



Reusing Verilog Designs in the Synchronous Language Esterel

Menachem Leuchter¹, Shmuel Tyszberowicz², and Yishai A. Feldman³

¹ The Open University, Israel

² Tel-Aviv University

³ IBM Haifa Research Lab

Abstract. Veriest is an automatic translator that converts synthesizable Verilog designs into the synchronous language Esterel. The translation into a synchronous language can expose hidden flaws in the original design, including subtle race conditions. In addition, the extensive libraries of verified Verilog designs can now be reused in synchronous designs.

Verilog and Esterel have different models and features, complicating the translation. For example, Verilog has flexible data types and operators for dealing with data buses of varying widths; it also supports three-state logic, which has no equivalent in languages not meant to describe hardware. Veriest creates functions in the hosting language (usually C) to represent concisely such features of Verilog that are not native to Esterel.

1 Introduction

Synchronous languages specify the reactions of reactive and real-time systems to their environment. They have rigorous mathematical semantics, which allows programmers to develop critical software faster and more reliably. Synchronous languages also enable validation and verification of the developed systems. They are particularly useful for designing the control of real-time embedded systems [8].

Modern hardware design is done using hardware description languages (HDLs); Verilog [6] is one of the two most popular such languages (the other being VHDL). As a result, there are many libraries of tested and verified Verilog designs available. These include many useful applications, such as communication protocols, video and audio compression algorithms, cryptographic algorithms, and more. These libraries far outnumber corresponding libraries for synchronous languages. Many of these asynchronous designs could also be useful for designers of reactive systems that use synchronous languages, if they could be translated into these languages. These could be used to synthesize synchronous designs in software as well as hardware.

Besides allowing reuse of existing designs, translating a Verilog design into a synchronous language can expose hidden flaws in the design that were not discovered by testing. Phenomena such as race conditions become obvious when stated in a synchronous language.

Such translation is complicated by the difference in models between Verilog and synchronous languages. Verilog is actually composed of two main sub-languages. *Synthesizable Verilog* is the subset of Verilog that can be directly compiled into hardware.

The rest of the language is used for designing stimulus environments (which are not a part of the design) for the simulation of synthesizable Verilog designs, and contains such features as the generation of random test vectors and assertions. Reusable Verilog designs are all written in synthesizable Verilog. This is a hardware-oriented language, and includes variable-length data types and flexible operators for combining and selecting parts of such buses. Verilog also supports three-state logic, in which a wire can be in an undriven (or “floating”) state, enabling multiple sources of control for the wire. Such features of the language have no equivalent in the target synchronous languages.

We have implemented an automatic translator, called Veriest, that converts synthesizable Verilog designs into the synchronous language Esterel [4]. Esterel is one of the most popular synchronous programming languages, designed for programming reactive systems. Esterel can be compiled into sequential code as well as into hardware. Veriest preserves the Verilog RTL hardware semantics in the generated Esterel code.

In order to create readable Esterel programs, Veriest attempts to keep the structure of the original program as much as possible. Features of Verilog that are not directly expressible in Esterel are implemented using Esterel’s support for external functions in the hosting language (usually C). Veriest defines a set of auxiliary functions, derived from the particular features and operations used in the source design. The function names follow a fixed scheme, so that their semantics can be immediately understood without reference to their implementation. In this way, implementation details are hidden while the design is still executable.

We have successfully translated a number of different Verilog designs using this tool. The generated Esterel code was about half again as long as the original code (excluding the generated C functions). The original intent was quite clear in the generated code, and it seemed to be understandable and maintainable.

1.1 The Gap

Esterel is a high-level synchronous language that supports abstraction, a variety of data types and the ability to define new ones, and interoperability with other languages. It contains special constructs for timing control, and is therefore appropriate as the target language for developing reactive systems.

In contrast, Verilog is meant for hardware design, and the synthesizable subset of Verilog lacks many of the abstraction mechanisms and data types of Esterel. It does emphasize timing, although in an asynchronous setting, and in that sense is closer to Esterel than most conventional programming languages. The translation of C/C++/Java code into Esterel is not as useful as that resulting from Verilog, since the former programs are sequential, and have no timing restrictions or concurrency. The loss in such languages of the synchronization characteristics in the design makes it less suitable for a synchronous language such as Esterel.

Some features of Verilog can be translated more-or-less directly into Esterel,⁴ although care must be taken to preserve the semantics. Other features pose more difficulties. Verilog supports arbitrary-width variables, with their associated operations. For

⁴ In most of this paper we refer to Esterel version 5, which was the latest version when this work was implemented [5]. We address the changes to Esterel in Section 3.1.

example, it is possible to define 5-bit variables, and arithmetical and shift operations on such variables truncate any results to 5 bits. Similarly, it is possible to define a variable consisting of 128 bits, which is beyond the capabilities of Esterel's built-in data types.

Verilog also supports flexible combinations of parts of such variables. For example, the expression

`{data_out[6:0], data_in}` represents the value containing the seven least-significant bits of `data_out` with the value of `data_in` concatenated on the right. In hardware, such operations correspond to simple wiring, without computational content. In Esterel, these have to be represented in code.

Hardware designs sometimes use *tri-state* (or *floating*) signals, which are distinct from logical high and low (or one and zero) values. The floating signals are not actively driven, so that any of a number of connected circuits can drive them in turn. This is useful in cases where one output value can result from one of several sources, each active at a different time, or when a single wire is used alternatively for input and for output. Verilog fully supports tri-state signals; for example, the expression `6'b101zzz` denotes a 6-bit value whose most-significant bits are 101, followed by three floating bits. These bits could be supplied by a different module. Esterel has no built-in support for tri-state values.

In addition, Esterel has no built-in support for arrays, a basic data type of Verilog (and many other languages). These have to be simulated in the generated code.

2 The Translation

Those constructs of Verilog that have direct counterparts in Esterel are translated separately and independently. Note that even seemingly-trivial operations such as addition are complicated by the presence of data types with flexible widths. For example, the Verilog expression `a+b` cannot be directly translated to Esterel if the original `a` and `b` are 5-bit variables, whereas their counterparts in Esterel are integers, since truncation of the sum to 5 bits needs to be added.

Unique Verilog constructs require a more detailed analysis of the Verilog source code. There are general translation solutions for each such construct, but these are sometimes overly complex. Veriest therefore attempts to recognize some frequently-used Verilog patterns, in order to translate those into more concise and efficient code.

Pure Esterel can express some Verilog features awkwardly, and others not at all. For example, it is impossible to represent in Esterel a 128-bit vector using the built-in types. Five-bit vectors *can* be represented using Esterel integers, but correctly translating operations on them to pure Esterel requires additional operations that increase code size and obscure its meaning. To achieve a full and efficient translation we use Esterel *user types* [1]. These types are considered completely abstract by Esterel itself; their implementation is given in the host language. Veriest defines a set of user types with associated operations. The types and operations generated depend on the particular source program; however, they follow common patterns, and their names completely describe their semantics. Designers who want to maintain and modify the generated Esterel code therefore do not need to read the C implementation at all.

```

module up_down_counter(data_out, carry,rst,clk,dir);
input clk,rst,dir;
output [23:0] data_out; output carry;
reg [23:0] data_out; reg carry;
always @(posedge clk or negedge rst)
begin
  if (!rst) begin data_out <= 0; carry <= 0; end
  else begin
    if (dir)
    begin
      data_out <= data_out + 1;
      if (data_out == 24'hffffff) carry <= 1;
      else carry <= 0;
    end
    else begin
      data_out <= data_out - 1;
      if (data_out == 0) carry <= 1;
      else carry <= 0;
    end
  end
end
end
endmodule

```

Fig. 1. A Verilog design of an up-down counter.

We start with a small yet complete example, then discuss the non-trivial issues involved in the translation one by one. The full details are available in [5].

Figure 1 contains a Verilog design of a simple 24-bit up-down counter. Its inputs are a clock, a counting direction selector line (`dir`) and a reset line (`rst`). Its outputs are a 24-bit bus that stores the current value and a carry for the next stage. The value of `data_out` is incremented or decremented at each rising clock cycle, depending on whether the value of `dir` is one or zero, respectively. Whenever `data_out` overflows or underflows, it is truncated, and `carry` is set to one for one clock cycle.

Figure 2 shows the Esterel code generated by Veriest. Both Verilog wires (transient values without memory, such as `clk` and `dir`) and registers (memory devices, such as `carry`) are translated into Esterel as signals with boolean values. A change in the variable value causes the corresponding signal to be emitted, while the value of the variable is stored in the signal's associated value. A binary value could have been represented by the presence or absence of a signal. However, the use of a signal with a value preserves two important characteristics of Verilog register variables: first, retaining the value after the signal disappears; and second, the ability to refer to the value of the signal at the previous clock tick even while a new value is assigned to it. Because of the need to refer to previous values, all signals are generated with appropriate initializations.

The Verilog design uses *procedural assignments*, which have a triggering condition. In this case, the triggering condition is `always @(posedge clk or negedge rst)`, which means that this section of code is activated on a positive edge of the `clk` signal or a negative edge of `rst`. This is translated into an Esterel loop that waits

```

module up_down_counter:
type VEC_23_0;    % definition of new type for 23-bit register
function CONV_STRING_VEC_23_0(string):VEC_23_0;
function PLUS_VEC_23_0(VEC_23_0, VEC_23_0, integer):VEC_23_0;
function COMP_EQ_VEC_23_0(VEC_23_0, VEC_23_0): boolean;
function MINUS_VEC_23_0(VEC_23_0, VEC_23_0, integer):VEC_23_0;
input clk:=false: boolean, rst:=false: boolean, dir:=false: boolean;
output data_out:VEC_23_0, carry:=false: boolean;
loop    % procedural assignment section
  await [clk or rst];
  if ((?clk) and not pre(?clk)) or (not(?rst) and pre(?rst)) then
    if (not (?rst)) then
      emit data_out(CONV_STRING_VEC_23_0("0"));
      emit carry(false);
    else if ((?dir)) then
      emit data_out(PLUS_VEC_23_0(pre(?data_out),
                                CONV_STRING_VEC_23_0("1")), 24);
      if (COMP_EQ_VEC_23_0(pre(?data_out),
CONV_STRING_VEC_23_0("16777215")))
        then emit carry(true);
        else emit carry(false);
      end if;
    else
      emit data_out(MINUS_VEC_23_0(pre(?data_out),
                                CONV_STRING_VEC_23_0("1")), 24);
      if (COMP_EQ_VEC_23_0(pre(?data_out), CONV_STRING_VEC_23_0("0")))
        then emit carry(true);
        else emit carry(false);
      end if;
    end if;
  end if;
end if;
end loop
end module

```

Fig. 2. The Esterel version of the up-down counter.

for a change in `clk` or `rst`. Because these are signals with values, the meaning of the statement `await [clk or rst]` is to wait for the *existence* of either signal, regardless of their values. A signal exists when a value is emitted for it. The type of change (from false to true, denoting a positive edge, or from true to false, denoting a negative edge) is checked separately, with the following `if` statement. This is necessary because Esterel does not support the notion of positive or negative edges of signals, and is the reason for the need to reference previous values. This treatment of signals allows the Esterel compiler to generate more efficient code, which only checks the conditions when the value is assigned, rather than at each instant.

The body of the Verilog `always` statement uses a number of *nonblocking assignments*, expressed using the `<=` operator. The right-hand sides of all concurrent

(a) Verilog source	(b) Esterel translation
<pre>assign o1 = a b; assign b = a;</pre>	<pre>loop await [a or b]; emit o1(?a or ?b); end loop loop await a; emit b(?a); end loop;</pre>

Fig. 3. Translating continuous assignments.

nonblocking assignments are computed before any value is changed. In the generated Esterel code, the same effect is achieved by using the previous values of the variables on the right-hand side. Thus, the two Verilog nonblocking assignments `a <= b;` `b <= c;` would be translated into `emit a(pre(?b)); emit b(pre(?c));` whereas the (sequential) blocking assignments `a = b;` `b = c;` would be translated into `emit a(?b); emit b(?c);`.

Other than these changes, the structures of the two programs are similar. The most striking change is the treatment of the 24-bit data type used for `data_out`. In Verilog, the definition `reg [23:0] data_out` causes the addition operator in `data_out + 1` to denote 24-bit addition, which truncates its result to the required length. To achieve the same result in Esterel, Veriest defines a user type called `VEC_23_0`. All the required operations on this data type are defined as user functions. This particular program uses addition and subtraction on 24-bit quantities, and compares them for equality. Veriest therefore defines the functions `PLUS_VEC_23_0`, `MINUS_VEC_23_0`, and `COMP_EQ_VEC_23_0` for these operations. The implementation of these functions in C is straightforward, and is generated automatically.

This example also shows the treatment of constants. In this example, the target host language could represent integers of up to 16 bits. The 24-bit vectors of the example are therefore represented as a struct of two integers, and there is no general way to specify constants of this type. The general solution in such cases is to use strings to encapsulate the constants, with conversion functions (like `CONV_STRING_VEC_23_0`) to create the internal representation. Of course, whenever possible, numeric constants are used.

Continuous Assignment In addition to the procedural assignments shown in the example, Verilog also supports *continuous assignment* statements of the form `assign var = expression`. Their semantics is that the variable always has the value of the expression; whenever the value of the expression changes, so does the value of the variable.

In order to translate continuous assignments into Esterel, they must be placed in a loop that waits on any variable involved in the expression to change. If there are multiple continuous assignments, they are translated separately into parallel Esterel loops. Figure 3 shows an example of such translation.

Variable-Width Vectors There are two possible ways of dealing with variable-width vectors. The first is to embed them into the closest built-in integer type large enough to hold them. This makes arithmetical operations relatively straightforward, but selecting

subfields and combining them to create new vectors is more difficult. The second approach is to keep each vector as a set of separate bits. This makes selection, shifting, and concatenation easier, but arithmetic becomes complex. Quite often, the same variable participates in both types of operations. Veriest takes the first approach. Each of the new types created for variable-width vectors is embedded in the next highest integer type, or in a C struct of integers in case the field is too wide. As shown above, special-purpose functions are created as necessary in order to perform arithmetical operations on the new types. Similar functions are created for selection and concatenation.

A set of concatenation functions is created when variable-width vectors are combined. For example,

$$\text{data}[7:0] = \{\text{data}[6:0], a\}$$

could be translated into

```
emit data(CONCAT_INT_BOOL(CONV_INT_INT(?data, 6, 0), 7, ?a, 1)),
```

where the second and fourth arguments specify the number of bits to take from the preceding argument. However, Veriest recognizes this pattern as a shift, and produces the more concise

```
emit data(OR_INT_BOOL(SHIFT_L_INT_INT(?data, 7, 1), ?a)).
```

The second argument to the shift function is the width of the shifted field, and the third is the shift amount.

Such heuristics are not necessary for correct translation, but they improve code readability and reduce its size. Since the goal of the translation is to generate maintainable code in the synchronous language, it is important to include such heuristics for common cases, such as shifts. For the same reason, built-in Esterel operators are used when possible. For example, Veriest will use “=” instead of the function `COMP_EQ_VEC_8_0`.

Arrays Verilog supports arrays of other types, and in particular arrays of registers. For example, the declaration `reg [7:0] mem[0:255]` defines an array containing 256 8-bit vectors. (This is a typical way to define memory.) Esterel, on the other hand, does not have built-in arrays. As in the case of vectors, the solution is to use user types. In this example, Veriest will create the type `ARRAY_REG_7_0_255_0`, with associated operations for reading and writing elements or ranges of arrays.

Tri-State Logic Verilog supports a floating state for bits in addition to zero and one. In Esterel, it must be simulated using an additional bit. For each variable that may have floating bits, Veriest creates another variable that has the same name with a `_Z` suffix. Each bit in the new variable has the value one exactly when the corresponding bit in the original variable is in the floating state. Veriest creates code to manage both variables together in order to represent the Verilog semantics. For example, the Verilog excerpt

```
inout [2:0] data;
data = (select) ? 3'b111 : 3'bzzz;
```

is translated into

```

inputoutput data: integer;
output data_Z: integer;
if (?select) then
    emit data(7); % set data to 3'b111
    emit data_Z(0); % set data to normal mode
else
    emit data_Z(7); % set data to tri-state mode
end if

```

In this case, the variable `data` is shared between all modules that may attempt to write to it, but `data_Z` is duplicated for each module. A module may only write to `data` if it asserts in `data_Z` that those bits it is writing to are not floating. Note that when `select` is false and all bits of `data` are set to tri-state mode in the Verilog source, there is no need to change the value of the `data` variable in Esterel, since its value is irrelevant when `data_Z` is set to all ones.

Veriest creates special comparison functions, such as `COMP_EQ_Z` and `COMP_NEQ_Z`, for comparing tri-state variables.

3 Results and Discussion

We have evaluated Veriest on four designs. These are all real and useful Verilog designs, each written by a different programmer. The example designs include the following: the up-down counter shown in Section 2; a Viterbi encoder, an error-correcting algorithm based on convolution encoding [7]; a real-time clock; and an implementation of the I²C protocol for connecting multiple devices using only two wires. In all cases, the generated code was functionally equivalent⁵ to the original. This section analyzes the results of the translation of these examples, and discusses their implications.

3.1 Maintainability

Esterel code generated by Veriest tends to be longer than the original Verilog code; the average increase in our examples was 60% (excluding the generated C functions). Generated lines also tend to be longer, due to the substitution of primitive operators such as arithmetic, selection, shifting, and concatenation, by generated functions with relatively long names. Nevertheless, most lines have a reasonable length.

There are several other factors for the increase in the total size of the code. The largest overhead in the translation comes from continuous assignments, each of which needs to be translated into a loop. This is mostly due to the need to separate the triggering condition into two parts, one in the `await` statement listing the signals to watch, and the other in the `if` statement that checks the particular triggering values. Another source for the increase in size of the generated code is the difference between the control structures in the two languages. For example, Esterel lacks a `case` or `switch` statement; Verilog's `case` statement is therefore translated into nested `ifs`. A very common

⁵ The transformations were designed to preserve the Verilog RTL hardware semantics. We also ran simulations of the source and target programs to check their equivalence.

construct in hardware designs is a state machine, which is often implemented as a long `case` statement. Translating that statement into an `if` results in many terminating `end if` statements, which increase code size and decrease readability.

In spite of all these factors, the increase in the total size of the code seems quite reasonable. The structure of the original design is preserved in the translated design to the best degree possible, including all module, register, and wire names. The original intent is therefore easily discernible in the generated design, making it easily maintainable.

Since the publication of the first author's thesis [5] (on which this paper is based), the Esterel language (currently at version 7) has been enhanced with new data types and operators. These include bitvectors (arrays of booleans), bitvector maps (records composed of fields and bitvector slices), encoders from unsigned expressions to bitvectors, and signal arrays. These were added to the language to support "hardware or low-level software designs" [3]. These changes remove the need to use auxiliary C functions to implement operations previously unavailable. The translation can now be even more concise and readable.

3.2 Discovering Race Conditions

Because of the asynchronous nature of the language, it is possible to write in Verilog designs that have race conditions. These are not flagged by the compiler, and may not be caught during simulation. Figure 4(a) shows a simple example of a race condition [2]. In this example, a `rst` signal sets `x` to 0 and `y` to 1. A subsequent `clk` signal causes a race condition, where the first `always` block tries to set `x` to the value of `y` while the second block simultaneously tries to set `y` to the value of `x`. The Verilog simulator will arbitrarily execute one block first. The result will be that both `x` and `y` will have the same value: 1 if the first block is executed first, and 0 if the second block executes first.

The Veriest translation of this example to Esterel is shown in Figure 4(b). Because of the synchronous semantics of Esterel, this program is illegal and the compiler rejects it with the error message "statically cyclic program, cannot be compiled by scssc."

In this simple Verilog example, the race condition is easy to spot by inspection. However, subtle race conditions in larger Verilog designs will be harder to discover. Simulation, and even hardware execution (for different reasons), may yield correct results even in the presence of race conditions, and the problem might manifest itself only under rare conditions that are difficult to duplicate. The translation to the synchronous language discovers these problems during compilation.

4 Conclusions

We have presented Veriest, a translator that can convert designs in synthesizable Verilog into the synchronous language Esterel. This makes the large body of intellectual-property hardware designs available to be incorporated into synchronous designs. As an added benefit, the translation of an asynchronous design into a synchronous language can help discover subtle timing conditions that may have escaped testing in an asynchronous setting.

```

module race(x, y, clk, rst);
    output x, y; input clk, rst; reg x, y;
    always @(posedge clk or posedge rst)
        if (rst) x = 0; else x = y;
    always @(posedge clk or posedge rst)
        if (rst) y = 1; else y = x;
endmodule

```

(a) A Verilog design with a race condition [2].

```

module race:
input clk:=false:boolean, rst:=false:boolean;
output x:=false:boolean, y:=false:boolean;
loop
    await [clk or rst];
    if (((?clk) and not pre(?clk)) or ((?rst) and not pre(?rst))) then
        if (?rst) then emit x(false); else emit x(?y); end if;
    end if
end loop
||
loop
    await [clk or rst];
    if (((?clk) and not pre(?clk)) or ((?rst) and not pre(?rst))) then
        if (?rst) then emit y(true); else emit y(?x); end if;
    end if
end loop
end module

```

(b) The translation to Esterel.

Fig. 4. A race condition.

The translation is complicated by several hardware-related features of the source language. These are solved by using user types implemented in the host language. The structure of the resulting translation is very close to that of the original, although it is somewhat more verbose. Experiments have shown that the results are still readable and maintainable in the target language.

Some Verilog designs are more appropriate than others for automatic translation to a synchronous language. Designs of large generic processing components such as CPUs and DSPs will probably not generate useful synchronous designs. The same effects can be better achieved by coding directly in Esterel. On the other hand, hardware-oriented protocols for specific purposes, such as communication protocols and compression algorithms, are good candidates for automatic conversion and are expected to yield results comparable to our example cases.

References

1. G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
2. C. E. Cummings. Nonblocking assignments in verilog synthesis, coding styles that kill! In *Synopsis Users Group (SNUG)*, 2000.
3. Esterel Technologies. The Esterel v7 reference manual. <http://www.esterel-technologies.com/files/Esterel-Language-v7-Ref-Man.pdf>.
4. N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer, 1993.

5. M. Leuchter. Translating Verilog designs into the synchronous language Esterel. Master's thesis, The Open University, Israel, February 2003.
6. S. Palnitkar. *A Guide to Digital Design and Synthesis*. Prentice Hall, 1996.
7. J. G. Proakis. *Digital Communications*. McGraw-Hill, 3rd edition, 1995.
8. R. Shyamasundar and J. Aghav. Realizing real-time systems from synchronous language specifications. Real Time Systems Symposium, Work in Progress Session, Orlando, Florida, USA, 2000.