Applying SPMD Verification Techniques to Hardware Description Languages

Pieter Bos
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
p.h.bos@student.utwente.nl

ABSTRACT

This research aims to find ways of applying existing verification techniques of multi-threaded programs to the domain of hardware description languages (HDL's), which can be used to describe electronic logic circuits. The need for this type of verification is clear: we need to be able to reason about and verify processors formally. We create a logic to reason about HDL's by adapting existing verification techniques for concurrent programs that enable us to prove the functional properties of HDL programs, in particular the verification of shared variables and parameterized code blocks.

Keywords

VHDL, formal verification, separation logic, SPMD

1. INTRODUCTION

Hardware description languages are languages that describe the structure of logic circuits and have been around since around the 1960's [2]. An example for which these languages are used is to specify processors [1]. Processors exist in many critical applications where failure is intolerable, for example in integrated circuits in industrial machinery. Verification of these processors often relies on simulation of the logic circuits, since formal mathematical verification using theorem provers quickly becomes too slow because of state space explosion.

The aim of this research is to create a logic for the verification of HDL programs. This is done by leveraging existing verification techniques for single program multiple data (SPMD) programs, such as those for parallel loops and general purpose GPU (GPGPU) programs. The language that is analyzed in this paper is VHDL, though the results should be easily adaptable to different HDL's as well. Concepts in VHDL will be explored and verification techniques for SPMD programs are adapted to work for HDL programs as well. The logic focuses on proving functional properties, but memory safety properties will be explored as well, insofar as they are applicable to HDL's. Applicable here means that memory in HDL's can be a functional property, since the memory cells themselves are defined in the HDL, but can also be implicitly defined.

25^{th} *Twente Student Conference on IT* July 1^{st}, 2016, Enschede, The Netherlands.

1.1 The Research

The research will at first focus on determining the similarities and differences between the SPMD programming model and HDL programs. The next step is to determine which of the verification techniques in the SPMD model carry over to HDL's and consequentially which parts of HDL's can not be reasoned about using these methods. Hence the research questions are:

1. What are the similarities and differences between the SPMD programming model and hardware description languages?
2. How can we adapt SPMD verifcation techniques to reason about HDL programs?
3. Which parts of HDL's can not be reasoned about using adapted verification techniques for SPMD programs?

1.2 Background

In SPMD identical programs run in parallel on different data, while these programs are allowed to be at different points at any time. These programs can however have overlapping data accesses (i.e. read or write to the same location), in which case the result of one program can depend on the moments that statements are executed in the other programs. Most verification techniques rely on annotating the program in some way to specify data accesses or behaviour to reason about the code formally.

One verification technique that will be investigated is the verification of paralellised loops. Blom et al. specified a way to annotate loop iterations [3] with fractional permission based separation logic [5]. These annotations allow for a soundness proof "that loops respecting specific patterns of iteration contracts can be either parallelised or vectorised".

Another verification technique is the verification of GPGPU (General Purpose GPU) programs by Blom et al. [4] This technique can prove "that a kernel does not have data races, and that it respects its functional behaviour specification." The annotations consists of behavioural specifications and a modified version of fractional permissions where the allowed values are 'read-write' and 'read-only'.

The investigated language, VHDL, defines circuits by defining entities, small pieces of hardware analogous to subroutines, and composing them to create complex logic. Furthermore, VHDL distinguishes between asynchronous logic by specifying direct (electrical) connections between points and sequential logic that triggers on changes in variables. VHDL documentation also often talks about synthesizable versus unsynthesizable code, which means that it can be compiled to a real target instead of run in a

Example 1. VHDL signal assignment

```
a <= b;
c <= d;
```

Example 2. Permission based separation logic for shared variables

```
architecture default of ent is
shared variable v: integer;
begin
    -- requires rising_edge(clk) ⇒ Perm(v, 1)
    -- ensures rising_edge(clk) ⇒ Perm(v, 1)
    process(clk)
    begin
        if rising_edge(clk) then
            v := v + 1;
        end if;
    end process;

    -- requires ¬rising_edge(clk) ⇒ Perm(v, 1)
    -- ensures ¬rising_edge(clk) ⇒ Perm(v, 1)
    process(clk)
    begin
        if not rising_edge(clk) then
            v := v + 1;
        end if;
    end process;
end;
```

Example 3. Transformed shared variables

```
architecture default of ent is
signal v_s: integer;
begin
    process(clk)
    variable v: integer;
    begin
        v := v_s;

        if rising_edge(clk) then
            v := v + 1;
        end if;

        if rising_edge(clk) then
            v_s <= v;
        end if;
    end process;

    process(clk)
    variable v: integer;
    begin
        v := v_s;

        if not rising_edge(clk) then
            v := v + 1;
        end if;

        if not rising_edge(clk) then
            v_s <= v;
        end if;
    end process;
end;
```

simulator, such as an FPGA or an ASIC design. The synthesizability of VHDL code depends entirely on the chosen target and available synthesizers.

1.3 Method

The first stage of this research consisted of diving into VHDL and determining the exact semantics of VHDL code. In this part of the research the specification for VHDL [7] was used extensively. Nearly all constructs described in the document were investigated. A summary of the comparison to the SPMD model can be found in section 3.1. From this comparison several interesting avenues of research were found. One of them is the concept of shared variables in VHDL and includes logic for them in section 3.2. Another interesting concept is the concept of parameterized VHDL, which enables the user to write generic code. This is described in section 3.3. The paper also describes future research that might be useful: synthesizability of VHDL code in section 4.2 and the stability of circuits generated by signal assignments 4.1.

2. RELATED WORK

The most common approach for the verification of HDL programs is the non-formal method of creating testbenches. This involves specfying stimuli to the model or generated circuit and observing wether the result is the same as the behaviour that was specified. The stimuli can be predefined or randomized [9].

Another approach involves modelling the circuit implementation as a state machine, sometimes represented in alternative forms such as a binary decision diagram [8, 6]. States are then enumerated to check functional behaviour formally, usually pruning the set of states that are required to be checked in some way.

3. RESULTS

3.1 Similarities and Differences

VHDL programs consist of a series of entity definitions. Each entity can define a port with in- and output signals, which it can use to communicate with other entities. An architecture implements an entity and consists of signal assignments, process statements and instantiations of other entities (components).

Signal assignments happen continuously, forever and in parallel. A simple set of assignments such as in Example 1 is similar in behaviour to a parallel program such as:

(**while** $true$ **do** $a := b$ **done** $\parallel$ **while** $true$ **do** $c := d$ **done**)

Processes are blocks of sequential code that react to changes in signals. Processes can also use variables, which are different from signals in VHDL. In Example 2 the process blocks react to changes in the *clk* signal. Executions of process blocks are governed by simulation cycles. At the start of each simulation cycle the processes that have been triggered are executed concurrently. Process blocks may also assign to signals, but the value read from the signal stays the same within the simulation cycle. Only after all processes that executed in the simulation cycle have finished, the last value assigned to the signal will be written to it.

Data races do not exist for variables or signals within process blocks, since variables are local to a single process, and signals can only be assigned to from a single process, unless the type of the signal explicitely defines how to deal with multiple assignments.

Finally, instantiations of other entities can be treated as though the architecture definition of the entitity was within the architecture where the component was instantiated. This means that a VHDL program can eventually be reduced to a collection of process blocks and signal assignments.

3.2 Shared Variables

Shared variables are variables that are defined for an architecture instead of a process in an architecture. All processes in the architecture can assign to and read from the shared variable. Behaviour is undefined when two processes access the same shared variable in the same simulation cycle, much like programs in the SPMD model. Furthermore, shared variabels are usually not synthesizable.

Permission based separation logic can be used to reason about shared variables, where each process in the architecture requires some permission on the shared variable, as in Example 2. In this example both processes react on changes to a signal called *clk*. The first process writes to a shared variable on a rising edge of the clock, and hence only needs a write permission on the rising edge of the clock, so the specification is $rising\_edge(clk) \Rightarrow Perm(v, 1)$. The other process only writes to the shared variable on the falling edge, so the specification becomes $\neg rising\_edge(clk) \Rightarrow Perm(v, 1)$. The separating conjunction of the two processes requires a full write permission on the shared variable. When the separating conjunction of the preconditions of all the processes in an architecture holds, we can transform the architecture to a synthesizable and functionally-equivalent architecture. This can be done by doing the following transformations:

1. Remove the shared variable from the architecture;
2. Add process-local variables with the same name to each of the processes in the architecture;
3. Add a uniquely-named signal to the architecture to communicate updated values of the emulated shared variable across simulation cycles;
4. For each process assign the signal value to the local variable before all other statements;
5. For each process assign the local variable to the signal after all other statements whenever it holds a write permission.

A converted example of Example 2 can be found in Example 3. This approach works because the permission based separation logic proves that every simulation cycle either (i) there are only processes that read from the shared variable or (ii) there is one process that reads and/or writes to the shared variable. We use a signal to communicate the updated value of the signal to the next simulation cycle. In the first case, we can simply use the value communicated by the signal, because the value does not change in the current simulation cycle. In the second case, we can use a local variable instead of the shared variable, because there is just one process that can read from and write to the shared variable. The value from the local variable is then written to the signal to communicate the value to the next simulation cycle.

Example 4. Component that rotates a bit string a given amount

```
entity rotate is
generic(
    N: integer
);
port(
    i: in std_logic_vector(N-1 downto 0);
    shift: in integer range 0 to N-1;
    o: out std_logic_vector(N-1 downto 0)
);
end rotate;

architecture rotate_arch of rotate is
begin
    output: for x in 0 to N-1 generate
        -- requires Perm(o(x), 1) ** Perm(i((x + shift) mod N), 1/2)
        -- ensures Perm(o(x), 1) ** Perm(i((x + shift) mod N), 1/2)
        o(x) <= i((x + shift) mod N);
    end generate;
end rotate_arch;
```

Example 5. Component that adds two numbers of arbitrary size

```
entity adder is
generic(
    N: integer
);
port(
    x: in std_logic_vector(N-1 downto 0);
    y: in std_logic_vector(N-1 downto 0);
    o: out std_logic_vector(N downto 0)
);
end adder;

architecture adder of adder is
signal carry: std_logic_vector(N-1 downto 0);
begin
    adders: for i in 0 to N-1 generate
        -- requires Perm(x(i), 1/2) ** Perm(y(i), 1/2) ** Perm(carry(i), 1)
        -- ensures Perm(x(i), 1/2) ** Perm(y(i), 1/2) ** Perm(carry(i), 1/2)
        -- ensures i > 0 ⇒ Perm(carry(i - 1), 1/2)
        -- ensures i = N - 1 ⇒ Perm(carry(i), 1/2)

        -- L1: if (i > 0) recv Perm(carry(i - 1), 1/2) from L2,1
        first_bit: if (i = 0) generate
            adder: full_adder_ent port map(
                c => '0',
                x => x(i),
                y => y(i),
                b0 => o(i),
                b1 => carry(i));
        end generate;

        other_bit: if (i > 0) generate
            adder: full_adder_ent port map(
                c => carry(i - 1),
                x => x(i),
                y => y(i),
                b0 => o(i),
                b1 => carry(i));
        end generate;
        -- L2: if (i < N - 1) send Perm(carry(i), 1/2) to L1,1
    end generate;

    o(N) <= carry(N-1);
end adder;
```

Example 6. Definition of a resolved type

```
type bit is ('0', '1');
type bit_vector is array (natural range <>) of bit;

function resolved(s: bit_vector) return bit;
    variable result: bit := '0';
begin
    for i in s'range loop
        result := result or s(i);
    end loop;

    return result;
end resolved;

subtype or_bit is resolved bit;
```

Example 7. Specified signal assignments with multiple assignments to a signal

```
architecture default of ent is
-- ghost int x_0
-- ghost int x_1
-- requires Perm(x_0, 1) ** Perm(x_1, 1)
-- ensures Perm(x_0, 1) ** Perm(x_1, 1) ** x == resolved(x_0, x_1)
begin
    -- requires Perm(x_0, 1)
    -- ensures Perm(x_0, 1)
    x <= a; -- set x_0 := a

    -- requires Perm(x_1, 1)
    -- ensures Perm(x_1, 1)
    x <= b; -- set x_1 := a
end ent;
```

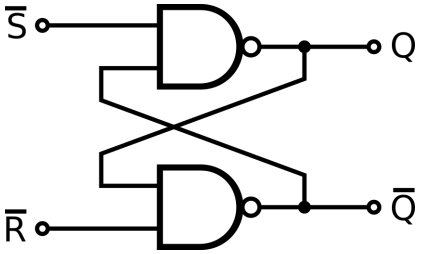

Figure 1. Flip flop with two NAND gates

3.3 Parameterized VHDL

Another feature of VHDL that was investigated is the support for parameterized entities. Parameterized entities in VHDL are indicated by a generic block in the entity. This block contains the definitions for the parameters that are given to each instantiation of the entity. The architecture can then use these parameters to alter behaviour in the entity and generate different numbers of components. The supported constructs for this are *for* and *if* and are indicated by a *generate* keyword. Example 4 and Example 5 include these constructs.

These *generate* constructs could be annotated with permission based separation logic as though each signal assignment is a regular assignment, reasoning about the *for generate* loop as though it is a paralellizable loop. This is a technique that is used for the verification of SPMD programs as well [3]. When the separating conjunction of the permissions holds, we can conclude that there are no cyclic dependencies between the signals and the signals will be stable eventually, as there must be an equivalent sequential ordering of the signal assignments.

In Example 4 the convergence of all the signals is quite obvious, as there are no dependencies between the iterations of the loop. However, there are dependencies between the iterations in Example 5. These dependencies are specified by the send and receive statement. Interestingly it also matters in VHDL whether the dependencies are forward or backward loop-carried, as they hint at the time it takes for signals to stabilize. Example 5 is a ripple carry adder, where each full adder needs to wait for all bits of lower significance to determine the carry. This is modeled by the backward loop-carried dependence, as the signal value will only stabilize after the result of the previous iteration is stabilized. If the statements in the loop body can be ordered such that there are only forward loop-carried dependencies, the values of the signals do not need to propagate (ripple) through the iterations.

However, regular permission based separation logic is too strict to verify all stable series of signal assignments. For example, signals are allowed to have multiple values assigned to them for some types that define a *resolution function*. A resolution function is a function in VHDL that takes an array of a type and returns a result of the type. [7, pp.27]. Example 6 is an example of such a resolved type. In this example the type *or_bit* is a subtype of *bit* and a signal of this type will take the disjunction of all the values supplied to it. These kind of VHDL programs can be reasoned about in the same way described above using ghost variables. One way to do this is to declare a separate ghost variable for each assignment to a signal that has multiple assignments, asserting that the actual signal value is the result of the resolution function over all the ghost variables. A simple example of this method can be found in Example 7. Note that here, as with parameterized VHDL, the signal assignments are reasoned about as concurrent programs. Instead of requiring write permission on the signal for each of the assignments, the write permission is required for the ghost variable.

4. EXTENSIONS

4.1 Stability of Circuits

The previous section showed that permission based separation logic can prove that a series of signal assignments results in a circuit that eventually stabilizes. However, there exist circuits that are stable under certain conditions, but can not be proved in this way. An example would be a memory cell consisting of two NAND gates, as shown in figure 4.1. A NAND flip flop is stable whenever the signal of the two output are the inverse of the other output, or exactly one of set and reset are high.

The challenge here is to prove in which cases the circuit is stable, perhaps specifying the condition as a precondition to the circuit.

4.2 Synthesizability

VHDL contains statements that are not synthesizable by all commercial synthesizers. An example of this is the *shared variable* declaration discussed earlier, which can be compiled down to a regular signal in some cases. Future research could investigate the possibility of translating other unsynthesizable constructs, such as the *wait* statement. The *wait* statement can be used in processes in two ways: by (i) waiting on a change in signal or a condition on a signal, and by (ii) waiting a specified real amount of time. Some usages of this statement might very well be translatable to a piece of VHDL that is synthesizable.

5. CONCLUSIONS

This paper has presented several ways of applying SPMD verification techniques to hardware description languages. VHDL is similar to SPMD programs in processes, but also has asynchronous parts which can be described in the SPMD model. Permission based separation logic can be used to specify the access of shared variables between processes, and VHDL that contains shared variables can be translated to VHDL without shared variables if the proof holds. Permission based separation logic was also used to create a logic to specify signals assignments in asynchronous VHDL in parameterized code. Specified code always generates a stable circuit, because there is a sequential equivalent to the signal assignments. Data dependencies between signals can also be modeled to some extent, although this logic cannot specify all series of signal assignments that are stable. Further research might look into how to prove in which cases a circuit is stable by extending the logic presented. Other unsythesizable constructs might prove to be translatable as well, next to shared variables. All in all the verification of VHDL programs proves to be quite similar to the verification of SPMD programs, though a lot of work remains to be done.

6. REFERENCES

[1] Opencores projects. `http://opencores.org/projects`, 2016. Accessed: 2016-05-15.

[2] M. R. Barbacci. A comparison of register transfer languages for describing computers and digital systems. *IEEE Transactions on Computers*, (2):137–150, 1975.

[3] S. Blom, S. Darabi, and M. Huisman. Verification of loop parallelisations. In *Fundamental Approaches to Software Engineering*, pages 202–217. Springer, 2015.

[4] S. Blom, M. Huisman, and M. Mihelčić. Specification and verification of gpgpu programs. *Science of Computer Programming*, 95:376–388, 2014.

[5] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *ACM SIGPLAN Notices*, volume 40, pages 259–270. ACM, 2005.

[6] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *Design Automation Conference, 1990. Proceedings., 27th ACM/IEEE*, pages 46–51. IEEE, 1990.

[7] IEEE. *IEEE Standard VHDL Language Reference Manual*, 2000.

[8] T. Kam and P. Subrahmanyam. Comparing layouts with hdl models: a formal verification technique. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 14(4):503–509, 1995.

[9] C. Spear. *SystemVerilog for verification: a guide to learning the testbench language features*. Springer Science & Business Media, 2008.