A META MODEL SUPPORTING BOTH HARDWARE AND SMALLTALK-BASED EXECUTION OF FPGA CIRCUITS

LE Xuan Sang
Lab-STICC, ENSTA Bretagne/ Institut Mines-Telecom, Mines Douai
xsang.le@gmail.com

Loïc Lagadec
Lab-STICC, ENSTA Bretagne
loic.lagadec@ensta-bretagne.fr

Luc Fabresse Jannik Laval
Noury Bouraqadi
Institut Mines-Telecom, Mines Douai
luc.fabresse@mines-douai.fr/ jannik.laval@gmail.com/ bouraqadi@gmail.com

Abstract

High level synthesis (HLS) refers to an automated process that creates a digital hardware from an algorithmic description of some computation. From the perspective of Smalltalk, this process consists of converting code from the oriented object level to the register transfer level (RTL), that supports direct compilation to the hardware level. In this paper, we present first steps to achieve this process. We introduce a Smalltalk-based meta-model that allows expressing descriptions (i.e. models) of digital circuits. These descriptions can be materialized as Smalltalk code. A such circuit description can be run on top of the Smalltalk VM, simulating the parallelism intrinsic of hardware. Alternatively, it can be compiled into a binary representation directly transferable to FPGA chips, which can run and exchange data with Smalltalk objects.

Keywords Smalltalk, Pharo, FPGA, VHDL, Meta-model, Dynamic.

1. INTRODUCTION

For the past three decades, high level synthesis (HLS) has represented an effort from the research community to provide "algorithm to gates" capabilities. Its ambition is to bring the advantage of modern software techniques into the hardware design world and therefore let designers efficiently build and verify hardware, which is traditionally a long and tedious process. Synthesis begins with a high-level specification of a problem, where the algorithm is described using a high-level language (Smalltalk in our case). The code is then analysed, architecturally constrained and scheduled to create a register transfer level (RTL) representation that can be implemented on the actual hardware. The HLS process consists of activities [1, 2] listed below:

- *Source code analysis*: from here the code is analysed lexically, the algorithm is optimised and the control/-data flow graph (CDFG) of the algorithm is built.
- *Resource allocation*: this step determines all data elements and operations on the CDFG. Thus we can estimate resources needed (memory, hardware block, etc.) on the FPGA.
- *Scheduling*: the time slot (latency in term of clocks) is assigned to each operation on the CDFG. Since the algorithms are often sequential, the scheduling is needed for synchronising between operations.
- *Resource binding :* operations and data elements on the CDFG are assigned to specific operations (adder, multiplier, etc.) and memory elements (registers, etc.) on the FPGA.
- *Low level representation of the algorithm* : transformation of the algorithm to a RTL representation that can be synthesized to the gate level by the use of a logic synthesis tool.

Most of today's HLS tools rely on strongly typed languages like C/C++ [3–5], Mathlab and Java which can simplify the resource allocation step. However, the static nature of these languages makes them more challenging for debugging generated circuits. For each minor change in source-code, a complete offline recompilation of the whole project is required. Furthermore, the generated low-level representation of the algorithm is often static and therefore, difficult to interact with for verification. Our approach uses the dynamic language Smalltalk instead to build the

International Workshop IWST'15, July 15th 2015, Brescia, Italy.
Copyright © 2015 ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

HLS chain. Thereby, we can benefit from Smalltalk's dynamicity to syntactically analyse the code, its architectural constraints and directly generate the model of the circuit.

The meta-model that we present in this article is our first step toward a HLS chain using Smalltalk. In traditional HLS tools, the circuit can be described using a Hardware description language (HDL) or a static meta-model [3, 6–8] which acts as an intermediate format that can be verified using an external simulator. Since these meta-models are invariant, we cannot dynamically interact with them (insert debug signal, probe signal, etc.). There is also another Smalltalk-based meta-model [9]. This meta-model is an object oriented description of the FPGA circuit and can be simulated using an internal simulator or executed on hardware. However, it allows defining only specifications that are not executable.

Our approach is different, our meta-model allows building directly executable models in which the circuit is treated as a Smalltalk object whose internal states can be changed (input, output, signal value, etc..) and can be inspected. The outside world can manipulate it as a normal object. The meta-model describes not only the circuit structure but also its behaviour. By using the meta-model, the circuit can be transparently verified with or without hardware. Finally, to support synthesis on the actual hardware, our meta-model is capable of exporting circuit descriptions to VHDL descriptions for low level synthesis.

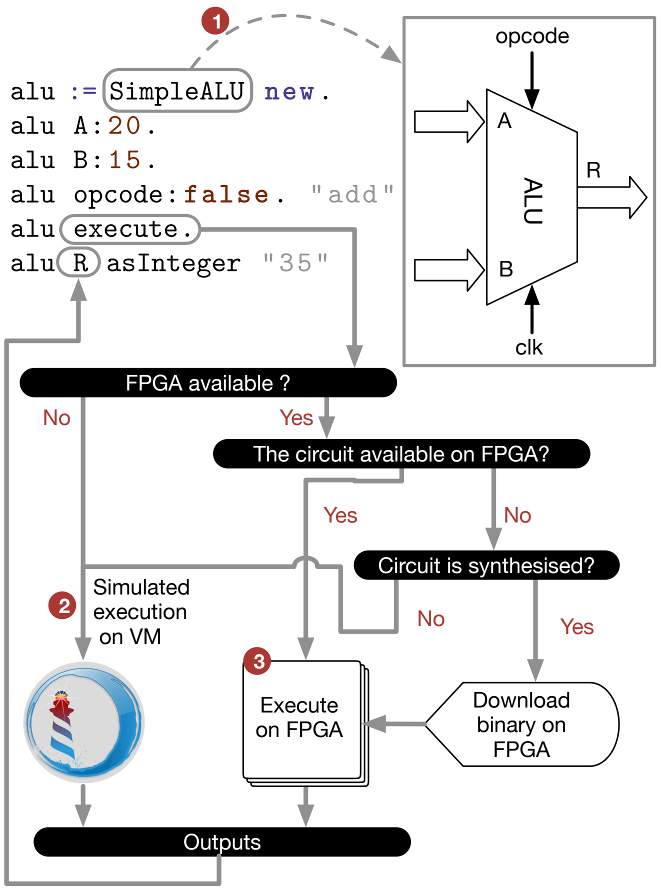
In this paper, we first introduce a simple example that presents our motivation for the meta-model as well as the challenges of modelling digital circuits (2). Section 3 describes architecturally our meta-model and section 4 shows how a FPGA circuit can be simulated in Smalltalk. We then describe how the circuit-model can be transparently executed and debugged on the hardware in section 5. In the last section of the paper (6), we discuss some limitation of the meta-model and depict some future works.

2. PROBLEMS THROUGH AN EXAMPLE

Consider the simple example of an Arithmetic Logic Unit (ALU) as shown in figure 1. This ALU (at the top right of the figure) has a very simple architecture and functionality: it has two data inputs (`A`,`B`) that represent two operands and one data output (`R`) which is the result of an operation. The ALU supports two basic operators (+ and `AND`) which can be chosen via the `opcode` signal (`'0'`/`'1'` for +/`AND` respectively). Each operation is performed during one single clock cycle (`clk`). Assume that we can model this circuit with a class named `SimpleALU`, and use it as shown in the figure's top left code.There are two cases of execution, depending on the execution context, as shown in the bottom of the figure. If the hardware is available, and the circuit is deployed or ready to be deployed, the circuit-model is performed on hardware by sending out inputs and getting back outputs as the circuit finishes its calculation. Otherwise, the model is executed by modelling the circuit in Smalltalk environment. For the last case, the model (`SimpleALU`) must be able to capture both the structure and the behaviour of the circuit.

Problem n°. 1

The first problem when dealing with this modelling approach is to make the circuit acts as a normal Smalltalk object. Apart from the specification of the circuit, the model must be able to maintain the state of the circuit over time. This allows the outside world to interact with the circuit by assigning value to its inputs or reading it outputs. As shown in the example code on figure 1, the state of the ALU is represented by the values of `A`, `B`, `R` and `opcode`. These values must be accessed as in a normal Smalltalk object, regardless of whether the circuit is on hardware mode or it is simulated.



```
alu := SimpleALU new.
alu A:20.
alu B:15.
alu opcode:false. "add"
alu execute.
alu R asInteger "35"
```

Figure 1: In the code (top left), the class `SimpleALU` models structurally and behaviourally the FPGA circuit of the ALU (top right). The schema on bottom shows how the circuit can be performed with or without the hardware.

Problem n^o. 2

The second problem relates to the execution nature of the FPGA circuit when simulated using Smalltalk. For a piece of code in Smalltalk, most VMs perform sequentially, one byte-code at a time. Since an operation frequently depends on the result of an earlier operation, the order of execution cannot be altered at will.

Digital hardware (like FPGA circuits), on the other hand, is very different. A digital system is constructed from a number of smaller components by wiring their inputs and outputs together. When a signal changes, all components connected to the signal are activated and hence operations related to these components are triggered accordingly. These operations are performed in parallel[1], each one takes a specific amount of time to complete which represents its propagation delay. After completion, each component updates the value of its output ports. In case of changes, these output signals, in turn, will activate all components they are connected to, and initiate another round of operations.

Consider the VHDL implementation of the previous ALU example in the listing 1. The `entity` part (lines 6-12) defines the external interface of the circuit: its name, its input/output ports, and each port's data type. The `architecture` part (lines 15-31) describes the internal structure and behaviour of the circuit. From here, we have 4 main parts: each of parts 1 to 3 is defined in a single line (lines 18-20) while the last one gathers lines 21 to 30. All of them operate in parallel and can be placed in any order inside the architecture body without changing the circuit. The first three parts are called combinational circuits which define their structure and how they are connected together. The last one is called a *process*. It describes the behaviour of the circuit that decide which operation is performed depending on the value of the `opcode`. All the statements inside this `process` body are executed sequentially, thus their execution order is important.

All 4 parts are activated only when their input values are changed. For example the assignment to the signal `r1` (line 18) is performed whenever the signal `A` takes a new value. The `process` (lines 21-30) is triggered each time the `clk` signal changes its state. All assignments to signals inside its body will only take effect at the end of the process. All these parts need a propagation delay before their associated signals' values become stable.

This description shows the unique characteristics of digital systems which are hard to be captured by a traditional programming language like Smalltalk. Therefore, these characteristics must be taken into account in a such Smalltalk-based executable meta-model of FPGA circuit (4).

```
1  -- libraries declararion
2  library IEEE;
3  use IEEE.STD_LOGIC_1164.ALL;
4  use IEEE.NUMERIC_STD.ALL ;
5  -- Entity declaration
6  entity SimpleALU is
7  port (
8    clk:in std_logic;
9    A,B:in std_logic_vector(31 downto
        0);
10   opcode:in std_logic;
11   R:out std_logic_vector(31 downto 0)
12 );
13 end SimpleALU;
14 -- Architecture
15 architecture arch of SimpleALU is
16   Signal r1,r2,r3:signed(31 downto 0)
        :=(others=>'0');
17 begin
18   r1<=signed(A);
19   r2<=signed(B);
20   R<=std_logic_vector(r3);
21   process(clk)
22   begin
23     if rising_edge(clk) then
24       case opcode is
25         when '0'    => r3<=(r1+r2);
26         when '1'    => r3<= r1 AND r2
                ;
27         when others => (others=>'0');
28       end case;
29     end if;
30   end process;
31 end architecture;
```

Listing 1: VHDL implementation for the simple ALU circuit

Problem n^o. 3

The third problem concerns the execution of the circuit on hardware. In this case, the circuit-model must take care of all steps necessary to communicate with the actual circuit on the FPGA (5) : (1) Automatically generate hardware and Smalltalk side communication interface. (2) Transparently encapsulate and deploy the circuit on the hardware.

Since the meta-model can capture the nature of digital circuits, it allows debugging circuit-model's behaviour using the traditional Smalltalk debug features like: stop the execution at a given point, process step-by-step or restart the execution, etc. This is however not obvious when the circuit model is executed on hardware. In this case, the cir-

[1] In digital hardware design, the term *concurrently* is preferred in this case. Here we use the term *parallel* to avoid the confusion about the concurrence concept in Smalltalk world

cuit acts as a black box that can be accessible only after execution finishes. Therefore, we are unable to inspect its intermediate states.

To enable the software-like debug control capabilities on hardware, the meta-model must generate and inject some dedicated debug components into the main circuit. These components allow halting the execution every time a debug trigger id raised (i.e. unconditional/conditional break point). The state of the circuit can be accessed during this phase, then the execution can be resumed at will. Evidently, the injected debug modules can be easily removed from the main circuit after the validation.

3. FPGA CIRCUITS MODELLING

To describe FPGA circuits, our meta-model relies strongly on the well known hardware description language VHDL. The language comes with many extensions and can be used for hardware description or simulation purposes. Since our objective is to model FPGA circuits that can be actually executed on hardware, the meta-model is based on a subset of VHDL dedicated for the synthesizable system: the IEEE 1076.66 RTL synthesis standard (VHDL-87)[10]. It can describe directly or indirectly almost every VHDL structures in the standard. Such structures are called meta-descriptions in our meta-model. A meta-description contains structural and behavioural informations about an element of a circuit (data, operation, control or other sub-circuit). Thereby, for each individual element, we know how it is organised, how it connects to other elements and how it is performed (executed) when activated. In addition, a meta-description holds also its equivalent VHDL representation (`#asVHDL` message) to facilitate the conversion. In this way, when all elements are connected together, we have a full structural and behavioural description of a circuit. A simplified class diagram of the meta-model can be found in the appendix of this paper.

3.1 Modelling circuit signals as data objects

The state of a circuit at a particular point in time is represented by values of its ports and of its internal signals. The circuit-model, apart from the specification, contains also the data objects which refer to these signals. A Signal is an object of a data type that represents either wires, or output of a combinational logic, or latches or registers. A Port is a Signal with additional information about its direction. In our meta model, a Port must be either input, output or bidirectional (inout).

The meta-model supports two main data types, the Logic and the LogicVector (in accordance with the `std_logic` and `std_logic_vector` types of the IEEE standard 1164) because they are supported by most synthesis tools. Each one is defined by a set of values that the data object (Signal, Port) can assume, and a set of operations that can be performed on objects of this data type. Based on these two data types, we can easily model other data types like `signed,unsigned,etc.` if needed. In addition, to ease the interaction with the circuit, the meta-model also supports the native Smalltalk Integer data type by providing conversions methods (`#asInteger`, `#asLogicVector:`, etc.).

As mentioned above, in digital circuits, each operation has a propagation delay, and thus the assignment of operation's output to a signal needs also a delay to take effect. That's why we model a Signal as an object with a history. To maintain this time history, a signal holds two informations: (1) the current value of the signal (before the operation) and (2) the new value that will take effect after the propagation delay of the operation. Each time the signal is updated (i.e. once the propagation delay elapses), the new value will become current.

```
HDLSketch subclass: #SimpleALU
  instanceVariableNames: 'r1 r2 r3'

SimpleALU»setUpPorts
  self in:{#A.#B}of:(LogicVector size:32
     ).
  self in:#opcode of:Logic new.
  self out:#R of:(LogicVector size:32)

SimpleALU»setUpSignals
  r1 := Signal of:(LogicVector size:32
     signed:true).
  r2 := Signal of:(LogicVector size:32
     signed:true).
  r3 := Signal of:(LogicVector size:32
     signed:true).
```

Listing 2: Initialise the state of the circuit

As shown in listing 2, the two methods `#setUpPorts` and `#setUpSignals` define the state of the `SimpleALU` circuit. The first method describes the external interface of the circuit by specifying its input and output ports as well as the data type of each port. The second method initialises ALU's internal signals. The two methods are normal Smalltalk methods which are used to initialise data objects for the circuit (modelled by the meta-model), therefore they are compiled by the default Smalltalk compiler.

3.2 Circuit architecture modelling

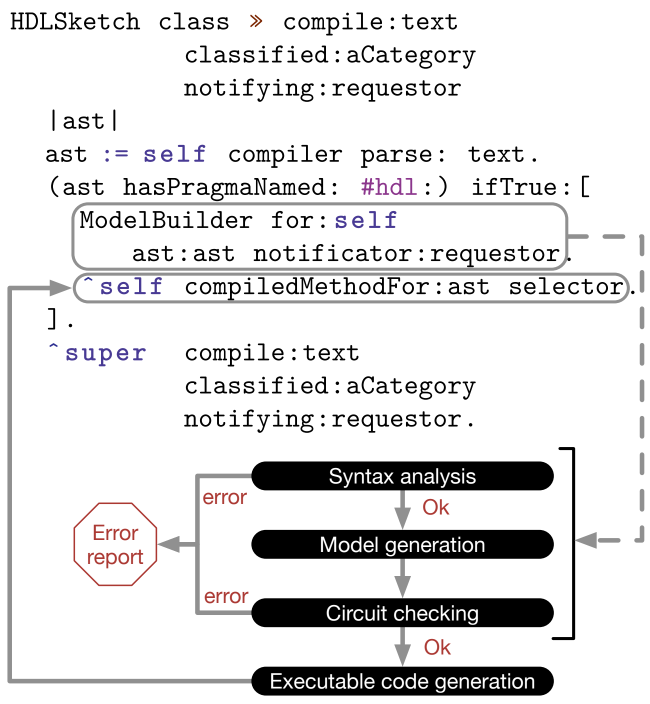
To simplify the circuit specification process, in addition to the meta-model, we introduce a dedicated domain specific language (DSL) based on a subset of Smalltalk. This DSL allows to easily and directly describe the FPGA circuits in Smalltalk.

Listing 3 shows the implementation of the `SimpleALU` circuit using this DSL. The method `#execute` defines the architecture of the circuit. As mentioned in section 2, since we cannot capture the characteristics of digital hardware using the traditional Smalltalk, the DSL is used instead in this case. We apply a convention that all methods with the pragma #hdl: are implemented using the DSL and are handled by a specific compiler.

```
SimpleALU»execute
<hdl:#combinational>
  r1 <- (self A).
  r2 <- (self B).
  self R <- r3.
  {self clk} onChange:[
    self done <- false.
    self clk posedge ifTrue:[
      self opcode caseOf: {
        ['0'] -> [r3 <- (r1 + r2)].
        ['1'] -> [r3 <- (r1 and:r2)].
      }.
      self done <- true.
  ]].
```

Listing 3: Implementation of the SimpleALU using our DSL

Figure 2 shows how the methods with the #hdl: pragma are compiled. We reimplement the class method #compile:classified:notifying: so that the class HDLSketch use a dedicated compiler for the DSL (the ModelBuilder). When a method is identified as a DSL method, the compiler will first analyse it and verify whether the syntax is supported by the DSL. Since the DSL is a subset of Smalltalk, we can benefit from many features of the default Smalltalk editor like syntax highlight, error report system, etc. If the syntax analysis succeeds, the model of the circuit described by the method is generated automatically by compiling each DSL statement to an equivalent meta-description. This model is just the skeleton of the circuit and doesn't have any specific data objects yet. When an instance of the class SimpleALU is created, its data objects are initialised (via the setup methods) and are assigned to this skeleton to create a complete executable model. Note that each DSL method generates a corresponding circuit-model. This feature has several benefits: (1) it allows having different implementations of the same circuit; (2) we can define multiple DSL methods as sub-circuits and use them in the main method (the main DSL methods). Although at this point the DSL code is syntactically correct, sometimes, its semantics are erroneous which makes the generated model improper. Therefore, it is worthwhile to make an integral check of the circuit-model. For instance, the meta-model is able to detect some common problems: (1) assignment to signals of different data types, (2) sequential statements outside of process or using combinational statements inside a process, (3) illegal operations on the data type of a signal, or (4) multi-source driving to a signal. In addition to this



```
HDLSketch class » compile:text
                  classified:aCategory
                  notifying:requestor
  |ast|
  ast := self compiler parse: text.
  (ast hasPragmaNamed: #hdl:) ifTrue:[
    ModelBuilder for:self
        ast:ast notificator:requestor.
    ^self compiledMethodFor:ast selector].
  ].
  ^super  compile:text
          classified:aCategory
          notifying:requestor.
```

Figure 2: The compilation process of the DSL

verification, in this phase, the meta-model handles also the automatic signal resizing in the operations of two or more LogicVector signals of different size.

At the end of the compilation process, the compiler generates the byte-codes that describes how the DSL code is executed (with or without hardware) when the method is called, as shown in listing 4. These byte-codes constitute a CompiledMethod. From the Smalltalk perspective, there is no difference between this method and other Smalltalk methods and thus it can be executed as any Smalltalk method.

```
HDLSketch class»compiledMethodFor:aMessage
  |source|
  source := aMessage,'
        self perform:#',aMessage.
  ^super compile:source classified:self
        category notifying:nil.
```

Listing 4: Generation of a CompiledMethod describing how the DSL code is executed

When the DSL method is called (e.g. the #execute method of SimpleALU), il will execute the #perform: method. This latter allows to automatically decide whether to perform the circuit in modelling mode or on the real hardware. The detail of this method can be found in listing 9 of section 5.

Like the traditional Smalltalk compilation process, the DSL method is compiled on-the-go whenever the DSL code is changed and saved, and therefore the circuit-model is updated accordingly.

3.3 Reuse of legacy VHDL code

One common problem when modelling hardware systems is how we can reuse existing VHDL designs in our model. Such reuse can enrich the models design while reducing the production cost by benefiting of an existing rich set of third-party VHDL code. Our meta-model supports the reuse of almost every VHDL designs conforming to the IEEE 1076.66 RTL standard. At the moment of this writing, the meta-model is able to import a VHDL design and reconstruct a black-box circuit model of it. That is, a model with only the external interface (input/output ports). Its internal architecture is "unknown" and thus we are unable to simulate it in Smalltalk. However, since it is a model based on our meta-model, we can easily integrate it with other circuit-models.

Obviously, a circuit that uses a black-box in our meta-model cannot be simulated in Smalltalk. However, it paves the way to co-simulation where the black-box runs on the real hardware and the rest is simulated by Smalltalk VM. We plan to extend our solution to support this feature in a future work.

The hardware execution of black-box models is easier since the VHDL code already exists and is ready for synthesis. There is no code generation needed. The meta-model only needs to generate the appropriate communication interface.

4. SMALLTALK-BASED EXECUTION OF THE FPGA CIRCUIT MODEL

4.1 Execution model: time-driven vs. event-driven

To understand how the simulated execution of FPGA circuits works in the Smalltalk environment, one must understand what kind of execution model is used in the meta-model. For such a model, two kinds of systems need to be taken in to account, the *continuous systems* and the *discrete systems* [11]. In the first ones, the state of the system (signals, ports) changes continuously with respect to time, whereas in the latter ones, the state changes instantaneously at separate points in times. In reality, there are few systems that are either completely continuous or discrete, although often, one type dominates the other. For example, a synchronous circuit that uses the global clock can be considered as a continuous system since its state can be changed at each clock. But at micro-level, when a part of the circuit is active, all related operations will be performed and make change on its outputs. This change, in consequence, will trigger instantaneously other parts connected to it. This process is repeated until the state of each part becomes stable. These parts, therefore, can be considered as discrete systems. The challenge here is to find an computational model that mimics closely the behaviour of such time-advance systems. There are, in fact, two models that can be used in this case: *time-driven* and *event-driven*.

A continuous system can be easily simulated using the *time-driven* [9, 11, 12]. With this approach, the simulation advances time with a fix increment of exactly Δt time units which is called simulation clock[2]. After each clock, the state of the system is updated for the interval of $[t, t+\Delta t]$. This approach, however, is not very appropriate for simulating a discrete system. For a such system, the time step Δt must be small enough to capture all events. Often, this time step is extremely small which is unacceptable as the simulation time involved. Furthermore, there are obviously empty time steps that cause wasting simulation time.

An *event-driven* simulation [9, 11, 13, 14] has a nature close to a discrete system. The simulation time in this case advances directly to the next-event time. An event represents a state change of the system caused by incoming data or internal processes. For the case of a discrete system, the approach consists of following steps: (1) the simulation time is initialised and all the occurrence times of future events are collected in an *event-queue*. (2) The simulation time is advanced to the closest *next-event* time in the *event-queue*. (3) The state of the system is updated by executing the scheduled events. (4) The *event-queue* is updated and the second step is repeated. The advantage of this method is that we can skip the periods of inactivity by advancing directly to the next event time and therefore, avoiding unnecessary empty cycles. In term of causality, this is perfectly safe because the state change of the system only happens at event times. This model of simulation can be easily applied to a continuous system since it is based on the occurrence of events and in such continuous system, each event occurs after a Δt interval of time.

This description shows why we use the *event-driven* simulation model in our meta-model.

4.2 Event-driven circuit-model execution simulation

4.2.1 Modelling the propagation execution of circuit's parts

As mentioned in section 2, a circuit may contain many combinational and process parts. A process (e.g. line 21-30, listing 1) has a sensitivity list consisting of signals (e.g. `clk`) that will trigger it as their values are changed. In our meta-model, all combinational parts are considered as *one-line* processes in which the inputs of each part are its sensitivity list. To trigger the processes we use the observer pattern. Each time a signal changes its value, it will announce

[2] The simulation clock is unrelated to the hardware clock and is used only by the simulator to keep track of the simulation time as the simulation proceeds

an event to all processes that have it in their sensitivity list, and thus active the corresponding processes. Listing 5 shows how these parts are executed when becoming active.

```
HDLSketch » partsExecution:msg
  [
    |processes|
    self signalsUpdate.
    processes := self getActiveProcesses:
        msg.
    processes do:[:p|p execute].
    processes notEmpty.
  ] whileTrue
```

Listing 5: Propagation execution of the circuit parts

Remember that each signal is an object with history, and it only takes the new value after being updated. In the signals update phase, if there is a value change in a signal, the #signalsUpdate message will activate all processes watching this signal. Since these processes are performed in parallel and the state of the circuit doesn't change until the next signals update, their execution order doesn't matter. This propagation execution is repeated until all signals stabilise and there are no active processes.

4.2.2 Circuit-model execution in responding to incoming data

Based on the #partsExecution, we can model the complete circuit execution in responding to the incoming data. There are two methods of execution depending on whether the simulation timing information is important or not.

Execution without timing information:

If we just want to assign input values to the circuit, execute it and get back its outputs regardless of how much time it takes to complete the calculation, the execution of the circuit can be implemented as shown in listing 6.

```
HDLSketch » modellingExecution:msg
  |processes|
  [ self done ] whileFalse:[
    self clk clock.
    self partsExecution:msg.
  ].
```

Listing 6: Circuit execution without timing informations

This is how we make the circuit acts like a normal object in Smalltalk. The inputs are assigned to values at the first time. By default, the circuits have a done signal indicating whether the calculation is finished. Execution is performed continuously at each clock until the done signal is asserted.

The code shown in figure 1 (section 2) is based on this principle. After the value assignment to A, B and opcode, the #execute method is called which in turn executes the #perform: method (listing 9). If the circuit is in modelling mode, the #modellingExecution: will carry out the execution task. The output R will be accessible after this execution finishes.

Execution with precise timing information:

If we want an execution simulation with all timing information (like in traditional simulators), the execution, in this case, is implemented along with a time queue. This latter contains the next-event times and the signals that will be assigned to new values in each event, as illustrated in listing 7. We can consider this time-queue as a test-bench in traditional simulators.

```
HDLSketch » modellingExecution:msg
    timeQueue:queue dumpOn:aStream
  |candidate|
  [
    candidate := queue nextTimeEvent.
    candidate notNil.
  ] whileTrue:[
    self signalsAssignment:(candidate
        signals).
    self partsExecution:msg.
    self takeSnapshotAt:candidate time
        on:aStream
  ].
```

Listing 7: Execution simulation with a time-queue

At each execution step, the simulation time advances to the closest next-event time, the inputs signals related to this event are assigned to new values and the propagation execution is performed. At the end of each event, the state of the circuit along with the current timing information is recorded in a Value Change Dump file (VCD). This process is repeated until the time-queue is empty. The final VCD file can then be viewed by a external VCD viewer to inspect the simulation result.

```
alu := SimpleALU new.
queue := {
  #clk clock: 50ns.
  #A change:{1. 3. 5. 7} every:50ns.
  #B change:{0. 2. 4. 6} every:50ns.
  #opcode change:{'0'.'1'} every:100ns.
} asTimeQueueFor:400ns.
stream := WaveFormStream on:'ALU.vcd'.
alu modellingExecution:#execute timeQueue:
    queue dumpOn:stream.
```

Listing 8: Simulating the SimpleALU circuit using the time queue. The simulation execution is performed for 400 nanoseconds

For example, the listing 8 shows how the SimpleALU can be simulated with a time queue and figure 3 how the simulation result can be inspected in a VCD viewer.

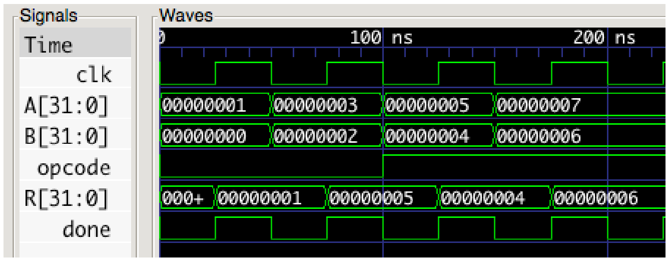


Figure 3: Waveform view of the simulation result (listing 8)

Although the meta-model is able to simulate the propagation delay behaviour of the circuit parts, there is no way to quantitatively measure this delay. Therefore, it is impossible for the meta-model to detect whether the delay time is greater than the global clock period. This verification is necessary to ensure that the circuit meets all timing constraints. Such verification need to be performed on the FPGA vendor's specific software.

5. HARDWARE-BASED EXECUTION OF THE FPGA CIRCUIT MODEL

5.1 Low-level synthesis of a circuit-model on FPGA

To deploy the circuit on FPGA for hardware execution, it needs to be synthesized first. The model of a circuit can be easily exported to VHDL for synthesis. The problem is that each circuit has an arbitrary external interface, which obstructs transparent Smalltalk interaction during FPGA execution. We need, therefore, an intermediate and normalised communication interface between the circuit and the Smalltalk, as illustrated in the figure 4. This interface must be capable of: (1) automatically decoding the data it receives from Smalltalk and assigning values to corresponding input ports of the circuit; (2) at the end of execution, encoding the output values of the circuit into a format that can be understood by Smalltalk. In the example shown in figure 4, we add an additional interface circuit (FX2 interface) to the global circuit and connect it to the `SimpleALU`. This interface circuit is generated automatically and directly using our meta model (without the DSL). The interface's internal architecture changes dynamically depending on which circuit it connects to.

After the final circuit is generated, synthesis proceeds as follows. A FPGA hardware description is produced and the synthesiser is initialized. This synthesiser is an abstract representation of the synthesis toolchain provided by the hardware vendor. From the circuit-model, the VHDL code of the circuit is exported and passed to this toolchain (along with the hardware configuration) for low level synthesis. This process, at the end, produces a binary representation (bit file) of the circuit which can then be deployed easily on the FPGA. Since the low-level synthesis process takes an significant amount of time to complete, it's worth to do it offline.

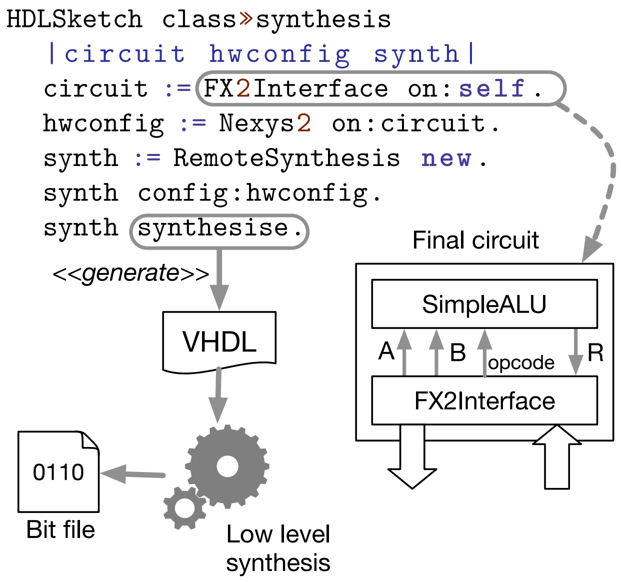

```
HDLSketch class»synthesis
   |circuit hwconfig synth|
   circuit := FX2Interface on:self.
   hwconfig := Nexys2 on:circuit.
   synth := RemoteSynthesis new.
   synth config:hwconfig.
   synth synthesise.
```

Figure 4: Offline synthesis process to generate the binary description of the circuit for deployment on the Nexys 2 FPGA board

5.2 Hardware-based execution of the circuit-model

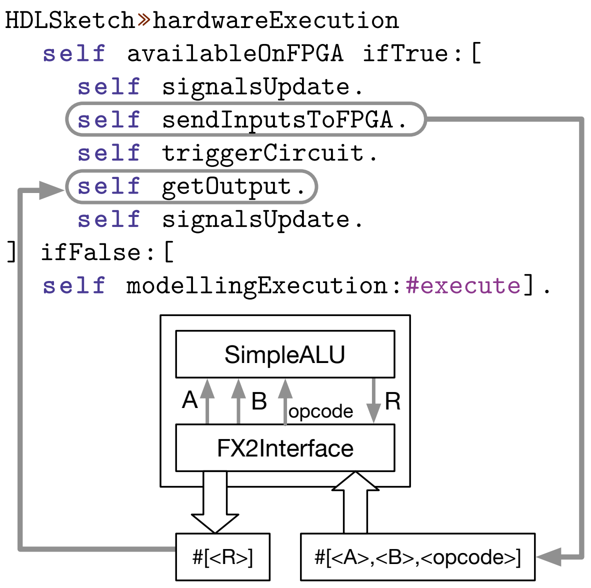
For each circuit-model, the meta-model generates automatically a signature which is an unique number that helps identifying the circuit. Normally, this value has no effect when simulating the circuit in Smalltalk, it is used only to detect whether the circuit is already deployed on the FPGA. In case the circuit is synthesized but isn't available on the FPGA, a deployment of the circuit is necessary before hardware execution. Listing 9 describes how the model switches between Smalltalk and hardware-based execution.

```
HDLSketch»performs:msg
  msg = self class mainMsg ifTrue:[
    self FPGAAvailable ifTrue:[
      self deployIfNotReady.
      self availableOnFPGA ifTrue:[
        ^self hardwareExecution.
    ]]].
  self modellingExecution:msg.
```

Listing 9: Execution switching between modelling mode and hardware mode

The deployed circuit acts as a blackbox whose internal state cannot be fetched. The model encodes its inputs (ports) as a Smalltalk `ByteArray` which is sent to the circuit via a Foreign Function Interface (FFI). This bytearray is then decoded by the interface circuit on hardware.

This process allows assigning the correct value to each input port of the target circuit. When the incoming data is ready, the target circuit is triggered and the calculation starts. The interface circuit will wait until the calculation is finished (i.e. the done signal is asserted) and encode the output data to a `ByteArray` that can be easily handled by the model. Figure 5 shows how the `SimpleALU` example can be executed on hardware in this way.



```
HDLSketch»hardwareExecution
   self availableOnFPGA ifTrue:[
      self signalsUpdate.
      self sendInputsToFPGA.
      self triggerCircuit.
      self getOutput.
      self signalsUpdate.
] ifFalse:[
   self modellingExecution:#execute].
```

Figure 5: Hardware execution of the SimpleALU circuit

5.3 Controllability and debugging

To enable software-like debug capabilities on hardware-based execution of the circuit model, the meta-model provides a mechanism to halt the execution flow by cutting out the clock supplied to the circuit. That is, instead of feeding the circuit with the global clock, the meta-model generates a *clock controller* which gets the global clock as input and issues a controllable clock signal to the circuit. This controller can disable the output clock when a control signal is triggered (i.e. a breakpoint is set or met its condition). When the clock is cut out, the execution of the circuit is stopped as a result and its current state is maintained. This state can easily be read back to the Smalltalk environment and mapped to the circuit-model via the communication interface.

To demonstrate these debug features, we consider a hardware implementation example of the famous $3n + 1$ problem also know as the *Collatz* conjecture, given any integer number $n > 0$:

$$F(n) = \begin{cases} \text{n/2} & \text{if } n \text{ is even} \\ \text{3n+1} & \text{if } n \text{ is odd} \end{cases}$$

Collatz conjecture states that, for any number n, repeatedly applying this procedure will eventually give us the number 1. Theoretically it could happen that this is not the case. So far, no such number has been found, but it has not been proven otherwise.

Assume the class `CollatzConjecture` implements a circuit model of this problem. The circuit takes an integer (i.e. n) as input and repeatedly recalculates the value of n at each clock. The execution is done when the output value is 1, otherwise, it is repeated indefinitely. The implementation details of `CollatzConjecture` using our DSL can be found in listing 13 in the appendix (it is not an efficient implementation though).

Static breakpoints

Setting a static breakpoint on a circuit model is easy by sending the `#halt` message to it, as in any Smalltalk code.

```
CollatzConjecture»execute
   <hdl:#combinational>
   ...
        nValue = 2 ifTrue:[
           self halt."n=2 halting"
        ] ifFalse:[
           (nValue at:0) ifFalse:[
              nValueNext <- ('0',(nValue
                  from: 31 downto:1))."n/2"
           ] ifTrue:[ "3n+1"
              nValueNext <- (((nValue from: 3
                  0 downto:0),'0')+nValue+1
                  ).
        ]].
   ...
```

Listing 10: A segment of the CollatzConjecture code shows how we can set the unconditional breakpoint for the circuit

For example, say we want to verify whether the value of n has converged to 2. We can modify the code in the listing 13 by halting the execution when the value of n is 2 as shown in listing 10.

When the meta-model compiles this code to build the circuit-model, it will generate automatically a `halt` signal that will be activated each time the value of `nValue` is 2. This signal is connected to the clock controller as control signal (figure 6). Therefore, its activation will disable the circuit clock and stop the execution forever (until reset).

The *clock counter* measures the execution time of the circuit in clock cycles. It starts as the circuit enters the computations and stops when the done signal of the circuit is asserted. Since the counter uses the same clock as the circuit, it is also halted when the clock is disabled. In this way, we can inspect exactly the execution time of the circuit at the breakpoint.

Because the communication interface (`FX2Interface`) uses the global clock, it is not affected by the `halt` signal and thus can normally fetch the circuit state and map

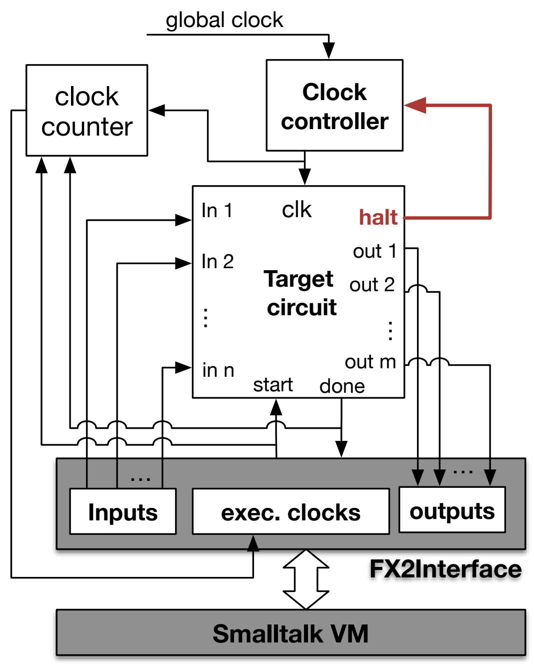


Figure 6: The clock `clk` is disabled each time the `halt` signal is asserted. The *clock counter* counts the total clock cycles of the execution

back values to the circuit-model on the Smalltalk side. Listing 11 shows how we can inspect the circuit state at the breakpoint.

```
obj := CollatzConjecture new.
obj input:10.
obj execute.
Transcript show: obj output asInteger;cr.
Transcript show: 'No. step: ', obj
    executionClocks asString.
"2
No. step: 5"
```

Listing 11: The modified CollatzConjecture circuit (with breakpoint) is executed with *input=10* and is halted at value 2 after 5 steps (clocks)

Since the unconditional breakpoint is hardwired in the model, the circuit needs to be re-synthesized and deployed to the hardware each time it is changed.

Dynamic breakpoints

Enabling the dynamic conditional breakpoint on the hardware-based execution is much more challenging, since the breakpoint is set manually and dynamically at run time without changing the circuit architecture. To enable this functionality, each time the circuit is deployed, the meta-model generates and injects automatically a debug-specific sub-circuit to the main circuit as shown in figure 7.

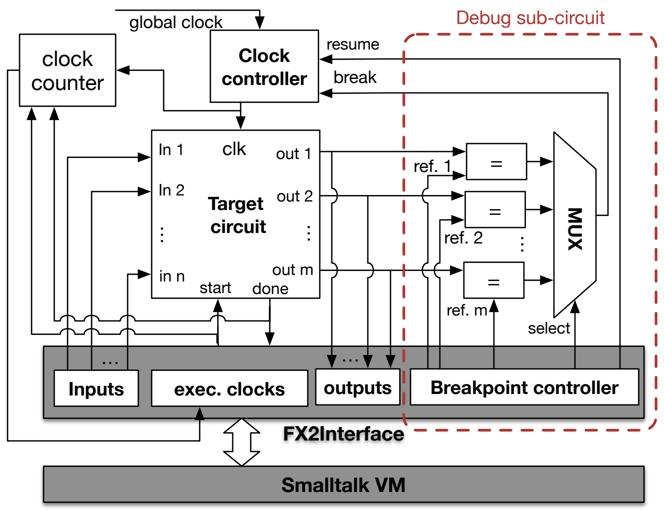


Figure 7: The debug sub-circuit added to the main circuit allows to dynamically set breakpoint conditions on each output of the target circuit

The debug sub-circuit connects each output of the target circuit to a *comparator* as first operand, the second operand is set manually by the *Breakpoint controller*. This allows to dynamically set a trigger condition to a breakpoint. The comparator is asserted when the two operands are equal. In this way, each output port of the circuit has a corresponding breakpoint. It is the responsibility of the *Breakpoint controller* to decide which breakpoint is used to stop the execution flow (by issuing the `select` signal to the multiplexer MUX). Through this *controller* we can manually set conditional breakpoint to the circuit from Smalltalk environment by sending the message `#setBreakpointOn:aPort value:aValue` to the circuit-model.

The advantage of this approach is that there's no need to modify the circuit architecture and the breakpoint condition can be dynamically changed at will. Furthermore, we can have multiple conditional breakpoints at a time and can specify which one will be triggered when its condition is met.

The specified breakpoint, when asserted, will trigger the `break` signal on the clock controller to disable the target circuit as well as the clock counter. At this point, the state of the circuit is ready for inspecting.

To resume the halted execution, the *Breakpoint controller* needs to trigger the resume signal to wake up the target circuit and the clock counter. The execution flow can then continue from that point. This resumption can be triggered from Smalltalk.

Back to the *CollatzConjecture* example (listing 13), assume the `input` value is 10, this time we want to verify

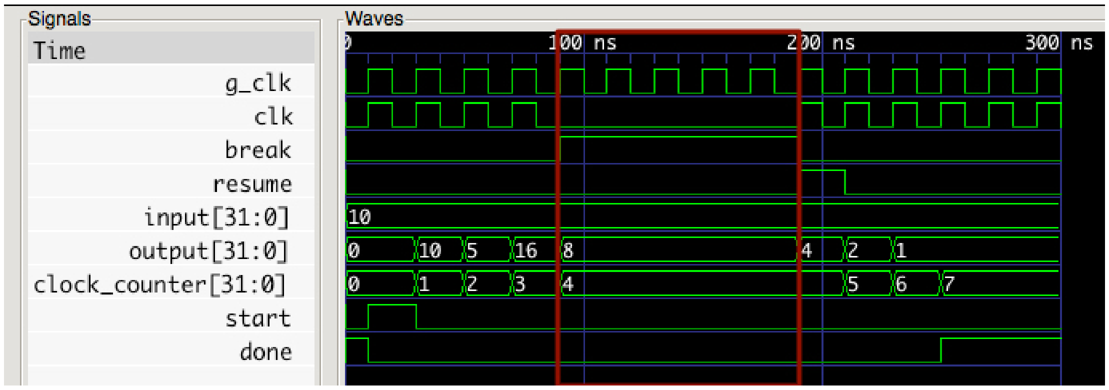


Figure 8: Software-based simulation of the CollatzConjecture circuit with breakpoint condition `output` = 8. The red zone shows the execution stopping time when the breakpoint condition was met

if the value of the output is 8 after some calculations (before it converges to 1). The listing 12 demonstrates how to stop the execution when output=8.

```
1  obj := CollatzConjecture new.
2  obj input:10.
3  obj setBreakpointOn:#output value:8.
4  obj enableBreakpointOn:#output.
5  obj execute.
6  Transcript  show:'Value:',obj output
       asString; show:', No. step: ', obj
       executionClocks asString.
7  obj resume.
8  Transcript  show:'Value:',obj output
       asString;show:', No. step: ', obj
       executionClocks asString.
9
10 "Value:8, No. step: 4"
11 "Value:1, No. step: 7"
```

Listing 12: At line 3, we set the breakpoint condition for the output port. Line 4 tells the *Breakpoint controller* to trigger the `break` signal when the `output` meets its condition. Line 7 continues the execution flow by triggering the `resume` signal

As shown in line 10-11, the execution of the circuit is stopped when the value of `output` is 8. Reaching this point takes 4 clock cycles. After resuming, `output` converges to 1 in 3 more clock cycles or 7 cycles altogether.

Figure 8 shows a software-based simulation result of the example to visually illustrate how the conditional breakpoint works. Obviously, the circuit-model with the conditional breakpoint can be executed directly on the real hardware.

6. CONCLUSION AND FUTURE WORK

In this paper, we have introduced a meta-model for modelling FPGA circuits using Smalltalk. This meta-model allows to describe and execute digital circuits at the RTL level. In this work, we focus mainly on an abstract execution of the circuit-model to make it act as a normal Smalltalk object. The meta-model captures the nature and the main characteristics of hardware circuits. We've developed a mechanism that helps run the generated circuit-model with (simulation mode) or without hardware. Moreover, to make the execution process transparent, a complete and automatic deployment and interface generation tool-chain is implemented and integrated inside the meta-model. The ability of setting breakpoint to the hardware-based execution of the circuit model is another valuable feature of the meta-model. This enables the software-like debug control capabilities on hardware debugging, and therefore facilitate the debug process.

In the short-term, we plan to extend this work in several directions to enrich the meta-model: (1) add more description to support archive elements like *library*, *package* or function call in VHDL. These elements enable efficient reuse of common hardware functionalities in the design process.(2) For the hardware-based execution method, the meta-model can only access the external state (ports) of the real circuit on hardware. Since the circuit acts as a blackbox in this case, its internal signals aren't externally visible. This can be resolved by adding "probe ports" to the circuit which connect to the internal signals that we want to inspect. (3) Support co-simulation of circuits that use black-box models in our meta-model. (4) Although the meta-model can convert existing legacy VHDL designs to black-box circuit models (i.e. only the external interface is imported), the current meta-model is unable to infer and entirely reconstruct the internal architecture of these VHDL designs. This can be overcome by implementing an additional VHDL parser that fully supports the IEEE 1076.66 RTL standard. With this parser, we can easily convert the VHDL descriptions to our meta-descriptions.

Based on this meta-model, our long-term plan is to build a Smalltalk-based HLS chain by rising up the abstraction level. Within our team, there is some work [2] that can be reused for this purpose. The challenge here is to propose a methodology to convert the Smalltalk dynamic code (dynamic type) to a typed data control and data flow graph (CDFG). Based on this CDFG, we can build the corresponding circuit using our meta-model.

References

[1] P. Coussy and A. Morawiec, *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer, 2008.

[2] M. Ben Hammouda, P. Coussy, and L. Lagadec, "A design approach to automatically synthesize ANSI-C assertions during High-Level Synthesis of hardware accelerators," *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 165–168, 2014. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6865091

[3] W. Meeus, K. Van Beeck, T. Goedeme, J. Meel, and D. Stroobandt, "An overview of today's high-level synthesis tools," *Design Automation for Embedded Systems*, vol. 16, no. 3, pp. 31–51, Aug. 2012. [Online]. Available: http://link.springer.com/10.1007/s10617-012-9096-8

[4] Synopsys, "Functional specification for SystemC 2.0," 2002.

[5] S. Swan, "An Introduction to System Level Modeling in SystemC 2 . 0," Cadence Design Systems, Inc., Tech. Rep. May, 2001.

[6] S. Vernalde, P. Schaumont, and I. Bolsens, "An object oriented programming approach for hardware design," *Proceedings. IEEE Computer Society Workshop on VLSI '99. System Design: Towards System-on-a-Chip Paradigm*, pp. 68–73. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=760478

[7] T. Kuhn and W. Rosenstiel, "Java based object oriented hardware specification and synthesis," *Proceedings 2000. Design Automation Conference. (IEEE Cat. No.00CH37106)*, pp. 579–581. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=835167

[8] M. Geilen, J. Voeten, P. van der Putten, L. van Bokhoven, and M. Stevens, "Object-oriented modelling and specification using SHE," *Computer Languages*, vol. 27, no. 1-3, pp. 19–38, Apr. 2001. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0096055101000145

[9] D. Picard and L. Lagadec, "Multi-Level Simulation of Heterogeneous Reconfigurable Platforms," *ReCoSoC'08, Barcelona, Spain*, 2008.

[10] IEEE, "IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis," Tech. Rep. October, 2004.

[11] P. Sloot, "Model Execution: Event driven versus Time driven." [Online]. Available: http://artemis.wszib.edu.pl/~sloot/1_4.html

[12] Wikipedia, "Time-driven programming," 2014. [Online]. Available: http://en.wikipedia.org/wiki/Time-driven_programming

[13] M. Samek, "State Machines for Event-Driven Systems," 2009.

[14] D. C. Schmidt and C. D. Cranor, "Half-sync/half-async - an architectural pattern for efficient and well-structured concurrent i/o," in *in Proceedings of the 2 nd Annual Conference on the Pattern Languages of Programs*. Addison-Wesley, 1995, pp. 1–10.

[15] "SystemC." [Online]. Available: http://systemc.org

[16] Wikipedia, "Collatz conjecture." [Online]. Available: http://en.wikipedia.org/wiki/Collatz_conjecture

A. Appendix

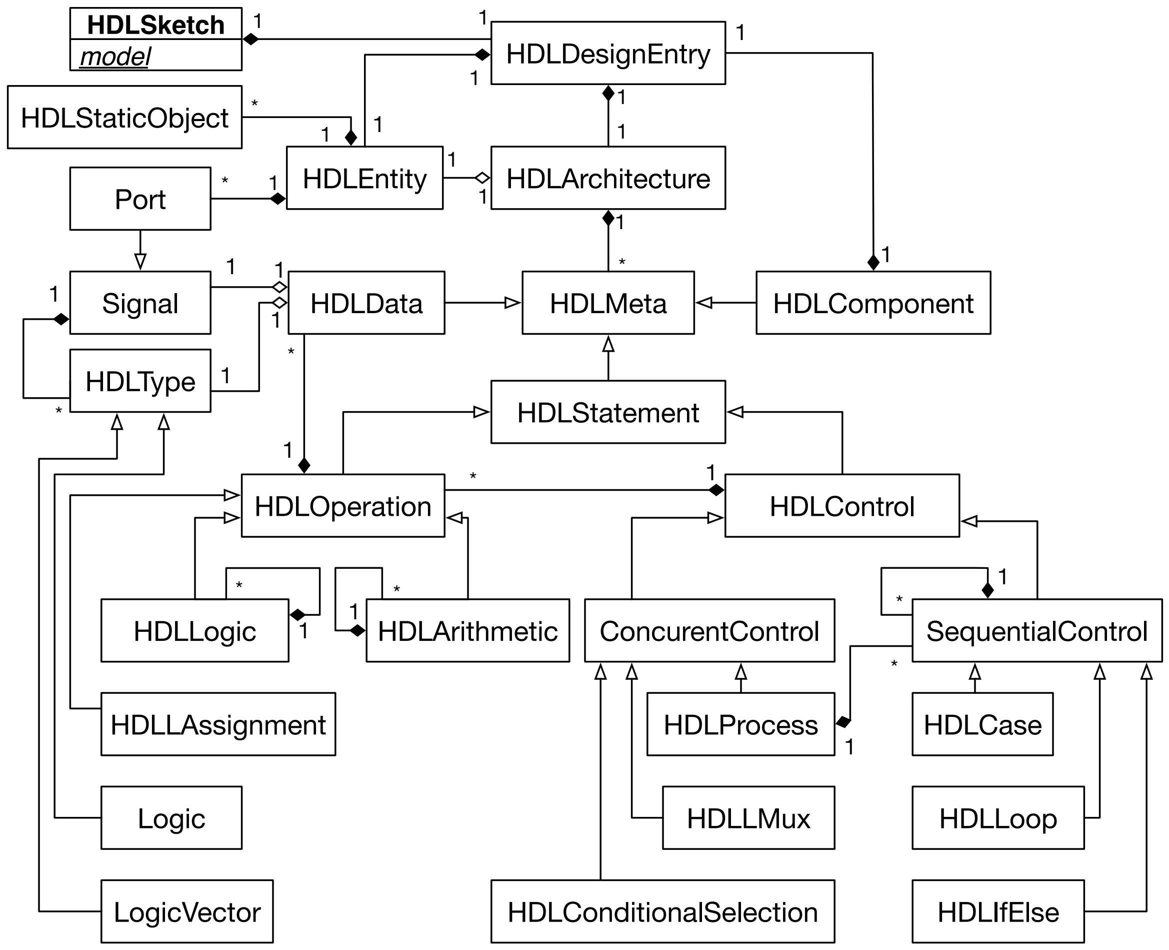


Figure 9: Simplified class diagram of our meta-model with respect to the IEEE 1076.66 RTL synthesis standard (VHDL 87) [10]

```
HDLSketch subclass: #CollatzConjecture
  instanceVariableNames: 'enum state stateNext nValue nValueNext'

CollatzConjecture»setUpPorts
  self in:#input of:(LogicVector size:32).
  self out:#output of:(LogicVector size:32).

CollatzConjecture»setUpSignals
  nValue := Signal of:(LogicVector size:32).
  nValueNext := Signal of:(LogicVector size:32).
  enum := Enum of:{ #IDLE. #EXEC}.
  state := Signal of:(enum?#IDLE).
  stateNext := Signal of:(enum?#IDLE).

CollatzConjecture»execute
  <hdl:#combinational>
  { self clk. self reset } onChange:[
    self reset = true ifTrue:[
      state <-(enum?#IDLE).
      nValue reset:false.
    ] ifFalse:[
      self clk posedge ifTrue:[
        state <- stateNext.
        nValue <- nValueNext.
      ].
    ].
  ].
  { state. self start. nValue. self input} onChange: [
    stateNext <- state.
    nValueNext <- nValue.
    state caseOf: {
      [ enum?#IDLE ]->[
        self done <- true.
        self start = true ifTrue: [
          self done <- false.
          nValueNext <- (self input).
          stateNext <- (enum?#EXEC)
        ].
      ].
      [ enum?#EXEC ]->[
        self done <- false.
        nValue = 1 ifTrue:[
          "n=1 finish the calculation"
          stateNext <- (enum?#IDLE).
        ] ifFalse:[
          (nValue at:0) ifFalse:[
            "n=n/2"
            nValueNext <-  ('0',(nValue from: 31 downto:1)).
          ] ifTrue:[
            "n=3*n+1"
            nValueNext <- (((nValue from: 30 downto:0),'0') + nValue + 1).
          ].
        ]
      ].
    }
  self output <- (nValue).
```

Listing 13: Hardware implementation of the Collatz conjecture problem using our DSL