Formalizing and Enhancing Verilog *

Jennifer Gillenwater, Gregory Malecha, Cherif Salama (presenter), Angela Yun Zhu, Walid Taha
Rice University, Houston, TX, USA

Jim Grundy and John O'Leary
Intel Strategic CAD Labs, Portland, OR, USA

***Abstract*— Hardware description languages (HDL) suffer from inconsistencies between their simulation and synthesis semantics: A program successfully compiled and simulated might fail to synthesize. In this work, we propose the usage of statically typed two-level languages (STTL) to eliminate such inconsistencies.**

Languages like VHDL and Verilog have constructs (loops, parameters, and other abstractions) that helps the digital circuit designer in writing more generic code. These constructs are eliminated during elaboration and replaced by appropriate expansions. In many cases synthesis failure is effectively due to failure to achieve this step. Capturing the mechanics of that preprocessing stage can be formalized using STTLs that allow us to define corresponding expansion semantics. We can therefore statically verify the synthesis feasibility of a certain program by type checking against rules specifically designed for that purpose.

In this paper, we show how this STTL approach can be applied to Verilog. To do so, we define the syntax, first-stage expansion semantics, and type system for a representative subset of the Verilog language that we call (Featherweigth SV). We also prove that according to our model, expansion preserves well-typedness of a program, that it results in an obviously synthesizable program (free from first stage constructs) and finally that it does not depend on wire values.

I. Introduction

All modern hardware description languages allow circuit designers to write generic code capable of describing circuit families. Using such constructs a family of simple encoders can be described in Verilog as:

```
module encode (L,X);
  parameter n = 2;  parameter m = 4;  // 2^n
  input [m-1:0] L;  output [n-1:0] X;
  reg   [n-1:0] X;  integer i;

 always @(L) begin
    X = 0;
    for (i=0; i<m; i=i+1) if (L[i]==1) X=i;
  end
endmodule
```

Unfortunately, circuit designers both in academia and industry usually avoid such descriptions opting for simpler, more explicit descriptions that are free from parameterized modules, iterations, and conditionals. It is common to see the above generic design manually expanded into a concrete instance like the following:

```
if (L[0]==1) X=0; if (L[1]==1) X=1;
if (L[2]==1) X=2; if (L[3]==1) X=3;
```

Clearly this leads to much longer descriptions and more importantly it completely prevents writing generic reusable module descriptions. The above example is a simple example, but our study of available industrial hardware descriptions shows that the problem is pervasive. The OpenRISC 1200's 32x32 multiplier [5] is 2538 lines long, but would be a mere 1405 lines if a well-designed mechanism for preprocessing was available. The situation is similar for the OpenSPARC T1's 64x64 multiplier [3] (1167 vs. 2510 in the original code).

The reason why developers avoid such constructs is that using them produces designs whose properties (including synthesizability) can only be determined after the elaboration phase where these constructs are expanded and therefore eliminated.

The line between the descriptions that are synthesizable, or will elaborate to synthesizable descriptions, and those that are not can be unclear and ad hoc. It doesn't have to be this way. Two-level languages [2], [4] and multi-level languages [6], [7] have been studied as a way to understand code generation in software. They provide a formal infrastructure that allows characteristics of programs to be checked without requiring expansion.

Our thesis is that the techniques developed for statically typed two-level languages are very suitable to hardware description languages. We believe that more systematic support for elaboration combined with more powerful static checking (before elaboration) can reduce the cost needed to produce large scale designs. Our long term goal is to demonstrate this thesis in the context of a practical extension of the Verilog language. Further, we believe that type systems for such languages can be developed to enforce bounds on hardware resources such as area, power and delay.

II. Syntax

In this section we present Featherweight SV, a calculus for a representative core of Verilog that we kept to a minimum to facilitate analysis of the essential features of the language.

The abstract syntax for Featherweight SV is parameterized by sets for identifier and operator names. In addition, it will be convenient to use several meta-variables to range over indices and index domains.

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, ParamaterNames are countably infinite sets, $\mathbb{O}$ is the finite set of operator names,

* An extended version of this internal SRC publication is currently being reviewed for publication. The extended version is available from the authors.

and $\mathbb{N}$ is the set of natural numbers.

The full grammar for the Featherweight SV is defined as follows:

Circuit Description	p	::=	$\langle D_i \rangle^{i \in I}\ m$
Module Definition	D	::=	`module` $m\ b$
Module Body	b	::=	$\langle x_i \rangle^{i \in I}\ \langle d_j\ t_j\ s_j \rangle^{j \in J}$ `is` $\langle t_k\ s_k \rangle^{k \in K}\ \langle P_r \rangle^{r \in R}$
Direction	d	$\subseteq$	{`in`, `out`}
Type	$t \in \mathbb{T}$	::=	`wire` | `reg` | `int`
Parallel Statement	P	::=	$m\ \langle e_i \rangle^{i \in I}\ \langle l_j \rangle^{j \in J}$ | `assign` $l\ e$ | `always` S | `for`$(y = e; e; y = e) \langle P_i \rangle^{i \in I}$
LHS value	l	::=	$s \mid s[e] \mid s[e : e]$
Sequential Statement	S	::=	$@E^{+}\ S \mid l = e$ | `if` e `then` S `else` S | `for`$(y = e; e; y = e) S \mid \langle S_i \rangle^{i \in I}$
Event	E	::=	$g\ l$
Edge	g	::=	`posedge` | `negedge` | `edge`
Expression	e	::=	$l \mid x \mid v \mid f \langle e_i \rangle^{i \in I}$
Value	v	::=	$(0 \mid 1)^{32}$

A circuit description $\langle D_i \rangle^{i \in I}\ m$ 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 and a module body. A module body itself consists of a sequence of module parameter names, a sequence of port declarations (carrying direction, type, and name for each port), a sequence of local variable declarations, and a sequence of parallel statements. A port direction indicates if it is an input, output, or bidirectional port. A bidirectional port can be used either as input or output. The type of a port or a local variable can be `wire`, `register`, or `integer`. A parallel statement can be a module instantiation, an `assign` statement, an `always` statement carrying a sequential statement, or a `for`-loop whose body is strictly composed of parallel statements. A module instantiation provides module parameters as well as what gets connected to various ports. An `assign` statement consists of a left hand side (LHS) value and an expression. An LHS value is either a variable, an array lookup, or an array range. A sequential statement is either a guarded statement, an assignment, a conditional statement, a `for`-loop whose body is a sequential statement, or a sequence of sequential statements. An event consists of an `edge` trigger (`positive`, `negative`, or either) and an LHS value. An expression is either an LHS value, a parameter name, a 32-bit integral value, or an operator application. The choice of 32-bit values reflects the peculiar manner in which Verilog interprets parameter values when they are viewed as wire signals.

Using this simplified and uniform representation, we can write the encoder presented in the introduction as follows:

```
< module encode <n,m> <in wire L, out reg X> is <>
    < always @(L)
       < X=0,
         for(i = 0; i < m; i=i+1 )
           if(L[i] == 1) then X = i else <>
       >
    >,
    module main <> <in wire L, out reg X> is <>
    < encode <2,4> <L,X> >
> main
```

III. Type System

This section presents the type system for Featherweight SV. The type system presented in this section specifies what is a synthesizable description. Almost all constructs are trivially synthesizable because they map directly to physical connections and modules. The only exception is iteration. We show how to type-check `for`-loop constructs and we show what preprocessing computations are implicitly embodied in a design that uses abstraction mechanisms such as `for`-loops and module parameters. In the terminology of two-level languages, preprocessing is the level 0 computation, and the remaining computation is considered to be the level 1 part. In a Verilog description, there are relatively few places where level 0 computations are required. In Featherweight SV, these places are restricted to: 1) expressions that relate to module parameters, 2) expressions that relate to the bounds on a `for`-loop, and 3) indices into arrays.

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 as a computation that has already been pre-processed. Expressions, however, may be computations that either must be performed during expansion or that must remain intact to become part of the result of preprocessing. For this reason, the judgment for expressions will be annotated with a level $n \in \{0, 1\}$ to indicate we are checking this expression for validity at level 0 or at level 1. This annotation will appear in the judgment as a superscript on the turnstyle as shown in $\vdash^n$.

A. Typing Environments

To define the type system we need some auxiliary notions. A module type consists of the number of its module parameters, as well as a sequence of directions and types for ports. An operator signature is a function that takes an operator, the level at which the operation is executed, 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 names of modules 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 considered level 1 variables while parameters and `for`-loop variables (denoted by x or y) are considered level 0 variables.

Module Type	M	::=	$k\ \langle d_i\ t_i \rangle^{i \in I}$
Operator Signatures	Σ	$\in$	$\Pi i.\ \mathbb{O} \times \mathbb{N} \times \mathbb{T}^i \to \mathbb{T}$
Level	n	::=	$0 \mid 1$
Module Environment	Δ	::=	$[] \mid m : M :: \Delta$
Variable Environment	Γ	::=	$[] \mid s : d\ t :: \Gamma \mid x : d\ t :: \Gamma$
Level 1 Variable Environment	Γ^{+}	::=	$[] \mid s : d\ t :: \Gamma^{+}$

B. Typing Rules

A circuit description is well-typed when the judgment $\vdash p$ is derivable. In Figure 1 we only show the most interesting typing rules that require some expressions to be typable at level

0. The first such rule is the module instantiation rule, which requires that the expressions relating to module parameters must be typable as level 0 computations. The rules for `for`-loop (T-For) and (T-SFor) require that the initialization, test, and increment expressions are all typable at level 0. The test and increment expressions require that the environment be extended to include the counter variable as being an integer (with direction $\{\texttt{in}\}$). If any of these expressions is not typable at level 0, the `for`-loop is rejected by the type system.

The rules for expressions (T-Index) and (T-Range) also require that the indices be typable at level 0 as being integers.

IV. Operational Semantics for Prepropcessing

A big-step operational semantics indexed by the level of the computation will be used to formally specify what preprocessing must be done. The specification will dictate how expansion should be performed, what the form of the preprocessed circuit descriptions should be, and what errors can occur during preprocessing. We assume a standard notion of substitution using the usual free and bound variable conventions as in the lambda calculus [1].

To model the possibility of error during preprocessing, we define the following two auxiliary notions:

Term	X	$::=$	$p \mid D \mid b \mid P \mid l \mid S \mid E \mid e$
Possible Term	$X_\perp$	$::=$	$X \mid \texttt{error}$

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

Preprocessing will be 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 will take a sequence of module declarations and an X and produce a new sequence of instantiated modules $\langle D_j \rangle$ and either a preprocessed X or the failure value $\perp$. When the $\langle D_i \rangle$ or $\langle D_j \rangle$ components are irrelevant to the judgment, they will simply be dropped. The first can occur if we are processing an entity that does not require knowledge about the modules available in the context, and the second can occur if we are processing an entity that cannot entail the instantiation of new modules.

The most interesting preprocessing rules are defined in Figure 2. Preprocessing a module instantiation generates a new module representing a unique instance of the module definition. An error occurs if the module is not defined in the context, or if the number of arguments used to instantiate a module does not match the number required by its definition.

Preprocessing a `for`-loop essentially amounts to evaluating a `for`-loop, except that the result of evaluation is a sequence of statements rather than a modification of the global state.

Expressions that are at level 1 are preprocessed, and ones at level 0 are evaluated normally. For expressions at level 0 the only active rule in evaluation pertains to operator applications. However, it is important that the semantics is explicit about the kinds of errors that can occur during evaluation, and in particular that encountering any identifier during level 0 evaluation constitutes a preprocessing error. This formalizes the property that any dependency on either an uninstantiated parameter or a wire value constitutes a preprocessing error.

V. Technical Results

In this section we state and report the validity of the results that formalize the desired properties of Featherweight SV. In particular, we establish three theorems (Theorems 1, 2, and 3).

A. Preprocessing Produces Well-Typed Circuit Descriptions

From the operational semantics, we see that the topmost constructs where substitution can happen are parallel statements and thus we can state the substitution lemma as follows:

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

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

The type preservation theorem can be defined for a circuit description as follows:

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. ■

While this is an important property, it still allows for two undesirable behaviors that our type system does in fact guarantee: First, it is possible for preprocessing to produce the value $\perp$. Second, it is possible for preprocessing to produce a value p' that is not $\perp$, but still contains constructs that we would like to be eliminated during synthesis. The next two results address these two issues.

B. Preprocessing does not Depend on Wire Values (and is Type Safe)

Preprocessing returns the error value if a traditional runtime type errors while a term is being evaluated during preprocessing. But the most interesting cause for such errors in our setting is when a preprocessing computation depends on a wire value. This cannot occur for a well-typed term.

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

Sketch: This results follows directly from Theorem 1 given that no typing rules will consider `error` to be well-typed. ■

C. Preprocessing Produces Fully Expanded Terms

To show that preprocessing produces fully expanded terms that do not contain any unprocessed preprocessing directives, we must formalize this property. The set of fully expanded terms is identical to the set of terms before expansion except that module bodies and instantiations can not have parameters and that `for`-loop statements are not allowed. We call this property preprocessing soundness that is established in Theorem 3 .

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

$$\frac{\{\Gamma \vdash^0 e_i : \{\texttt{in}\}\ \texttt{int}\} \quad \Delta(m) = |\langle e_i\rangle|\langle d_j t_j\rangle \quad \{\Gamma \vdash^1 l_j : d'_j t_j\} \quad \{d_j \subseteq d'_j\}}{\Delta;\Gamma \vdash m\langle e_i\rangle\ \langle l_j\rangle}\ (\mathrm{T-ModInst})$$

$$\frac{\{\Delta;\Gamma, y : \{\texttt{in}\}\ \texttt{int} \vdash P_i\} \quad \Gamma, y : \{\texttt{in}\}\ \texttt{int} \vdash^0 e_2, e_3 : \{\texttt{in}\}\ \texttt{int} \quad \Gamma \vdash^0 e_1 : \{\texttt{in}\}\ \texttt{int}}{\Delta;\Gamma \vdash \texttt{for}(y = e_1; e_2; y = e_3)\langle P_i\rangle}\ (\mathrm{T-For})$$

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

$$\frac{\Gamma(s) = d\ t \quad \Gamma \vdash^0 e : \{\texttt{in}\}\ \texttt{int}}{\Gamma \vdash^1 s[e] : d\ t}\ (\mathrm{T-Index}) \qquad \frac{\Gamma(s) = d\ t \quad \Gamma \vdash^0 e_1, e_2 : \{\texttt{in}\}\ \texttt{int}}{\Gamma \vdash^1 s[e_1 : e_2] : d\ t}\ (\mathrm{T-Range})$$

Fig. 1. Some Typing Rules From The Featherweight SV Type System

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

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

$$\frac{\begin{array}{l}e_1 \overset{0}{\hookrightarrow} v_1 \\ e_2[y \mapsto v_1] \overset{0}{\hookrightarrow} v_2 \qquad v_2 \neq 0^{32} \\ \{\langle D_i\rangle \vdash P_k[y \mapsto v_1] \overset{1}{\hookrightarrow} \langle P_r\rangle^{r\in R(k)}, \langle D_h\rangle^{h\in H(k)}\} \\ \langle D_i\rangle \vdash \texttt{for}(y = e_3[y \mapsto v_1];\ e_2;\ y = e_3)\ \langle P_k\rangle \overset{1}{\hookrightarrow} \langle P_j\rangle, \langle D_q\rangle\end{array}}{\langle D_i\rangle \vdash \texttt{for}(y = e_1;\ e_2;\ y = e_3)\ \langle P_k\rangle \overset{1}{\hookrightarrow} \biguplus_k \langle P_r\rangle^{r\in R(k)} \uplus \langle P_j\rangle, \biguplus_k \langle D_h\rangle^{h\in H(k)} \uplus \langle D_q\rangle}\ (\mathrm{E-ForTrue})$$

$$\frac{\begin{array}{l}e_1 \overset{0}{\hookrightarrow} v \\ e_2[y \mapsto v] \overset{0}{\hookrightarrow} 0^{32}\end{array}}{\langle D_i\rangle \vdash \texttt{for}(y = e_1;\ e_2;\ y = e_3)\ \langle P_k\rangle \overset{1}{\hookrightarrow} \langle\rangle, \langle\rangle}\ (\mathrm{E-ForFalse})$$

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

$$\frac{e \overset{0}{\hookrightarrow} v}{s[e] \overset{1}{\hookrightarrow} s[v]}\ (\mathrm{E-Index1}) \qquad \frac{\begin{array}{l}e_1 \overset{0}{\hookrightarrow} v_1 \\ e_2 \overset{0}{\hookrightarrow} v_2\end{array}}{s[e_1 : e_2] \overset{1}{\hookrightarrow} s[v_1 : v_2]}\ (\mathrm{E-Range1})$$

Fig. 2. Some Expansion Rules From The Featherweight SV Operational Semantics

Fully Expanded Term $\hat{X}$ =
$\{u \mid u \in X_\perp \quad \wedge Y \in subterms(u) \Rightarrow$
$((Y = \langle x_i\rangle^{i\in I}\ \langle d_j\ t_j\ s_j\rangle^{j\in J}\ \texttt{is}\ \langle t_k\ y_k\rangle^{k\in K}\ \langle P_r\rangle^{r\in R} \Rightarrow I = \emptyset)$
$\wedge(Y = m\ \langle e_i\rangle^{i\in I}\ \langle l_j\rangle^{j\in J} \Rightarrow I = \emptyset)$
$\wedge(Y \neq \texttt{for}(y = e; e; y = e)S)$
$\wedge(Y \neq \texttt{for}(y = e; e; y = e)\langle P_i\rangle^{i\in I}))\}$

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. ∎

VI. Conclusions and Future Work

In this paper, we presented Featherweight SV, a core calculus (syntax, type system, and preprocessing rules) that shows how such preprocessing constructs can be developed in the context of a revision of a mainstream hardware description language (Verilog). We also formalized three technical properties that capture the key features of this calculus. More generally, we have shown the usefulness of STTLs when applied to hardware description languages. Using STTLs, we can statically check the synthesizability of a description having as much abstraction constructs as required. If it type-checks, the same description can be expanded into an trivially synthesizable circuit. These results imply that abstraction constructs can safely be used by designers and that they are actually desirable.

In future work, we expect that this framework will play a key role in providing other guarantees about the results of synthesis, including matching bus sizes, area, timing, and power.

References

[1] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, 1984.

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

[3] Sun Microsystems. Opensparc t1 processor file: mul64.v. http://opensparc-t1.sunsource.net/nonav/source/verilog/html/mul64.v.

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

[5] Opencores.org. Or1200's 32x32 multiply for asic. http://www.opencores.org/cvsweb.shtml/or1k/or1200/rtl/verilog/or1200_amultp2_32x32.v.

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

[7] 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.