif.Salama@eng.asu.edu.eg

Walid Taha
Halmstad University, Halmstad, Sweden
and Rice University, TX, USA
Email: Walid.Taha@hh.se

***Abstract*—To cope with more complex circuits, well-understood higher-level abstraction mechanisms are needed. Verilog is already equipped with promising generative constructs making it possible to concisely describe a family of circuits as a parameterized module; however these constructs suffer from limited expressivity even in the latest IEEE standard. In this paper, we address generative constructs expressivity limitations, identifying the key extensions needed to overcome these limitations, and showing how to incorporate them in Verilog in a disciplined, backward-compatible way.**

I. Introduction

Building circuits using a library of pre-designed fixed modules is the most basic form of code reuse in Verilog. To maximize reusability, library modules should be customizable to fit in a wide variety of circuits. For example, a 32-bit adder is not very useful when designing a circuit that needs a 16-bit adder. On the other hand, a customizable-width adder is useful not only in this situation but whenever an adder is needed. Such a customizable adder is called a *parameterized* adder and represents a *family* of circuits (in this case a family of adders) as opposed to a single circuit. In general a parameterized module represents a circuit family whose individual members correspond to instantiations with specific parameter values.

The 2001 IEEE Verilog standard [1] introduces generative constructs that offer a concise way of describing a family of circuits as a parameterized module. These constructs are known as *generative constructs* because a circuit family can be viewed as a template for generating various concrete instances. These constructs are built on top of purely structural constructs providing a higher level of abstraction than gate level descriptions while retaining the same level of control over the structure of the circuit. Generative constructs are designed to be expanded away during a phase known as *elaboration*. If successful, the result of elaboration is a purely structural (gate level) description. Unfortunately, these constructs suffer from two severe problems making their usage impractical in most settings. First, generative constructs are not adequately supported by current tools. For instance, there are no static guarantees about the properties of the description generated as a result of instantiating a generic description with particular parameter values. The second problem is the limited expressivity of generative constructs. Even in the latest IEEE standard [2], these constructs are limited to loops and conditionals, while module parameters are still restricted to integer values.

The first problem was previously addressed through formalizing Featherweight Verilog (FV), a core subset of structural Verilog with generative constructs, as a statically typed two-level language [3]. This formalization included elaboration semantics and a static type system providing basic synthesizability guarantees.The type system was later expended using indexed types to provide for more static guarantees [4], [5].

This paper addresses the second problem showing how to increase Verilog's generative power in a disciplined, backward-compatible way through the following contributions.

A. Contributions

To increase Verilog's generative power we:

- List the main missing features that limit Verilog's expressivity (Section II).
- Describe the proposed extended Verilog syntax by means of illustrative examples (Section III).
- Informally explain the elaboration semantics of the proposed constructs (Section IV).
- Informally describe how to extend the static analyses to support the new constructs (Section V).
- Provide an implementation supporting the new constructs (Section VI).

II. Missing Language Features

Verilog's support for generative constructs provides a natural way to capture the design of circuits with linear structures. While many important circuits like adders, multipliers, and counters fit this pattern, there are other equally important circuits that have so far required ad hoc program generation techniques to describe. Examples of such circuit patterns include tree shaped circuits [6] and butterfly circuits [7]. Additionally, even when they only feature linear structures, generic composition patterns like the common ripple pattern shown in Figure 1, cannot be expressed in Verilog. These expressivity restrictions do not exist in functional hardware description languages like Lava [8]. For example, the ripple pattern can be described in Lava as follows:

```
ripple circ (cin, []) = ([], cin)

ripple circ (cin, a:as) = (b:bs, cout)
  where
    (b, carry) = circ (cin, a)
    (bs, cout) = ripple circ (carry, as)
```

Describing such patterns requires the use of 1) recursion, 2) polymorphism, and 3) the ability to parametrize a circuit

This work was supported by the National Science Foundation (NSF) CPS award 1136099 and the Semiconductor Research Consortium (SRC) Task ID: 1403.001 (Intel custom project).

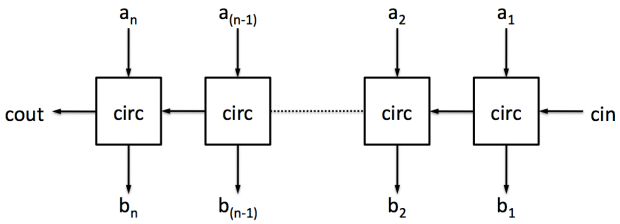


Fig. 1. The Ripple Pattern Connecting `n` Instances of `circ` (adapted from [8])

by other circuits. These points were made in early works by Sheeran et al on Lava [8], and by O'Leary et al on HML [9]. These ideas have also been incorporated into systems such as BlueSpec [10].

Our goal is to re-introduce these three features into Verilog as extensions that can be implemented through a preprocessor, that is, as syntactic sugar. In addition, and most importantly, we demonstrate that powerful static analyses are still possible with these extensions.

Achieving these goals requires: 1) extending the syntax in a manner that is compatible with standard Verilog, 2) defining the semantics for the new extensions by a translation into Verilog, and 3) defining the static analyses at the level of this extended source language.

III. Proposed Syntax

For syntax, we propose a syntax which would allow expressing the above Lava example as follows:

```
module ripple
  (output 't1 [N-1:0] b, output 't2 cout,
   input  't3 [N-1:0] a, input  't2 cin);

   parameter        N=8 where N >= 1;
   parameter circ (output 't1 y, output 't2 co,
                   input  't3 x,  input 't2 cin);

   't2     carry;

   circ c (b[0],carry,a[0],cin);

   if (N == 1)
     assign cout = carry;
   else begin
      ripple ##(circ) #(N-1) ra
        (b[N-1:1],cout,a[N-1:1],carry);
   end
endmodule
```

The above generic module recursively describes the ripple pattern. It is parametrized by an integer parameter `N` and by the module `circ` whose full signature is specified. It also allows the ports `a` and `b` to be arrays of arbitrary types. Being parameterized by a module, the `ripple` pattern is a higher order module. When instantiating a higher order module, module parameters must be passed in before passing in the integer parameter values (if any). To easily differentiate between module and integer parameters, the former is preceded by `##` as opposed to `#`.

Although fairly concise, the Verilog description is still significantly longer than its Lava counterpart primarily due to its explicit type declarations (as opposed to Lava's inferred types). This price cannot be avoided since we aim to provide extensions that fit naturally with the rest of Verilog, and that existing Verilog users can easily adopt.

In addition, the ripple pattern example shown above features a `where` clause in the parameter declaration statement. This is also a Verilog extension that we propose. The purpose of a `where` clause is to restrict the possible values of a parameter that can be passed in when instantiating the module. This is a useful feature that makes it possible to explicitly state that a module was not designed to work for all integer values.

The last extension we propose is to fully support multidimensional arrays. The latest Verilog standard supports multidimensional arrays but not for module ports. A port can either be a single bit or a simple array (a bitvector). We find this restriction quite limiting and we propose removing it. This will allow us for example to define a generic bus multiplexer as follows:

```
module gen_bus_mux(mux_out,mux_in,sel);
   parameter M=2 where M >= 1;
   parameter W=8;

   input  [W-1:0]    mux_in [2**M-1:0];
   input  [M-1:0]    sel;
   output [W-1:0]    mux_out;
   genvar       i,j;

   wire [2**M-1:0]  decsel;
   wire [2**M-1:0]  p [W-1:0];

   decoder #(M) dec1 (decsel,sel);

   for(i=0;i<2**M;i=i+1)
     for(j=0;j<W;j=j+1)
       and (p[j][i],decsel[i],mux_in[i][j]);

   for(i=0;i<W;i=i+1)
     assign mux_out[i] = |p[i]; //reduction operator
endmodule
```

This module uses two parameters `M` and `W` to specify the bus width and the number of selection lines respectively. The number of selection lines is restricted to be at least one. This module has 3 ports, two of which are simple arrays (`mux_out` and `sel`). The third port (`mux_in`), however, needs to be an array of buses which means it must be an array of array. We use Verilog's standard notation in specifying multidimensional arrays to specify that `mux_in` is a multidimensional array (an array of bitvectors to use Verilog's terminology).

To summarize, we propose the five following extensions:

- Recursive modules
- Higher order modules
- Parametric polymorphism
- Parameter constraints
- Multidimensional ports

In the next sections we consider the implications of these extensions on the elaboration semantics and the static analyses previously defined [3], [4].

IV. Semantics of Proposed Extensions

To ensure backward compatibility, any extensions we make to Verilog Syntax need to be fully expanded away during elaboration. We assume that we only elaborate well-typed descriptions since type checking is performed before elaboration [3]. Here is what needs to be done for each addition:

- **Recursive modules**: Perhaps surprisingly, nothing needs to be changed in the standard elaboration semantics to support recursive modules. A well-typed recursive module never instantiates itself with the same parameter values it received. This is guaranteed by the termination analysis we present in the next section. This means that standard elaboration will successfully unroll recursion completely. To see why, consider an N-bit ripple adder defined recursively and then instantiated to create a 16-bit ripple adder. Elaborating this instantiation causes the creation of a 16-bit ripple adder module composed of a full adder and an instantiation of a 15-bit ripple adder. This instantiation in turn creates a 15-bit ripple adder module using a 14-bit ripple adder, and so on until the base case (a 1-bit ripple adder) is reached. After elaboration none of the resulting modules is recursive which means that recursive modules are expanded away on their own without any modifications in the standard elaboration process.
- **Higher order modules**: Elaborating higher order modules is very similar to elaborating parameterized modules. When instantiated, a higher order module must be given the name of the module(s) to use internally. This in turn creates a specialized (first order) module where the names of the modules being instantiated are substituted by the given module names.
- **Parametric polymorphism**: A polymorphic module can only be instantiated with concrete types. The polymorphic types are replaced by their corresponding concrete types creating a specialized (non polymorphic) module. The concrete types to use are determined during the type checking phase as explained in the next section.
- **Parameter constraints**: Elaboration soundness implies that during elaboration all parameter declarations and their associated `where` clauses are dropped.
- **Multidimensional ports**: Multidimensional ports are replaced by a series of single dimensional ports during elaboration. This can only be done when a module is instantiated since the number of generated ports might depend on a parameter value as in the generic bus multiplexer example. This change is reflected in the module ports declaration and when the module is instantiated.

V. Static Analyses

Adding new constructs to a language is almost guaranteed to increase the complexity of static analyses defined for that language. This is particularly true of generative constructs. Here is what needs to be done for each extension:

- **Recursive modules**: Recursive modules are the most challenging extension to handle. There are 3 modifications to type checking that are required: 1) Type checking needs to be done in two passes. The first one to infer all the module types and the second one to actually type the module definitions. 2) Static resource estimation needs a recurrence relation solver because the amount of resources required by a module may potentially be a recurrence. We suggest using PURRS [11] or PUBS [12] to obtain a closed form estimate or content ourselves with recurrences as gate count estimates. 3) Type checking now requires a more elaborate termination analysis due to the possibility of preprocessing divergence. We present the analysis we use in Section V-A.
- **Higher order modules**: To support higher order modules, the type checker needs to check that in the definition of the higher order module each instantiation using a module parameter has a type compatible with its declaration. Additionally, when a higher order module is instantiated, the type checker needs to verify that the module passed as a parameter is compatible with the expected type.
- **Parametric polymorphism**: To verify that a polymorphic module is instantiated correctly, the type checker needs to infer the concrete types corresponding to the polymorphic types. We use unification [13] as suggested by Hindley-Milner type inference [14], [15].
- **Parameter constraints**: Supporting `where` clauses is a straightforward extension. First, when type checking the definition of the parameterized module with a restricted parameter, we consider the parameter constraint to be a given that we can use to prove other consistency constraints. Second, the type checker verifies that the default parameter values satisfy the constraints imposed on them. Similarly, at the instantiation site, the type checker verifies that the passed parameter values also satisfy the same constraints. Parameter constraints are also used for termination analysis as described in the next section.
- **Multidimensional ports**: Nothing special needs to be done to support multidimensional ports during type checking.

A. Termination Analysis

Termination analysis is an active research area [16]–[18]. In our proposed version of Verilog, there are only two potential reasons for elaboration divergence: loops and recursive calls. Since we restrict loops to a very simple form, our type checker is already able to verify their termination by using an SMT solver to prove that the loop index increases with each iteration. Since we allow mutual recursion, proving termination of recursive calls requires more work despite the restricted nature of Verilog's modules which only allow integer parameters. We propose an abstract interpretation [19], [20] of the description using a directed graph.

We construct a directed graph where each vertex represents a module and each edge an instantiation. We draw an edge

from module A to module B iff module A instantiates module B. If module A instantiates module B twice or more, then two or more edges going from A to B are drawn. Each module is labelled with the names of its parameters while edges are labeled with: 1) The expressions passed as parameter values during the instantiation and 2) The instantiation guard (if any). The instantiation guard is the conjunction of all the constraints known to be true at the point of the instantiation. This includes parameter constraints and conditions implied from the surrounding generative conditionals.

Figure 2 shows the module instantiation graph for the `ripple` pattern described above. The description includes a single module `ripple` that contains only two instantiations. First, it has an instantiation of module `circ`, and second, it has a recursive instantiation of module `ripple`. This is why the graph shows exactly two edges both originating at the `ripple` module vertex. One of the edges connects it to the `circ` vertex and the other is a loop back. The `ripple` vertex is labeled with N and the other vertex is not labelled since `circ` is not a parameterized module. The edge going to `circ` is labelled with $N >= 1$ which is the only constraint known to be true at its instantiation (due to the parameter constraint on N). Again, since `circ` is not a parameterized module, there are no parameter values on that edge. The other edge however is labeled with $N - 1$ as the parameter value passed at instantiation time and with $N >= 1 \wedge N \neq 1$ as the instantiation guard. The left hand side of the conjunction comes from the parameter constraint and the right hand side is known because the instantiation is in the alternative block of the conditional expression checking for $N = 1$.

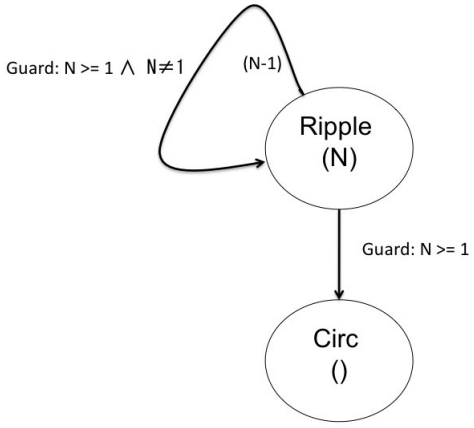
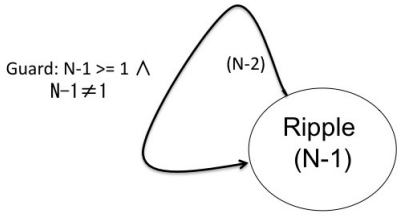
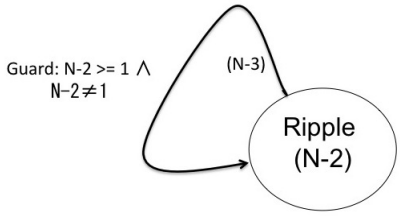


Fig. 2. Instantiation Graph for the Ripple Pattern

The first step of the algorithm after constructing the graph is to detect all cycles. If no cycles are detected, then there are no recursive definitions at all and termination is trivially guaranteed. For each cycle detected, however, termination needs to be proved. In Figure 2, we only have one cycle composed of one edge and to prove termination we need to show that its instantiation guard cannot hold forever, or in other words, we need to show that each cycle causes the parameter values used at instantiation to change in a way that will eventually cause the guard to fail. In this example, the guard is the conjunction $N >= 1 \wedge N \neq 1$. Showing that either $N >= 1$ or $N \neq 1$ cannot hold forever is enough. To do so we need to track how the parameter values change over time by propagating them through the cycle. Since vertex `ripple` has an incoming edge labelled with $N - 1$, we record that the initial parameter value passed to `ripple` when instantiated recursively is $N - 1$. This value replaces the parameter N everywhere which means that the edge should now be labelled with $N - 1 - 1$ (or $N - 2$) as shown in Figure 3(a). Figure 3(b) shows what happens when the $N - 2$ is propagated through the edge, replacing the original N with $N - 2$. This means that after one cycle, the parameter value has changed from $N - 1$ to $N - 2$. Proving that $N - 2$ is less than $N - 1$ is enough to prove that N is decreasing. This in turn is the only thing needed to prove that the guard conjunction will eventually fail and that the recursive instantiations will eventually terminate.



(a) Initial Value



(b) Value After Once Cycle

Fig. 3. Propagating Parameter Values Through The Loop

In general, to prove termination of a cycle, we first check that there is at least one guard on one of the edges in the cycle. If none is found then we know that this cycle does not terminate. If we can find at least one guard and prove that it will eventually fail to hold then we have proved termination. Again, in order to do so, we need to propagate the parameter values through the cycle and see how this affects the guard. This approach is general enough to handle mutually recursive module instantiations. We illustrate this by looking at the sample cycle in Figure 4.

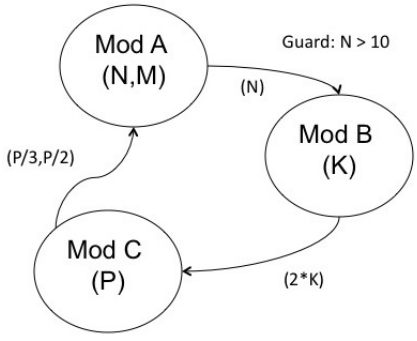


Fig. 4. A Cycle in the Instantiation Graph Indicating Mutually Recursive Modules

In this cycle a single guard ($N > 10$) exists on module `B`'s instantiation by module `A`. To prove termination we need to

prove that N decreases in each cycle. The first time module `A` is instantiated, N is set to $P/3$ and M is set to $P/2$. Given these values, we know that if module `B` is instantiated, it will be instantiated with $P/3$ and subsequently module `C` will be instantiated with $2 * P/3$ and then module `A` again with N equals to $2 * P/9$. Since $2 * P/9$ is less than $P/3$, then we have proved that the value of N decreases which means that eventually it will fail to satisfy the guard guaranteeing the termination of this recursive cycle.

VI. Implementation

Luckily, all generative constructs are meant to be removed during the elaboration phase resulting in purely structural Verilog code. As a proof of concept, we implemented an unobtrusive Verilog Preprocessor, VPP, that expends away all the extensions suggested in this paper. VPP's type checker carries out all the analyses described. It relies on Yices [21], a state of the art SMT solver, to ensure instantiation cycles termination by proving that the expression obtained after propagating the parameters around the cycle is less than (or greater than depending on the guard) the initial expression. VPP, however, does not use any recurrence solvers at the moment. Instead it reports the gate count as a recurrence relation.

A. Experimental Results

Increasing Verilog's generative power by introducing the above constructs would not be viable if the generated circuits were less efficient than their manually written counterparts. To experimentally verify that this is not the case, we used the ripple pattern above to generate a 16-bit ripple adder. After elaboration we ended up with a purely structural Verilog description of a 16-bit ripple adder built from only 2 components: a full adder and a 15-bit ripple adder. The 15-bit ripple adder was itself defined in terms of a full adder and a 14-bit ripple adder and so on. The generated description seemed much longer and much more complex than the manual implementation of a 16-bit adder composed of 16 full adders. Although described very differently, both 16-bit adders have exactly the same structure and are expected to need exactly the same amount of resources and to have identical delays. To verify this experimentally, we synthesized both adders using Xilinx ISE and we found that both required 32 four-input look up tables (LUTs) utilizing 18 slices of the target FPGA. Both circuits had 18 levels of logic with a delay of 33.378 nanoseconds. We also obtained almost identical netlist files when we tried synthesizing both adders using the Icarus Verilog compiler.

VII. Summary

We extended Verilog's generative constructs to include the most important features from functional hardware description languages, and we believe that the newly available language is fairly expressive. We were successfully able to generalize our static analyses to support the extended language demonstrating that, by having the right kind of abstractions, a language can be both expressive and predictable. As opposed to RTL and algorithmic level abstractions, the proposed abstractions can be statically checked and do not limit the designer's full control over the circuits he is creating.

References

[1] IEEE Standards Board, *IEEE Standard Verilog Hardware Description Language*, ser. IEEE Standards. IEEE, 2001, no. 1364-2001.

[2] ——, *IEEE Standard for Verilog Hardware Description Language*, ser. IEEE Standards. IEEE, 2005, no. 1364-2005.

[3] J. Gillenwater, G. Malecha, C. Salama, A. Y. Zhu, W. Taha, J. Grundy, and J. O'Leary, "Synthesizable high level hardware descriptions: using statically typed two-level languages to guarantee Verilog synthesizability," in *PEPM '08: Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*. New York, NY, USA: ACM, 2008, pp. 41–50.

[4] C. Salama, G. Malecha, W. Taha, J. Grundy, and J. O'Leary, "Static consistency checking for Verilog wire interconnects: Using dependent types to check the sanity of Verilog descriptions," in *PEPM '09: Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*. New York, NY, USA: ACM, 2009, pp. 121–130.

[5] C. Salama, W. Taha, J. Grundy, and J. O'Leary, "The VPP verilog preprocessor," in *HFL '09: Hardware design and Functional Languages*, 2009.

[6] A. Bunker, "A hardware combinator for tree-shaped circuits," Master's thesis, Brigham Young University, 1998.

[7] O. Mukhanov and A. Kirichenko, "Implementation of a fft radix 2 butterfly using serial rsfq multiplier-adders," *Applied Superconductivity, IEEE Transactions on*, vol. 5, no. 2, pp. 2461–2464, Jun 1995.

[8] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: Hardware design in Haskell," *ACM SIGPLAN Notices*, vol. 34, no. 1, pp. 174–184, Jan. 1999.

[9] J. W. O'Leary, M. H. Linderman, M. Leeser, and M. Aagaard, "HML: A hardware description language based on standard ML." in *CHDL*, 1993, pp. 327–334.

[10] *Bluespec SystemVerilog Version 3.8 Reference Guide*, Bluespec, Inc, 2006.

[11] R. Bagnara, A. Pescetti, A. Zaccagnini, E. Zaffanella, and T. Zolo, "PURRS: the parma university's recurrence relation solver," http://www.cs.unipr.it/purrs/, the University of Parma.

[12] E. Albert, P. Arenas, S. Genaim, and G. Puebla, "Automatic inference of upper bounds for recurrence relations in cost analysis," in *SAS '08: Proceedings of the 15th international symposium on Static Analysis*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 221–237.

[13] J. A. Robinson, "Computational logic: The unification computation," *Machine Intelligence*, vol. 6, pp. 63–72, 1971.

[14] J. R. Hindley, "The principle type-scheme of an object in combinatory logic," *Transactions of the American Mathematical Society*, vol. 146, pp. 29–60, 1969.

[15] R. Milner, "A theory of type polymorphism in programming," *Journal of Computer and System Sciences*, vol. 17, pp. 348–375, August 1978.

[16] A. Abel and T. Altenkirch, "A semantical analysis of structural recursion," in *Journal of Functional Programming*, 1999.

[17] C. S. Lee, N. D. Jones, and A. M. Ben-Amram, "The size-change principle for program termination," in *POPL*, vol. 28. ACM press, january 2001, pp. 81–92.

[18] D. Sereni and N. D. Jones, "Termination analysis of higher-order functional programs," in *Proceedings of the Third Asian Symposium on Programming Languages and Systems (APLAS 2005)*, ser. Lecture Notes in Computer Science, vol. 3780. Springer-Verlag, 2005, pp. 281–297.

[19] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. New York, NY, USA: ACM, 1977, pp. 238–252.

[20] ——, "Systematic design of program analysis frameworks," in *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. New York, NY, USA: ACM, 1979, pp. 269–282.

[21] B. Dutertre and L. de Moura, "The Yices SMT solver," Tool paper at http://yices.csl.sri.com/tool-paper.pdf, SRI International, August 2006.