Static Consistency Checking for Verilog Wire Interconnects *

Using Dependent Types to Check the Sanity of Verilog Descriptions

Cherif Salama Gregory Malecha
Walid Taha
Rice University
{cherif,gmalecha,taha}@rice.edu

Jim Grundy John O'Leary
Intel Strategic CAD Labs
{jim.d.grundy,john.w.oleary}@intel.com

Abstract

The Verilog hardware description language has padding semantics that allow designers to write descriptions where wires of different bit widths can be interconnected. However, many of these connections are nothing more than bugs inadvertently introduced by the designer and often result in circuits that behave incorrectly or use more resources than required. A similar problem occurs when wires are incorrectly indexed by values (or ranges) that exceed their bounds. These two problems are exacerbated by generate blocks. While desirable for reusability and conciseness, the use of `generate` blocks to describe circuit families only makes the situation worse as it hides such inconsistencies making them harder to detect. Inconsistencies in the generated code are only exposed after elaboration when the code is fully-expanded.

In this paper we show that these inconsistencies can be pinned down prior to elaboration using static analysis. We combine dependent types and constraint generation to reduce the problem of detecting the aforementioned inconsistencies to a satisfiability problem. Once reduced, the problem can easily be solved with a standard satisfiability modulo theories (SMT) solver. In addition, this technique allows us to detect unreachable code when it resides in a block guarded by an unsatisfiable set of constraints. To illustrate these ideas, we develop a type system for Featherweight Verilog (FV), a core calculus of structural Verilog with generative constructs and previously defined elaboration semantics. We prove that a well-typed FV description will always elaborate into an inconsistency-free description. We also provide a freely-available implementation demonstrating our approach.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Code generation; Preprocessors; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs

General Terms Languages, Standardization, Theory, Verification

* This work was supported by the National Science Foundation (NSF) SoD award 0439017, and the Semiconductor Research Consortium (SRC) Task ID: 1403.001 (Intel custom project).

PEPM'09, January 19–20, 2009, Savannah, Georgia, USA.

Keywords Verilog Elaboration, Static Array Bounds Checking, Verilog Wire Width Consistency, Dead Code Elimination, Dependent Types

1. Introduction

Digital circuit design is becoming increasingly complex. Processors nowadays have hundreds of millions of transistors. Hardware description languages (HDLs) like Verilog and VHDL would have been completely useless had they required the designers to describe the circuits they want to build at the transistor level. Thankfully, this is not the case. HDLs provide designers with various kinds of abstractions to be able to describe hardware at higher levels. For example, instead of using transistors, a designer can use gates, flip-flops, latches, or complete hardware modules as circuit building blocks offering a hierarchal approach to hardware design. Productivity can be dramatically increased using higher levels of abstractions, which are therefore very desirable. Verilog (6) currently provides two types of constructs toward this end:

- **Behavioral Constructs**: These constructs allow a designer to describe a circuit functionality in a algorithmic way as one would do in C. The resultant description is suitable for simulation purposes but unless adhering to a rather ad hoc set of guidelines it cannot be synthesized into a hardware circuit. Even when synthesizable, the hardware designer has no control on the generated hardware.
- **Generative Constructs**: These constructs allow a designer to generically and structurally describe circuit families. These were introduced in the 2001 IEEE standard (5). They are particularly useful when describing frequently used modules that can be thought of as a library. When used correctly, generative constructs are fully synthesizable as they are replaced with purely structural code during a phase known as elaboration.

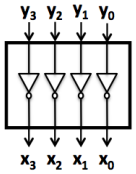
In this work we are concerned only with generative constructs since we are interested about circuits whose structure can be reasoned about. We believe that generative constructs should be used more aggressively. Hardware designers are reluctant to adopt them because current tools do not provide enough support to make using them safe. Despite the fact that generative constructs only makes sense in the context of synthesizable circuits, current tools do not attempt to check for synthesizability statically (i.e. before elaboration), let alone statically checking for inconsistencies that Verilog tolerates due to its padding semantics. Most tools will accept a description, elaborate it, and then synthesize it into a graph of connected components known as a netlist. If anything goes wrong in the process, an elaboration or synthesis error is thrown depending on when the problem occurred. Even worse: due to Verilog's padding semantics, some violations do not cause errors at all but

instead synthesize into a circuit that behaves incorrectly or uses more resources than required.

To illustrate Verilog's padding semantics and the kind of errors they can hide, let's first consider a module that does not use any generative constructs at all:

```
module invert4(x, y);
   input [3 : 0]  y;
   output [3 : 0]  x;

   assign x = ~ y;
endmodule
```



This module named `invert4` is a 4-bit inverter. It has two 4-pin ports `x` and `y` where `x` is an output port and `y` is an input port. Pins of both ports are indexed from 0 to 3. The assignment statement uses a bitwise negation operator to invert all input bits and then connect each of them to the corresponding output pin. No padding semantics are used so far but what if the declaration of `y` was changed to `input [4:0] y`? What if instead the declaration of `x` was changed to `output [4:0] x`? In both cases, the assignment statement would be connecting two wires of incompatible widths. Surprisingly, this would still be considered to be a valid description due to Verilog's padding (and trimming) semantics. In the first case the wider input signal will be trimmed discarding the most significant bit, while in the second case the slimmer input signal will be padded with an extra zero as its most significant bit. Note that in the first case, if the circuit is naively synthesized, an extra inverter will be used to invert the most significant input bit and then its output will be discarded.

Let's consider now a more interesting example: A 4-bit synchronous counter. The counter receives two inputs: a clock `clk` and an enable signal `en`. If `en` is high (equal to 1), then the 4-bit output `count` is incremented at each clock. The `next` output can be used to create a wider counter by connecting it to the `en` port of another counter thus cascading both. The schematic diagram in figure 1 shows how to construct such a counter out of T flip-flops and `and` gates.

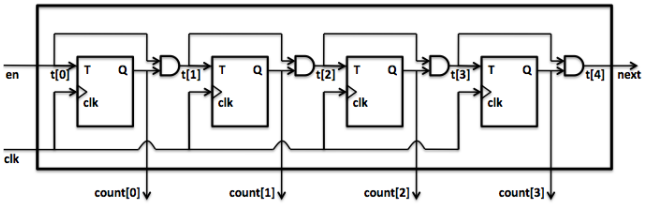


Figure 1. A 4-bit synchronous counter using T flip-flops

This module can be described using structural Verilog as follows:

```
module  counter4(count, next, en, clk);
   output [3 : 0] count;
   output next;
   input en;
   input clk;
   wire [4 : 0] t;

   assign t [0] = en;
   tflipflop  tff_0 (count [0], t [0], clk);
   assign t [1] = t [0] & count [0];
   tflipflop  tff_1 (count [1], t [1], clk);
   assign t [2] = t [1] & count [1];
   tflipflop  tff_2 (count [2], t [2], clk);
   assign t [3] = t [2] & count [2];
   tflipflop  tff_3 (count [3], t [3], clk);
   assign t [4] = t [3] & count [3];
   assign next = t[4];
endmodule
```

A regularity can easily be detected either by looking at the code or at the schematic diagram. This regularity can be captured to generalize the definition to represent the whole family of N-bit synchronous counters instead of just the 4-bit synchronous counter in figure 1. The result is the following parameterized module using a generative loop:

```
module counter_gen(count, next, en, clk);
   parameter N = 4;
   output [N - 1 : 0] count;
   output next;
   input en;
   input clk;
   wire [N : 0] t;
   genvar i;

   assign t [0] = en;
   generate
      for(i = 0;i < N;i = i + 1) begin
         tflipflop tff (count [i], t [i], clk);
         assign t [i + 1] = t [i] & count [i];
      end
   endgenerate
   assign next = t [N];
endmodule
```

This module named `counter_gen` is parameterized by N whose default value can be anything (in this case we picked 4). This module now acts as a generator that can be run by instantiating the module with a particular parameter value. Each time this module is instantiated as part of a larger design, the width of the counter should be specified otherwise the default value is used. If this module is instantiated with `N=4`, the elaboration will create a specialized version of it which as expected will be identical to the first version we have written.

In the generic version an off-by-one error causing array bounds violations can be easily introduced if the hardware designer inadvertently wrote the `for`-loop condition as `i<=N`. Again in this case an extra `and` gate and an extra T flip-flop will be created if padding is used naively. Thankfully, current Verilog tools would reject this description but they would only do so during or after elaboration when the array bounds violations are clearly exposed (In this example, as soon as `tflipflop tff_4 (count [4], t [4], clk)` is generated).

So far we have mentioned two kind of inconsistencies: Wire width mismatches and array bounds violations. As we have seen in the previous example, these inconsistencies might not be immediately obvious in designs with generative constructs because the code can not be directly inspected. A reasonable approach would be to elaborate the code first to eliminate these constructs at which point checking for these inconsistencies is trivial. This is the approach used by current tools except that they ignore some inconsistencies that are considered legal (and synthesizable) due to padding semantics. The Icarus Verilog compiler for example will ignore matching wire widths violations and ignore array bounds violations unless the index is a constant expression (composed only of constant literals and parameters) in which case it is reported as an elaboration error. We have two strong objections here: First, waiting till elaboration to detect inconsistencies is not a good strategy for the following reasons:

- Elaboration is time consuming. Static analysis could save us a lot of time

- After elaboration a hardware description can be significantly larger which could make analysis more expensive
- When an elaboration error is found, it is hard to trace it back to the pre-elaborated design. Therefore the designer can only get a cryptic error message referring to generated code that he never wrote
- Only the specific instantiations of a module which occur in the final circuit are checked. Statically checking a generic module before elaboration will provide guarantees about all its possible instantiations. Once this is done, this module can be safely used in any design without any further checks. On the other side, elaboration errors might not show up depending on the parameter values used when instantiating a module. This means that successfully elaborating and synthesizing a circuit does not guarantee that all the modules composing it will never violate array bounds or wire width requirements if instantiated with different parameter values. As an example, consider what would happen if we had accidently forgotten to change the boundaries of `count` from `[3 : 0]` to `[N-1 : 0]` while generalizing the `counter4` module. It is clear that instantiating this module in a larger design with `N` set to any natural number less then or equal to 4 will elaborate successfully but will fail if `N` was set to a value greater than 4. What does this tell us about this module? Does it violate array bounds or not? Without enough static checking, the answer is: "It depends"! which is clearly undesirable.

The second objection is with padding semantics hiding inconsistencies. This is even worse than detecting them during or after elaboration for the following reasons:

- Verilog's padding semantics allow designers to write descriptions where wires of different widths can be interconnected. We believe that in most cases, such inconsistencies are the result of typographic errors and not the designer's intent. Going back to the `invert4` example, we believe that replacing the declaration of `y` with `input [4:0] y` should be rejected. In case the designer really want the most significant bit of `y` to be discarded, he can always modify the assign statement as follows: `assign x = ~y[3:0]`. Similarly replacing the declaration of `x` with `output [4:0] x` should be rejected forcing the designer to explicitly change the assign statement to `assign x = ~{1'b0,y}` in case he really want `y` to be zero-padded.
- The right value to pad with is often not clear. Should we pad with 0s, 1s, or just extend the most significant bit? Does it make difference if the value we are padding is a 2's complement or an unsigned number?
- Ignored violations will synthesize into potentially "incorrect" circuits. If the intention of the designer is different from the one assumed by the synthesizer, the circuit will occasionally behave incorrectly. However, it will only do so when the computation is affected by the padded/trimmed bits. This kind of errors is usually very frustrating and might remain hidden until the circuit gets manufactured
- Ignored violations might also lead to a circuit using more resources than required

The most important property of a hardware description is its synthesizability. However we are not interested in descriptions that synthesize into circuits that behave differently from the designer's intentions. Unfortunately, there is no way of capturing such intentions except by requiring the designer to explicitly express them. Verilog automatic padding semantics, although convenient when they correspond to the designer's intention, are dangerous in all other cases because they tend to hide bugs. This is why we opt to systematically reject systems relying on them. This does not limit what the designer can express because the desired bits can be explicitly appended when this is what is desired. For a description to be "meaningful" and synthesize into a "correct" circuit, wire widths must match and array bounds must be respected.

An ideal tool should be able to statically verify (once and for all) that a module is free from all inconsistencies. If the module is parameterized, the tool's target should be to verify that this is the case for all possible instantiations (for all possible parameter values). It would be possible to imagine adding some restrictions on the parameter values accepted by such modules but for the moment we don't.

1.1 Contributions

Our key contribution is a method to statically verify that a Verilog description is free from 1) Wire width mismatches 2) Array bounds violations. We combine dependent types and constraint generation to reduce the static checking problem to a satisfiability problem. Once reduced, the problem can be handed over to a standard satisfiability modulo theories (SMT) solver. We informally explain our approach by applying it to the `counter_gen` example (Section 2). We also show how the same framework allows us to detect and reject "meaningless" unreachable code (Section 2.5). This ability follows naturally from the constraint gathering approach we have to do in order to detect other inconsistencies.

We formalize our approach by embedding it in a powerful type system. To do so we extend the syntax of Featherweight Verilog (FV) to allow for explicit array width declarations (Section 4.1). FV is a core calculus of structural Verilog with generative constructs that we defined in a previous work (2). FV is a statically-typed two level language (3; 8; 11; 12) with elaboration semantics modeling elaboration and a two-level type system guaranteeing synthesizability (2). We extend FV's type system to use dependent types and to enforce the required constraints (Section 4.2).

We prove three important properties of our formalization: 1) Type preservation, 2) Type Safety, and 3) Preprocessing soundness (Section 5). Together these three theorems guarantee that a well-typed FV description will always elaborate into an inconsistency-free description with no generative constructs remaining. The complete proofs of these theorems are in the extended technical report version of this paper (10).

Finally we provide an implementation (Section 3) of these ideas in the form of a Verilog Pre-Processor that we call VPP. VPP performs elaboration on Verilog descriptions that it can prove to be well-typed and rejects all other descriptions. The output is a structural Verilog description free from any generative constructs.

2. Approach

Dependent types, recording control flow information, and generating consistency constraints are three key ingredients to reduce the problem of static consistency checking to a satisfiability problem. In the following subsections, we informally describe each of these and how everything is put together.

2.1 Dependent Types

A dependent type is a type that depends on a value. For our purposes, we use dependent types to enrich wire types with their upper and lower bounds.This follows naturally from the way arrays are declared in Verilog. For example if `x` is declared using the following statement: `input [N:M] x` then `x` has type `wire(min(N,M),max(N,M))` where `min(N,M)` is its lower bound and `max(N,M)` is its upper bound instead of having the simpler but less informative type `wire array`.

Using dependent types, the type of `count` in module `counter_gen` is `wire(min(N-1,0),max(N-1,0))`. Similarly the type of `t` in the same module is `wire(min(N,0),max(N,0))`. Types of other input/output signals is just `wire` since they are all single bit ports.

2.2 Recording Control Flow Information

The information recorded in the type is not quite all the information we need, nor is it all the information we can gather. We can gather additional information from generative constructs (both `if`-conditions and `for`-loops). In case of loops, the additional information is the loop invariant whose inference is undecidable in the general case but very easy to infer if the loop construct is restricted enough. In this work we restrict our attention to simple loops that can be easily analyzed.

In the `counter_gen` example we do not have a conditional but we do have a generative loop. We can easily infer that within the body of the loop, the loop index i is always less than N and always greater than or equal to 0.

2.3 Generating Consistency Constraints

Given the information we know about wire arrays and the information we have collected by analyzing the control flow, we now need to prove that inconsistencies never occur in a given description. To do so we generate a constraint that we need to verify for each potential inconsistency source. In particular we generate constraints whenever we encounter any of these:

- Wire Assignment: Width of the left hand side (LHS) must be equal to the width of the right hand side (RHS)
- Module Instantiation: Passed wires must have widths compatible with those in the module signature
- Array Access: Array indices are within bounds

As an example, let's take the T flip-flop instantiation statement in the body of the loop in module `counter_gen`. This statement generates 7 constraints (some of them are redundant). First we need to make sure the passed wires have widths compatible with the signature of the T flip-flop module. The T flip-flop module outputs a single bit, and requires a single bit clock and another single bit input value. Since the widths of all the passed signals is actually 1, the required constraint (generated 3 times) is $1 = 1$ and is trivially true. Second we need to make sure that `count[i]` does not cause any array bounds violation which requires 2 constraints: $i \geq min(N-1,0)$ and $i \leq max(N-1,0)$ where $min(N-1,0)$ and $max(N-1,0)$ are the lower and upper bounds of `count` as recorded in its type. Similarly the constraints $i \geq min(N,0)$ and $i \leq max(N,0)$ are generated to make sure that `t[i]` does not cause array bounds violations.

2.4 Verifying Consistency

We need to prove that each constraint holds given the type information and the collected information at this point. In general we find ourselves required to prove that the conjunction of a set of facts (givens) implies a different fact (consistency constraint) for all possible variable values. A bit more formally we need to prove the correctness of a logic formula of the form: $\forall x_1, x_2, ..., x_n . \bigwedge_{i=1}^{k} c_i \Rightarrow c$ where c_i represent known facts and c is the consistency constraint. In the following sections this same formula will be written more concisely as $\langle c_i \rangle \rhd c$

Verifying the truth of this universally quantified formula can be converted into a satisfiability problem by negating the formula and checking for its unsatisfiability. Negating the above formula and using basic logic rules we obtain a formula of the form: $\exists x_1, x_2, ..., x_n . \bigwedge_{i=1}^{k+1} c_i$ where $c_{k+1} = \neg c$. Once converted to a satisfiability problem, it can can be handed over to a standard SMT solver.

For example, let's take the $i \leq max(N,0)$ constraint generated by the instantiation statement in `counter_gen`. We basically need to prove that this always holds given what we have inferred about i. So we need to prove that: $\forall i, N.(i < N \wedge i \geq 0) \Rightarrow i \leq max(N,0)$. We can prove this by proving the unsatisfiability of its negation: $\exists i, N.(i < N \wedge i \geq 0 \wedge i > max(N,0))$. Otherwise stated we want to make sure that no integer value i can simultaneously satisfy the information we inferred from the loop invariant and be greater than the upper bound of `count`. If such a value is found then our program should be rejected.

2.5 Unreachable Code Detection

The same satisfiability framework can be used to serve a slightly different purpose. It can be used to detect unreachable code. Consider the following example:

```
module adder (sum, a, b);
   parameter N = 8;
   input [N-1 : 0] a;
   input [N-1 : 0] b;
   output [N : 0] sum;
   generate
      if(N<16)
         if(N<8)
            ripple_adder #(N) radder (sum, a, b);
         else
            cla_adder #(N) cladder (sum, a, b);
      else
         cselect_adder #(N) csadder (sum, a, b);
   endgenerate
endmodule
```

This is an N-bit adder that will add `a`, `b` and produce their sum. Internally this adder is either a ripple, a carry look ahead, or a carry select adder depending on the parameter value used to instantiate it. Note that the value 8 assigned to `N` is only a default value, any different value can be used when the module is actually instantiated as part of a bigger design. What if the condition `N<16` is replaced by `N>16`? The module still means something except that `ripple_adder #(N) radder (sum, a, b)` will be unreachable as `N` cannot be greater than 16 and less then 8 at the same time. If such code is presented it usually indicates a bug and should be pointed out or even better it should be rejected as it is "meaningless". Clearly this can be done using the same framework we just described. All we need to do is make sure that all the facts collected by analyzing loops and conditionals are consistent with each other.

3. Implementation

A prototype implementation of the Verilog Pre-Processor (VPP) is available for download at `http://www.resource-aware.org/twiki/bin/view/RAP/VPP`. VPP includes a type checker based on the typing rules defined in Section 4.2. If the given description is well-typed then it is elaborated into an equivalent description that is guaranteed to be inconsistency free.

To check for satisfiability conditions, VPP makes use of Yices (1), a state of the art SMT solver. Since VPP is developed in OCaml, we developed an OCaml library to communicate with the C APIs provided by Yices. Yices is needed in 2 different situations:

1. When gathering constraints from conditionals and loops: Each new piece of information must not conflict with previously collected information. Each time the set of collected constraints is extended, VPP passes the extended set to Yices to verify its satisfiability. In case it is unsatisfiable, the construct causing

the extension is considered ill-typed as it guards an unreachable block of code.

2. When verifying consistency requirements: Whenever a new consistency requirement is reached, VPP passes its negation along with the set of collected constraints to Yices looking for unsatisfiability. In case the extended set is satisfiable, the construct being type-checked is considered ill-types as it violates one of the consistency constraints.

Figure 2 illustrates the interaction between VPP and Yices. Given an unelaborated description, VPP does static type checking and a big part of that is generating a bunch of integer satisfiability problems that are handed over to Yices. VPP finally decides the well-typedness of the description based on its own typing rules and the set of Yices responses.

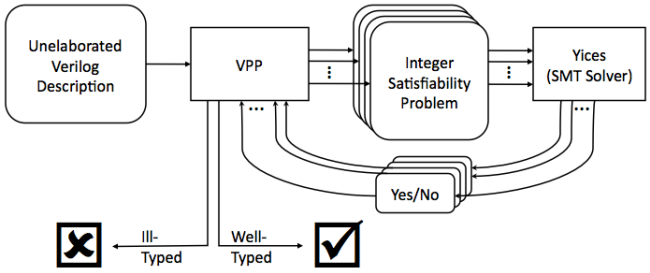


Figure 2. VPP Interaction with Yices

In particular VPP uses the integer linear arithmetic and the uninterpreted function theories of Yices. The integer linear arithmetic theory is used in various places, especially when translating Verilog expressions into Yices expressions. The uninterpreted functions have been very useful to define non-linear functions like min and max. For example here is how we define min in Yices:

```
(define min:: (-> x::int y::int
                  (subtype(r::int)
                          (and (or (= r x)(= r y))
                               (<= r x)
                               (<= r y)))))
```

We define min as an uninterpreted function that takes two integers `x` and `y` and returns an integer `r` such that its value is either equal to `x` or `y` while being at the same time less than or equal to both.

3.1 Limitations

A more subtle issue that we encountered is the fact that converting Verilog expressions into Yices expressions is not always straight forward. Not all non-linearities can be handled using uninterpreted function as nicely as min and max were handled. When dealing with a description whose consistency cannot be verified statically due to non-linearity, we can either reject it or accept it with a warning and then perform additional checks during elaboration. The current implementation will reject such descriptions but this might be changed in the future if need arises.

4. Featherweight Verilog

To be able to prove that our approach is reasonable, we need to formalize it and prove that it can indeed be used to statically check that a description is free from the aforementioned inconsistencies. We basically want to define a type system that will only consider inconsistency-free descriptions to be well typed. In a previous work (2) we defined FV, a core calculus of structural Verilog with generative constructs. We opted for defining FV as a statically-typed two level language to be able to naturally model the elaboration phase, which was defined using big-step operational semantics. In (2) we proved three key properties of FV:

- Type Preservation: The resulting description obtained from elaborating a well-typed description is well-typed
- Type Safety: Preprocessing a well-typed description will never fail
- Preprocessing Soundness: After elaboration a description is guaranteed to be free from any generative constructs

Combining these results and by defining synthesizability we were able to show that a well-typed description was guaranteed to be synthesizable. This means that we were able define a type system capable of statically checking the synthesizability of a description

In this paper we extend FV definitions to allow us to statically check for inconsistencies. In the following subsections we present the required extensions to FV syntax, operational semantics and type system. To make these extensions easier to spot and focus on, we highlight them in gray. As expected the most interesting changes occur in the type system. We also prove that the key properties of FV still hold.

4.1 FV Syntax

In this section we reprise the FV grammar definition from (2) with minor modifications. The abstract syntax for FV makes use of the following meta-variables:

Module	m	$\in$	ModuleNames
Signal	s	$\in$	IdentifierNames
Elaboration Variable	x, y	$\in$	ParameterNames
Operator	f	$\in$	$\mathbb{O}$
Index	h, i, j, k, q, r	$\in$	$\mathbb{N}$
Index Domain	H, I, J, K, Q, R	$\subseteq$	$\mathbb{N}$

where ModuleNames, IndentifierNames, and ParameterNames are countably infinite sets used to draw modules, signals, and parameters names respectively. $\mathbb{O}$ is the finite set of operator names, and $\mathbb{N}$ is the set of natural numbers. The full grammar for FV is defined as follows:

Circuit Description	p	::=	$\langle D_i \rangle^{i \in I}\ m$
Module Definition	D	::=	$\texttt{module}\ m\ b$
Module Body	b	::=	$\langle x_i \rangle^{i \in I}\ \langle d_j\ t_j\ s_j \rangle^{j \in J}\ \texttt{is}$ $\langle t_k\ s_k \rangle^{k \in K}\ \langle P_r \rangle^{r \in R}$
Direction	d	$\subseteq$	$\{\texttt{in}, \texttt{out}\}$
Type	$t \in \mathbb{T}$	::=	$\texttt{wire} \mid \texttt{int} \mid t(e, e)$
Parallel Statement	P	::=	$m\ \langle e_i \rangle^{i \in I}\ \langle l_j \rangle^{j \in J} \mid \texttt{assign}\ l\ e$ $\mid \texttt{if}\ e\ \texttt{then} \langle P_i \rangle^{i \in I} \texttt{else} \langle P_j \rangle^{j \in J}$ $\mid \texttt{for}(y = e; y < e; y = y + e)\ \langle P_i \rangle^{i \in I}$
LHS value	l	::=	$s \mid s[e] \mid s[e : e]$
Expression	e	::=	$l \mid x \mid v \mid f\langle e_i \rangle^{i \in I}$
Value	v	::=	$(0 \mid 1)^+$

A circuit description p is a sequence of module definitions $\langle D_i \rangle^{i \in I}$ followed by a module name m. The module name indicates which module from the preceding sequence represents the overall input and output of the system. A module definition is a name m and a module body b. A module body itself consists of fours sequences: (1) module parameter names $\langle x_i \rangle^{i \in I}$, (2) port declarations (carrying direction, type, and name for each port) $\langle d_j\ t_j\ s_j \rangle^{j \in J}$, (3) local variable declarations $\langle t_k\ s_k \rangle^{k \in K}$, and (4) parallel statements $\langle P_r \rangle^{r \in R}$. A port direction indicates whether the port is input, output, or bidirectional. The type of a local variable can be `wire`, `int`, or an array with upper and lower bounds. A

parallel statement can be a module instantiation, an `assign` statement, a conditional statement, or a `for`-loop. A module instantiation specifies module parameters $\langle e_i \rangle^{i \in I}$ as well as port connections $\langle l_j \rangle^{j \in J}$. An `assign` statement consists of a left hand side (LHS) value l and an expression e. An LHS value is either a variable s, an array lookup $s[e]$, or an array range $s[e : e]$. An expression is either an LHS value l, a parameter name x, an integer v, or an operator application $f\langle e_i \rangle^{i \in I}$.

The main difference from the syntax defined in (2) is that wire declarations have explicit upper and lower bounds associated with them unless they are single bit wires. This in turn requires both module ports and local variables to have types associated with them. Making these explicit makes FV closer to the original Verilog syntax. We had omitted those previously because we were not concerned by them for synthesizability issues. It is important to note that in FV syntax (just like in Verilog syntax) the expressions used to specify the array bounds can be in any order. The first expression does not necessarily correspond to the lower bound. It might correspond to the upper bound as well. This is different from the convention we use in our type system that always assumes that the first expression is the lower bound expression and that the second one corresponds to the upper bound.

The second difference is that `for`-loop headers are syntactically restricted in a way that makes loop invariants easy to infer. Finally, integer values are not restricted to 32 bit values. An integer is now a non-empty sequence of 0s and 1s of arbitrary length.

As a notational convenience, we also define a general term X that is used to range over all syntactical constructs of FV as follows:

$$\textit{Term} \qquad X \quad ::= \quad p \mid D \mid b \mid P \mid l \mid e$$

In the formal treatment of FV we use standard notation. For detailed notational conventions or detailed comparison between FV syntax and the syntax of the corresponding Verilog subset, we refer the reader to (2)

4.2 FV Type System

As previously defined, the used type system is a two-level type system. In the terminology of two-level languages, preprocessing is the level 0 computation, and the result after preprocessing is the level 1 computation that is performed by the circuit. In a Verilog description, there are relatively few places where preprocessing is required. In FV, these are restricted to four places: 1) expressions passed as module parameters, 2) conditional expressions in `if` statements, 3) expressions that relate to the bounds on `for`-loops, and 4) array indices.

By convention, the typing judgment (generally of the form $\Delta \vdash X$) will be assumed to be a level 1 judgment. That is, it is checking for validity of a description that has already been preprocessed. Expressions, however, may be computations that either are performed during expansion or remain intact to become part of the preprocessed description. For this reason, the judgment for expressions will be annotated with a level $n \in \{0, 1\}$ to indicate whether we are checking the expression for validity at level 0 or at level 1. This annotation will appear in the judgment as a superscript on the turnstyle, i.e. $\vdash^n$.

To define the type system we need the following auxiliary notions:

$$\begin{array}{llll}
\textit{Module Type} & M & ::= & \langle x_i \rangle^{i \in I}\ \langle d_j\ t_j \rangle^{j \in J} \\
\textit{Operator Signatures} & \Sigma & \in & \Pi i.\ \mathbb{O} \times \mathbb{N} \times \mathbb{T}^i \to \mathbb{T} \\
 & & & \\
\textit{Level} & n & ::= & 0 \mid 1 \\
\textit{Module Environment} & \Delta & ::= & [\,] \mid m : M\ ::\ \Delta \\
\textit{Variable Environment} & \Gamma & ::= & [\,] \mid s : d\ t\ ::\ \Gamma \mid x : d\ t\ ::\ \Gamma \\
\textit{Level 1 Variable Env.} & \Gamma^+ & ::= & [\,] \mid s : d\ t\ ::\ \Gamma^+ \\
 & & & \\
\textit{Constraint Environment} & C & ::= & [\,] \mid c\ ::\ C \\
\textit{Constraint} & c & ::= & e\,R\,e \mid \neg\, c \\
\textit{Relational Operator} & R & ::= & < \mid > \mid \leq \mid \geq \mid = \mid \neq
\end{array}$$

A module type now consists of a sequence of parameters and a sequence of directions and types for ports. These types might be dependent on the parameters values. The rest is unchanged: An operator signature is a function that takes an operator, the level at which the operation is executed, and the types of the operands and returns the type of the result. As noted above, levels can be 0 or 1. A module environment associates module names with their corresponding types while a variable environment associates variable names with their corresponding directions and types. We do not have to keep level information in the variable environment because we can differentiate between levels syntactically. All signals and declared local variables (denoted by s) are level 1 variables while parameters and for-loop variables (denoted by x or y) are level 0 variables.

We also added a constraint environment C which is a list of constraints c known to be true. A constraint can be a relation between two verilog expressions, or a negation of another constraint.

To make the typing rules more readable, we recursively define a function $width$ that returns the width given type information:

$$\begin{array}{lcl}
width(int) & = & \infty \\
width(wire) & = & 1 \\
width(t(e_1, e_2)) & = & (e_2 - e_1 + 1) * width(t)
\end{array}$$

Note that width of `int` is considered to be infinite (or unbounded). This means that integer variables like parameters can be of arbitrary sizes and therefore can never be assigned to wires. The width of a `wire` is 1 and the width of an array of elements of width w is equal to w multiplied by the array size, which is the upper bound - lower bound + 1. Note that at this point we assume that the boundaries are ordered. This is achieved using a function $order$ that returns a type where bounds are ordered given a type with no restriction on the order of bounds. Here is the recursive definition of $order$:

$$\begin{array}{lcl}
order(int) & = & int \\
order(wire) & = & wire \\
order(t(e_1, e_2)) & = & order(t)(min(e_1, e_2), max(e_1, e_2))
\end{array}$$

For types that do not have bounds, $order$ is just an identity function otherwise $order$ makes sure that the lower bound (which the minimum of the given bounds) is presented before the lower bound (which is the maximum of the given bounds).

Figure 3 defines the rules for the judgment $\vdash p$. A circuit description p is well-typed when this judgment is derivable. These rules are the same as presented in (2) except for the highlighted parts. These modifications where done for three main reasons: 1) The type system now requires dependent types. 2) It is also required to maintain an additional environment C of given constraints. 3) Typing rules now have additional premises to guarantee consistency.

For a detailed explanation of each typing rule, we refer the reader to (2). The modification in T-Prog is due to the module signature change. In addition to the trivial changes, T-Body now requires the types of ports and local variables to be well-typed themselves in an environment that only contains parameters. The typing judg-

ment for types is added in the end of Figure 3. T-Mod now requires the widths of signals passed to modules to be compatible with the instantiated module signature after substitution. T-Assign1 and T-Assign2 now check for wire width mismatches. T-If appends the known constraints environment depending on the branch. Similarly T-For appends the constraint environment with the loop environment. To make sure the loop terminates, e_3 is required to be greater than 0 and e_2 is not allowed to use the loop index y. T-Idx and T-Rg add the necessary conditions to check for array bounds violations.

This type system could also be used to detect unreachable code although no explicit extensions need to be added. All we need to do is to change C's definition to be:

$$\textit{Satisfiable Constraint Environment} \quad C \quad ::= \quad [\,] \mid c :: C$$

A satisfiable constraint environment cannot contain conflicting constraints.

4.3 FV Operational Semantics

In (2) we formalized the elaboration process by defining a big-step operational semantics indexed by the level of the computation to formally specify the preprocessing phase. The specification dictates how expansion should be performed, what the form of the preprocessed circuit descriptions should be, and what errors can occur during preprocessing.

To model the possibility of errors during preprocessing, we defined the following auxiliary notion:

$$\textit{Possible Term} \qquad X_\perp \quad ::= \quad X \mid \texttt{err}$$

This allows us to write $p_\perp$ or $e_\perp$ to denote a value that may either be the constant `err` or a value from p or e, respectively.

Preprocessing is defined by the derivability of judgments of the general form $\langle D_i \rangle \vdash X \overset{n}{\hookrightarrow} X_\perp, \langle D_j \rangle$. Intuitively, preprocessing takes a sequence of module declarations $\langle D_i \rangle$ and a term X and produces a new sequence of specialized modules $\langle D_j \rangle$ and a possible term $X_\perp$.

The required notion of substitution is extended to define substitution inside types, type environments, and constraint environments. Most of the operational semantics remain unchanged except for minor adjustments to accommodate for the syntax change. The only notable change is in the module instantiation rule (E-Mod) where the types of the newly created module need to be substituted into to reflect the values of the parameters that were used in the instantiation. We display the substitution definition with the required extensions in Figure 4 and the definition of the operation semantics in Figure 5. The potential errors of operational semantics errors are defined in (10).

5. Technical Results

In this section we restate the main theorems from our previous work and show that they still hold. The complete proofs of these theorems are in (10).

We first need to define the substitution lemma. Since we only need to define substitution on parallel statements, we state the substitution lemma as follows:

Lemma 1 (Substitution). *If* $\Delta; \Gamma, x : d\ t\ ;C \vdash P$ *and* $\Gamma\ ;C \vdash^n v : d\ t$ *and* $C, x = v$ *is satisfiable then* $\Delta; \Gamma[x \mapsto v]\ ;C \vdash P[x \mapsto v]$

Sketch. The proof proceeds by induction on the derivation of the first judgment. □

Using this lemma, we show that preprocessing of a well-typed description produces a well-typed description. Formally:

Theorem 1 (Type Preservation). *If* $\vdash p$ *and* $p \overset{1}{\hookrightarrow} p'$ *then* $\vdash p'$

Sketch. The proof proceeds by induction on the derivation of the second judgment. □

We also want to prove type safety:

Theorem 2 (Type Safety). *If* $\vdash p$ *and* $p \overset{1}{\hookrightarrow} p'$ *then* $p' \neq \texttt{err}$

Sketch. This result follows directly from Theorem 1 since no typing rules will consider `err` well-typed. □

Finally we want to prove preprocessing soundness, which means that preprocessing produces fully expanded terms. The set of fully expanded terms is defined as follows:

$$\begin{aligned}
&\textit{Expanded Term}\ \ \hat{X} \ = \\
&\quad \{u \mid u \in X_\perp \wedge Y \in \mathit{subterms}(u) \Rightarrow \\
&\qquad ((Y = \langle x_i \rangle^{i \in I}\ \langle d_j\ t_j\ s_j \rangle^{j \in J}\ \texttt{is}\ \langle t_k\ y_k \rangle^{k \in K}\ \langle P_r \rangle^{r \in R} \Rightarrow I = \emptyset) \\
&\qquad \wedge (Y = m\ \langle e_i \rangle^{i \in I}\ \langle l_j \rangle^{j \in J}\ \Rightarrow I = \emptyset) \\
&\qquad \wedge (Y \neq \texttt{if}\ e\ \texttt{then} \langle P_i \rangle^{i \in I} \texttt{else} \langle P_j \rangle^{j \in J}) \\
&\qquad \wedge (Y \neq \texttt{for}(y = e; e; y = e) \langle P_i \rangle^{i \in I}))\}
\end{aligned}$$

Theorem 3 (Preprocessing Soundness). *If* $p \overset{1}{\hookrightarrow} p'$ *then* $p' \in \hat{p}$

Sketch. The proof proceeds by induction on the derivation of the first judgment. □

6. Related Work

To the best of our knowledge, no one has considered Verilog static checking before. Current tools do perform some checks but these are mostly done after elaboration. On the other hand, various techniques for static array bounds checking for general purpose languages have been explored earlier. Static buffer overflow analysis for C-like programming languages has been extensively studied. Checking legacy code is usually very expensive as it requires rather complex inter-procedural analysis. Examples include Polyspace Ada/C/C++ Verifier (13), C Global Surveyor(14). This approach is of limited scalability. An alternative approach would be to explicitly specify the preconditions and postconditions of each function using special annotations. When this information is available, the checker is only required to check each function independently making the task of detecting violations much more precise and scalable. Many annotation-based approaches have been proposed in the literature. Splint(7) is such an example. It does lightweight analysis but it is neither sound nor complete. ESPX(4) is sound and complete and uses an incremental annotation approach. If the code is fully annotated, it is comprehensively checked for buffer overflows. ESPX also uses SALInfer an annotation inference engine that facilitates the annotation process by automating some of it. Annotation based approaches are not really useful until the programmer has manually annotated the source code which requires him to learn the annotation language and spend a significant amount of time actually adding the annotations.

Our type checker does not need to do inter-modular analysis and it does not require the programmer to put any additional annotations. The reason behind that is that each module has a well defined signature that specifies its inputs, outputs and the boundaries of each of these. This readily available information provides us with the needed pre- and post-conditions. Also we don't want to rely on how a module is instantiated to decide its well-typedness.

$\boxed{\vdash p}$

$$\frac{\vdash \langle D_i \rangle : \Delta \qquad \Delta(m) = \ \langle\rangle \ \langle d_i \ t_i \ \rangle}{\vdash \langle D_i \rangle \ m} \ (\mathrm{T-Prog})$$

$\boxed{\Delta \vdash \langle D_i \rangle : \Delta}$

$$\frac{}{\Delta \vdash \langle\rangle : [\,]} \ (\mathrm{T-MEmpty}) \qquad \frac{\Delta \vdash b : M \qquad m : M :: \Delta \vdash \langle D_i \rangle : \Delta'}{\Delta \vdash \texttt{module}\ m\ b :: \langle D_i \rangle \ : \ \ m : M :: \Delta'} \ (\mathrm{T-MSeq})$$

$\boxed{\Delta \vdash b : M}$

$$\frac{\begin{array}{l}\{d_j \neq \emptyset\} \qquad \{\Delta; \Gamma; \ \langle\rangle \ \vdash P_r\} \\ \Gamma = \langle x_i : \{\texttt{in}\}\ \texttt{int} \rangle \uplus \langle s_j : d_j \ \mathit{order}(t_j) \ \rangle \uplus \langle s_k : \{\texttt{in}, \texttt{out}\} \ \mathit{order}\ (t_k) \rangle \\ \{\langle x_i : \{\texttt{in}\}\ \texttt{int} \rangle \vdash t_j\} \qquad \{\langle x_i : \{\texttt{in}\}\ \texttt{int} \rangle \vdash t_k\}\end{array}}{\Delta \vdash \langle x_i \rangle \ \langle d_j \ t_j \ s_j \rangle \ \texttt{is} \ \langle t_k \ s_k \rangle \ \langle P_r \rangle : \ \langle x_i \rangle \ \ \langle d_j \ t_j \ \rangle} \ (\mathrm{T-Body})$$

$\boxed{\Delta; \Gamma \ ; C \ \vdash P}$

$$\frac{\begin{array}{l}\{\Gamma \ ; C \ \vdash^0 e_i : \{\texttt{in}\}\ \texttt{int}\} \qquad \Delta(m) = \ \langle x_i \rangle \ \langle d_j \ t_j \ \rangle \\ \{\Gamma \ ; C \ \vdash^1 l_j : d'_j \ t'_j\} \qquad \{d_j \subseteq d'_j\} \\ \{C \rhd \mathit{width}(t_j\{[x_i \mapsto e_i]\}) = \mathit{width}(t'_j)\}\end{array}}{\Delta; \Gamma \ ; C \ \vdash m\langle e_i \rangle \ \langle l_j \rangle} \ (\mathrm{T-Mod})$$

$$\frac{\begin{array}{l}\texttt{out} \in d_1 \qquad \texttt{in} \in d_2 \\ \Gamma \ ; C \ \vdash^1 l : d_1 \ t_1 \\ C \rhd \mathit{width}(t_1) = i\end{array}}{\Delta; \Gamma \ ; C \ \vdash \texttt{assign}\ l \ (0 \,|\, 1)^i} \ (\mathrm{T-Assign1}) \qquad \frac{\begin{array}{l}e \neq (0|1)^+ \\ \texttt{out} \in d_1 \qquad \texttt{in} \in d_2 \\ \Gamma \ ; C \ \vdash^1 l : d_1 \ t_1 \\ \Gamma \ ; C \ \vdash^1 e : d_2 \ t_2 \\ C \rhd \mathit{width}(t_1) = \mathit{width}(t_2)\end{array}}{\Delta; \Gamma \ ; C \ \vdash \texttt{assign}\ l \ e} \ (\mathrm{T-Assign2})$$

$$\frac{\begin{array}{l}\Gamma \ ; C \ \vdash^0 e : \{\texttt{in}\}\ \texttt{int} \\ \{\Delta; \Gamma \ ; C, e \ \vdash P_i\} \qquad \{\Delta; \Gamma \ ; C, \neg e \ \vdash P_j\}\end{array}}{\Delta; \Gamma \ ; C \ \vdash \texttt{if}\ e\ \texttt{then}\langle P_i \rangle \texttt{else}\langle P_j \rangle} \ (\mathrm{T-If})$$

$$\frac{\begin{array}{l}\Gamma, y : \{\texttt{in}\}\ \texttt{int} \ ; C \ \vdash^0 e_3 : \{\texttt{in}\}\ \texttt{int} \\ \Gamma \ ; C \ \vdash^0 e_1 \ , e_2 \ : \{\texttt{in}\}\ \texttt{int} \qquad \{\Delta; \Gamma, y : \{\texttt{in}\}\ \texttt{int} \ ; C, y \geq e_1, y < e_2 \ \vdash P_i\} \\ C \rhd e_3 > 0\end{array}}{\Delta; \Gamma; \ C \ \vdash \texttt{for}(\ y = e_1; y < e_2; y = y + e_3 \ )\langle P_i \rangle} \ (\mathrm{T-For})$$

$\boxed{\Gamma \ ; C \ \vdash^n l : d \ t}$ See $\Gamma \ ; C \ \vdash^n e : d \ t$

$\boxed{\Gamma \ ; C \ \vdash^n e : d \ t}$

$$\frac{\Gamma(s) = d \ t}{\Gamma \ ; C \ \vdash^1 s \ : \ d \ t} \ (\mathrm{T-Id}) \qquad \frac{\begin{array}{l}\Gamma(s) = d \ t \ \ (e_l, e_u) \\ \Gamma \ ; C \ \vdash^0 e : \{\texttt{in}\}\ \texttt{int} \\ C \rhd e \geq e_l \\ C \rhd e \leq e_u\end{array}}{\Gamma \ ; C \ \vdash^1 s[e] : d \ t} \ (\mathrm{T-Idx}) \qquad \frac{\begin{array}{l}\Gamma(s) = d \ t \ \ (e_l, e_u) \\ \Gamma \ ; C \ \vdash^0 e_1, e_2 : \{\texttt{in}\}\ \texttt{int} \\ C \rhd e_1 \geq e_l \qquad C \rhd e_1 \leq e_u \\ C \rhd e_2 \geq e_l \qquad C \rhd e_2 \leq e_u\end{array}}{\Gamma \ ; C \ \vdash^1 s[e_1 \ : \ e_2] : d \ t} \ (\mathrm{T-Rg})$$

$\boxed{\Gamma \vdash t}$

$$\frac{\Gamma(x) = \{\texttt{in}\}\ \texttt{int}}{\Gamma \ ; C \ \vdash^n x \ : \ \{\texttt{in}\}\ \texttt{int}} \ (\mathrm{T-Par}) \qquad \frac{}{\Gamma \ ; C \ \vdash^n v : \{\texttt{in}\}\ \texttt{int}} \ (\mathrm{T-Int}) \qquad \frac{\{\Gamma \ ; C \ \vdash^n e_i : \{\texttt{in}\}\ t_i\}}{\Gamma \ ; C \ \vdash^n f\langle e_i \rangle : \{\texttt{in}\}\ \Sigma|\langle e_i \rangle|(f, n, \langle t_i \rangle)} \ (\mathrm{T-Op})$$

$$\frac{}{\Gamma \vdash \texttt{int}} \ (\mathrm{T-TInt}) \qquad \frac{}{\Gamma \vdash \texttt{wire}} \ (\mathrm{T-TWire}) \qquad \frac{\Gamma; \langle\rangle \vdash^0 e_1, e_2 : \{\texttt{in}\}\texttt{int}}{\Gamma \vdash t(e_1, e_2)} \ (\mathrm{T-TArray})$$

Figure 3. Type System

According to our definitions, a module's well-typedness is independent from how it is used and that is why a well-typed parameterized circuit description can safely be used in any design without further checking.

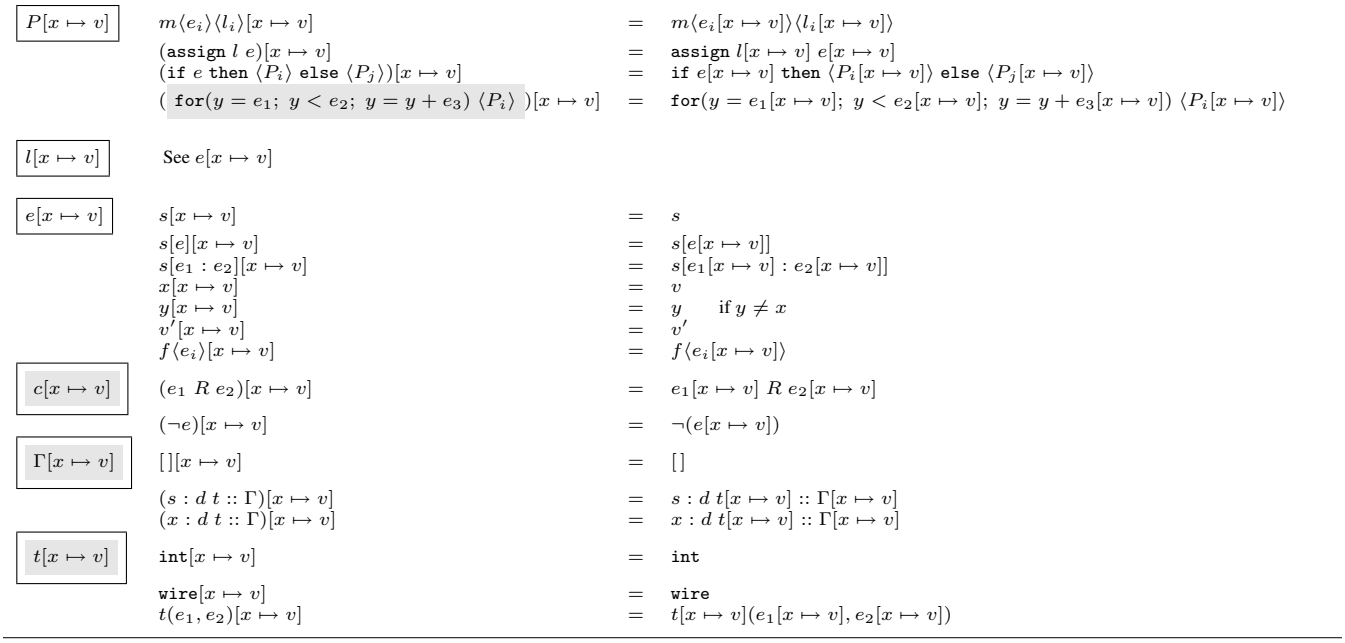


Figure 4. Substitution

Our approach was inspired by Xi and Pfenning's approach in DML (15), which is a dependently typed extension of ML. The main differences that distinguish our work are:

- DML uses type inference requiring two passes over the code: The first to infer types and the second to generate constraints based on dependent types annotations explicitly added by the programmer. FV does not require the type inference pass or any annotations as all the types and their associated widths are explicitly declared.
- FV is formalized as a two-level language while DML is not. Some of the challenges we encountered in the formalization process were due to the two-level nature of FV.
- The consistency checks needed for Verilog are different and emerge from understanding hardware constraints.
- DML uses Fourier Variable Elimination for constraint solving while we use a more general SMT solver which gives us more power because of the availability of other theories.

7. Conclusions and Future Work

In this paper we have shown how dependent types can be combined with constraint generation to create a type system that is powerful enough to statically detect array bounds violations, wire assignment inconsistencies, and unreachable code. We have proved type preservation and safety of our type system with respect to elaboration and proved the soundness of elaboration itself. Nowadays it is still the case that hardware designers find themselves making heavy use of Perl scripts or emacs advanced editing modes to help them generate hardware modules of various sizes. These techniques are hard to read, reason about, and verify. VPP (or any other tool implementing the main ideas in our paper) allows designers to safely describe circuit families using structural and generative constructs.

An important goal of this project is to increase the productivity of hardware designers by giving them means to describe circuits at a higher level of abstraction while still giving them enough control on the generated hardware. As such we would like to take a closer look at examples that are hard to express using the currently available generative constructs and add constructs that enable us to describe them in a convenient manner. These constructs will be expanded away by VPP to make sure the final outcome is in standard Verilog to allow integration with current tool chains.

We also believe that the power of dependent types is even bigger and that it can be used to capture physical properties of the hardware design. In particular we would like to be able to use the current framework to estimate the number of gates, the area, and the power required by a certain description.

References

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

[2] Jennifer Gillenwater, Gregory Malecha, Cherif Salama, Angela Yun Zhu, Walid Taha, Jim Grundy, and John 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*, pages 41–50, New York, NY, USA, 2008. ACM.

[3] Carsten K. Gomard and Neil D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69, 1991.

[4] Brian Hackett, Manuvir Das, Daniel Wang, and Zhe Yang. Modular checking for buffer overflows in the large. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 232–241, New York, NY, USA, 2006. ACM.

[5] IEEE Standards Board. *IEEE Standard Verilog Hardware Description Language*. Number 1364-2001 in IEEE Standards. IEEE, 2001.

[6] IEEE Standards Board. *IEEE Standard for Verilog Hardware Description Language*. Number 1364-2005 in IEEE Standards. IEEE, 2005.

[7] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, pages 14–14, Berkeley, CA, USA, 2001. USENIX Association.

[8] Flemming Nielson and Hanne Riis Nielson. Two-level semantics and code generation. *Theoretical Computer Science*, 56(1):59–133, 1988.

$\boxed{p \overset{1}{\hookrightarrow} p_\perp}$

$$\frac{\texttt{module}\ m\ b \in \langle D_i\rangle \qquad \langle D_i\rangle \vdash b \overset{1}{\hookrightarrow} b', \langle D_r\rangle}{\langle D_i\rangle\, m \overset{1}{\hookrightarrow} \langle D_r\rangle \uplus \langle \texttt{module}\ m\ b'\rangle\, m}\ (\mathrm{E-Prog})$$

$\boxed{\langle D\rangle \vdash b \overset{1}{\hookrightarrow} b_\perp, \langle D\rangle}$

$$\frac{\{\langle D_i\rangle \vdash P_k \overset{1}{\hookrightarrow} \langle P_r\rangle^{r\in R(k)}, \langle D_h\rangle^{h\in H(k)}\}}{\langle D_i\rangle \vdash \langle\rangle\langle d_j\ t_j\ s_j\rangle\ \texttt{is}\ \langle t_q\ s_q\rangle\langle P_k\rangle \overset{1}{\hookrightarrow} \langle\rangle\langle d_j\ t_j\ s_j\rangle\ \texttt{is}\ \langle t_q\ s_q\rangle\ \biguplus_k \langle P_r\rangle^{r\in R(k)}, \biguplus_k \langle D_h\rangle^{h\in H(k)}}\ (\mathrm{E-Body})$$

$\boxed{\langle D\rangle \vdash P \overset{1}{\hookrightarrow} \langle P_\perp\rangle, \langle D\rangle}$

$$\frac{\begin{array}{l}\{e_i \overset{0}{\hookrightarrow} v_i\} \qquad \{l_j \overset{1}{\hookrightarrow} l'_j\} \qquad \texttt{module}\ m\ \langle x_i\rangle\ \langle d_j\ t_j\ s_j\rangle\ \texttt{is}\ \langle t_q\ s_q\rangle\ \langle P_k\rangle \in \langle D_i\rangle \\ \{\langle D_i\rangle \vdash P_k\{[x_i \mapsto v_i]\} \overset{1}{\hookrightarrow} \langle P_r\rangle^{r\in R(k)}, \langle D_h\rangle^{h\in H(k)}\} \qquad m' \notin \langle D_i\rangle \qquad m' \notin \biguplus_k \langle D_h\rangle^{h\in H(k)}\end{array}}{\begin{array}{l}\langle D_i\rangle \vdash m\langle e_i\rangle\langle l_j\rangle \\ \overset{1}{\hookrightarrow} \langle m'\langle\rangle\langle l'_j\rangle\rangle, \langle \texttt{module}\ m'\ \langle\rangle\ \langle d_j\ t_j\{[x_i \mapsto v_i]\}\ s_j\rangle\ \texttt{is}\ \langle t_q\ \{[x_i \mapsto v_i]\}\ s_q\rangle\ \biguplus_k \langle P_r\rangle^{r\in R(k)}\rangle \uplus \biguplus_k \langle D_h\rangle^{h\in H(k)}\end{array}}\ (\mathrm{E-Mod})$$

$$\frac{l \overset{1}{\hookrightarrow} l' \qquad e \overset{1}{\hookrightarrow} e'}{\langle D_i\rangle \vdash \texttt{assign}\ l\ e \overset{1}{\hookrightarrow} \langle \texttt{assign}\ l'\ e'\rangle, \langle\rangle}\ (\mathrm{E-Assign})$$

$$\frac{e \overset{0}{\hookrightarrow} v \qquad v \neq 0^+ \qquad \{\langle D_i\rangle \vdash P_k \overset{1}{\hookrightarrow} \langle P_r\rangle^{r\in R(k)}, \langle D_h\rangle^{h\in H(k)}\}}{\langle D_i\rangle \vdash \texttt{if}\ e\ \texttt{then}\ \langle P_k\rangle\ \texttt{else}\ \langle P_j\rangle \overset{1}{\hookrightarrow} \biguplus_k \langle P_r\rangle^{r\in R(k)}, \biguplus_k \langle D_h\rangle^{h\in H(k)}}\ (\mathrm{E-IfTrue})$$

$$\frac{e \overset{0}{\hookrightarrow} 0^+ \qquad \{\langle D_i\rangle \vdash P_j \overset{1}{\hookrightarrow} \langle P_r\rangle^{r\in R(j)}, \langle D_h\rangle^{h\in H(j)}\}}{\langle D_i\rangle \vdash \texttt{if}\ e\ \texttt{then}\ \langle P_k\rangle\ \texttt{else}\ \langle P_j\rangle \overset{1}{\hookrightarrow} \biguplus_j \langle P_r\rangle^{r\in R(j)}, \biguplus_j \langle D_h\rangle^{h\in H(j)}}\ (\mathrm{E-IfFalse})$$

$$\frac{\begin{array}{l}e_1 \overset{0}{\hookrightarrow} v_1 \qquad e_2 \overset{0}{\hookrightarrow} v_2 \qquad v_1 < v_2 \\ \{\langle D_i\rangle \vdash P_k[y \mapsto v_1] \overset{1}{\hookrightarrow} \langle P_r\rangle^{r\in R(k)}, \langle D_h\rangle^{h\in H(k)}\} \\ \langle D_i\rangle \vdash \texttt{for}(y = v_1 + e_3[y \mapsto v_1];\ y < e_2;\ y = y + e_3)\ \langle P_k\rangle \overset{1}{\hookrightarrow} \langle P_j\rangle, \langle D_q\rangle\end{array}}{\langle D_i\rangle \vdash \texttt{for}(y = e_1;\ y < e_2;\ y = y + e_3)\ \langle P_k\rangle \overset{1}{\hookrightarrow} \biguplus_k \langle P_r\rangle^{r\in R(k)} \uplus \langle P_j\rangle, \biguplus_k \langle D_h\rangle^{h\in H(k)} \uplus \langle D_q\rangle}\ (\mathrm{E-ForTrue})$$

$$\frac{e_1 \overset{0}{\hookrightarrow} v_1 \qquad e_2 \overset{0}{\hookrightarrow} v_2 \qquad v_1 \geq v_2}{\langle D_i\rangle \vdash \texttt{for}(y = e_1;\ y < e_2;\ y = y + e_3)\ \langle P_k\rangle \overset{1}{\hookrightarrow} \langle\rangle, \langle\rangle}\ (\mathrm{E-ForFalse})$$

$\boxed{l \overset{1}{\hookrightarrow} l_\perp}$

$\boxed{e \overset{1}{\hookrightarrow} e_\perp}$

See $e \overset{1}{\hookrightarrow} e_\perp$

$$\frac{}{s \overset{1}{\hookrightarrow} s}\ (\mathrm{E-Id}) \qquad \frac{e \overset{0}{\hookrightarrow} v}{s[e] \overset{1}{\hookrightarrow} s[v]}\ (\mathrm{E-Idx}) \qquad \frac{e_1 \overset{0}{\hookrightarrow} v_1 \qquad e_2 \overset{0}{\hookrightarrow} v_2}{s[e_1 : e_2] \overset{1}{\hookrightarrow} s[v_1 : v_2]}\ (\mathrm{E-Rg})$$

$$\frac{}{v \overset{1}{\hookrightarrow} v}\ (\mathrm{E-Int1}) \qquad \frac{\{e_i \overset{1}{\hookrightarrow} e'_i\}}{f\langle e_i\rangle \overset{1}{\hookrightarrow} f\langle e'_i\rangle}\ (\mathrm{E-Op1})$$

$\boxed{e \overset{0}{\hookrightarrow} e_\perp}$

$$\frac{}{v \overset{0}{\hookrightarrow} v}\ (\mathrm{E-Int0}) \qquad \frac{\{e_i \overset{0}{\hookrightarrow} v_i\}}{f\langle e_i\rangle \overset{0}{\hookrightarrow} [\![f]\!]\langle v_i\rangle}\ (\mathrm{E-Op0})$$

Figure 5. Operational Semantics

[9] Oregon Graduate Institute Technical Reports. P.O. Box 91000, Portland, OR 97291-1000, USA. Available online from: ftp://cse.ogi.edu/pub/tech-reports/README.html.

[10] Cherif Salama, Gregory Malecha, Walid Taha, Jim Grundy, and John O'Leary. Static consistency checking for Verilog wire interconnects. Technical report, Rice University and Intel Strategic CAD Labs, http://www.resource-aware.org/twiki/pub/RAP/VPP/FV-TR2.pdf, 2008.

[11] Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. available from (9).

[12] Walid Taha and Patricia Johann. Staged notational definitions. In Krzysztof Czarnecki, Frank Pfenning, and Yannis Smaragdakis, editors, *Generative Programming and Component Engineering (GPCE)*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

[13] PolySpace Technologies. http://www.polyspace.com.

[14] Arnaud Venet and Guillaume P. Brat. Precise and efficient static array bound checking for large embedded C programs. In William Pugh and Craig Chambers, editors, *PLDI*, pages 231–242. ACM, 2004.

[15] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 249–257, Montreal, June 1998.