UML Modeling in Design of Error Detection and Correction Circuits

St. Stancescu, L. Raicea, R. Marinescu, E. Enoiu
University POLITEHNICA of Bucharest
Iuliu Maniu 1-3 Blvd, Bucharest, 061071
E-mail: stst@elia.pub.ro, lavinia.raicea@gmail.com

Abstract - The Unified Modeling Language (UML) is used for designing, testing and validating system on chip architectures, real time systems, and embedded systems. The UML models are easily up-gradable an reused in new designs, being a powerful tool in extending functionality whenever necessary. Our approach describes an UML model for error detection and correction algorithms and also for hardware components that will perform the functions correspondingly, based on polynomial registers mod p(x).

I. INTRODUCTION

The aim is to enable a method to compile a UML based specification of embedded systems into Verilog code which can be subsequently synthesized for implementation onto an FPGA. Ideally [1], the UML specification should consist of a mixture of Class and State Diagrams (see figure 1). The Class diagrams will define a configuration of predefined modules and their connectivity. State diagrams facilitate the specification of a state based behavior for any component whose behavior can be thus expressed [2].

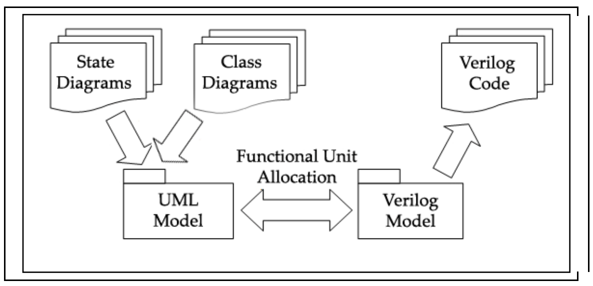


Fig. 1. Design Flow

II. UML MODELLING

We use two types of UML diagrams for modeling, namely, class and state diagrams. Class diagrams are predominantly used to describe the component structure of a system while state diagrams describe the behavior of the components.

II.A.Class Diagrams

We use the class hierarchy to describe the computational entities via their methods and their data types. However, class diagrams are also used in a crucial way to give an overview of a system in terms of components and how the components are connected to each other.

Classes can be related by the following UML relations [1]:

• Generalization (or inheritance): when a channel implements an interface, it inherits that interface. Moreover, an interface, channel and module can inherit another interface, channel and module respectively.

• Aggregation/composition: modules and channels may be hierarchical.

• Association: classes that exchange messages with each other are associated to one another. We model messages by UML events with or without arguments. Furthermore, a module may have an association relationship with an interface when it accesses a channel through this interface.

II.B.Object Diagrams

The UML object diagram lists the concrete objects used to compute test cases, and defines the initial state of the model. Each object diagram must be an instantiation of the associated class diagram. Notice the following restriction from [1]: objects can not be created or deleted dynamically by the actions designed in the model. So all objects used to describe the life cycle of the system must be defined in the object diagram. The dynamic creation (respective deletion) of entities in the concrete system is simulated by creation (respective deletion) of links between objects in the UML model.

The following UML elements can be used in object diagrams:

• Objects, or class instances, are the concrete objects of the system that are used in the generated tests. Every slot – or attribute instance – of every object must have a value.

• Links, or association instances, define the dependencies between the objects in the initial state of the system.

II.C. State Diagrams

State diagrams describe the behavior of a class. A state can be a simple state or a composite state. A composite state may be concurrent (often called an AND state). A composite state which is not a concurrent state is called an OR state. Being in an AND state means being in all of its sub-states. Being in an OR state means being in exactly one of its sub-states.

The UML state diagram formalism is a variant of State charts invented by David Harel [1]. Harel introduced diagrams that extend traditional state-transition diagrams with the notions of hierarchy and concurrency. A state diagram is a technique to describe the behavior of a system and represents the event triggered flow of control where objects change state by means of transitions.

III. FUNCTIONAL UNIT ALLOCATION

Functional Unit Allocation is the unit who sets and classifies simple digital circuit units from a special made repository. From [3] we know that hardware is latch-capable if it intrinsically can latch outputs from a specified task without requiring additional latches to be explicitly placed. A FIFO and a D-Flip/Flop are examples of latch-capable hardware.

When functional units are allocated, they become an additional set of constraints. The functional unit allocation runs this new set of constraints to the Verilog model. Functional units (or IP cores) that can be used are multipliers, adders, shifters, FIFO, etc.

Using the UML Tool's capabilities for exporting UML models as XMI files (XML based syntax) we can approach the mapping of the UML meta-models to Verilog modules, by using an intermediary step. The XMI file describes exactly the structure of the UML model and using a simple parser and translator can be brought to the form of Verilog Code. An interesting idea may be using XSLT for this transformation, which is a non-proprietary XML based language also.

IV. VERILOG MODELING

Verilog provides the capability to design a digital system in a modular fashion. Entire systems can be viewed as being composed of numerous individual modules. A Verilog module is a system block with well defined:1. input signals, say x1,x2,...,xn; 2. output signals, say y1,y2,...,ym and 3. internal structure and connections or behavior.[4]

Combinational circuits (CCs) produce output signals based on the value of their current input signals. A CC can have any number of input variables (input signals). We consider a CC as a graph of logic gates interconnected according to the logic function they compute. CCs route and transform the input values to the desired output values.

IV. A. Structural Hardware Modeling

In this style we model circuits by specifying the internal structure of the block. It is common to directly use the most primitive building-blocks available, such as, logic gates, or larger blocks which have already been defined elsewhere. Verilog provides NOT, OR, XOR, NAND, XNOR, NOR gates, among others. Verilog allows each logic gate to have any valid number of inputs.

From [4] we know that Verilog HDL allows circuits to be described in two ways, structural (how they are built), or in terms of behavior, (what they do). It is important to remember that when you describe your circuits' behavior rather than its structure then the synthesis tool has to translate the description into logic. The synthesis tool although very good at its job may not always translate the description into the logic you intended, and also may not produce the most efficient circuit possible. For these reasons it is useful to have an idea of how your description will be translated into logic.

The two types of coding styles in Verilog:

Structural or Continuous - used for primitive descriptions or data flow without storage

and(q, a, b); //structural assignment using Verilog primitive gate

assign q = a & b | c; //continuous assignment

Behavioural or Procedural - can be used to describe circuits with storage, and / or combinational logic:

```
always@(posedge clock)
begin
count <= count + 1;
if(a)
z <= b;
end
```

Gate level of abstraction describes the circuit purely in terms of gates. This approach works well for simple circuits where the number of gates is small, because the designers can instantiate and connect each gate individually. Also this level is very intuitive to the designer because of the direct correspondence between the circuit diagram and the Verilog description.

If the design is very large then a higher level of abstraction should be used above gate level. Data Flow allows the designer to design a circuit in terms of the data flow between registers and how a design processes data rather than the implementation of individual gates. The design tools would then convert the description to a gate level circuit.[5]

This level of abstraction allows designers to describe the circuit's functionality in an algorithmic way. That is the designer describes the behavior of the circuit without any consideration to how the circuit transfers to hardware.

IV.B. Verilog Basic Unit Mapping

We will present an example of an UML model for a universal CRC serial register (modulo P(x) register): UML Class Diagram and Simplified UML State Machine Diagram.

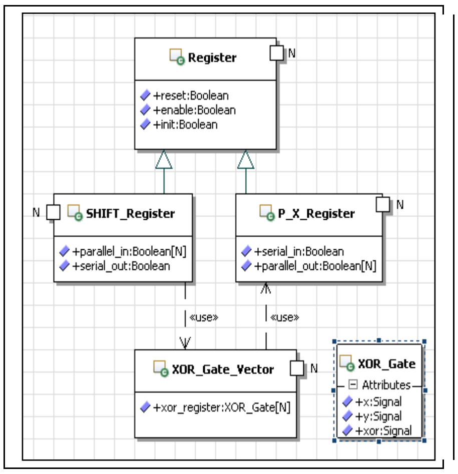


Fig 2. UML Class Diagram of a modulo P(x) register

The structure is centered on the "Register" class from which the other two classes are derived: "SHIFT_Register" as the register that performs the CRC calculations and the "P_X_Register" that memorizes the current P(x) coefficients. The class “XOR_Gate” is necessary to personalize the CRC register shifting according to the P(x) polynomial coefficients.

The behavior of the modulo P(x) register for a particular 0x1021 16-bit polynomial can be represented with a simplified UML State Diagram as in Figure 3.

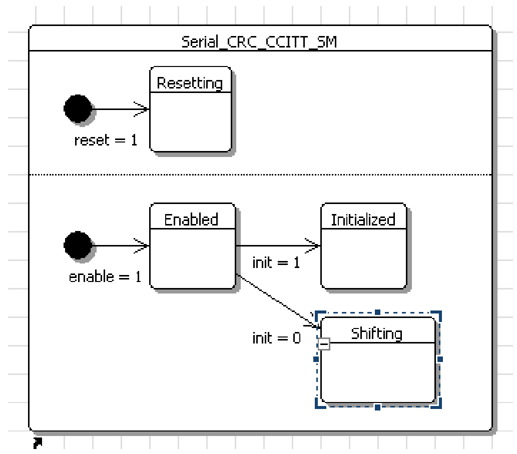


Fig. 3. Simplified UML Diagram for the modulo P(x) register

IV.C. *Error Decoding*

Decoding ECC [6] involves the extraction of two information entities from the received word. These are the set of Partial Syndromes, and the Error Locator Polynomial. From these two parameters, Error Locations and Error magnitudes are extracted, and are directly applied upon the rec We will present an example of an UML simplified model for a universal Reed-Solomon Decoder.

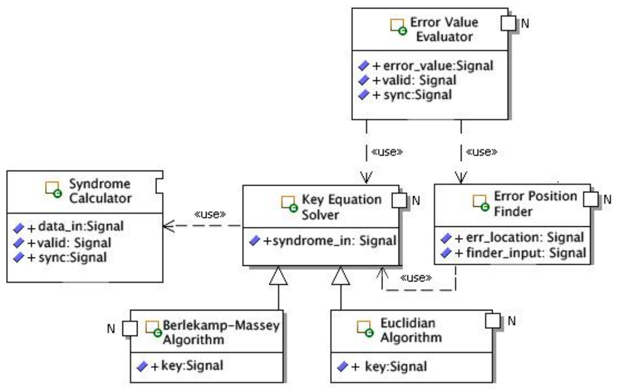


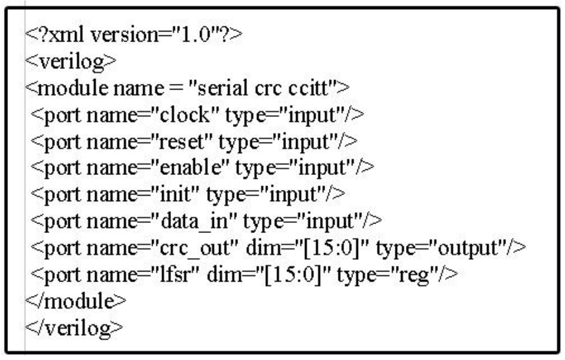
Figure 4. UML Class Diagram of a simplified Reed-Solomon Decoder

In the “Syndrome Calculator”class errors are detected by calculating the syndrome polynomial which is used by the “Key Equation Solver” Class to solve the key equation. The structure is centered on the “Key Equation Solver” class from which other two classes are derived: “Berlekamp-Massey” and the “Euclidian”. The Berlekamp-Massey algorithm is a shift-register synthesis algorithm [7] that has a complex structure but use fewer gates to be implemented than the Euclidian algorithm.

Based on the “Error Position Finder” class locations of the errors are determined. The magnitude of the errors are determined based on the error evaluator polynomial in the “Error Value Evaluator” Class.

IV.D. *Verilog Basic Unit Mapping*

Based on the UML Class Diagram and the UML State Diagram, we can obtain the XMI file generated by an UML Design Tool. This format is the OMG (Object Management Group) format for describing UML specifications and has been adopted by most UML Tools.



```
<?xml version="1.0"?>
<verilog>
<module name = "serial crc ccitt">
 <port name="clock" type="input"/>
 <port name="reset" type="input"/>
 <port name="enable" type="input"/>
 <port name="init" type="input"/>
 <port name="data_in" type="input"/>
 <port name="crc_out" dim="[15:0]" type="output"/>
 <port name="lfsr" dim="[15:0]" type="reg"/>
</module>
</verilog>
```

Fig. 4. Structural XML-based representation of Serial CRC CCITT

We present an example, how the markup technology XML generation can be applied to structurally represent this ECC model. As an example, we take the interface description of the Serial CRC CCITT model in Verilog language (module) (see Figure 4). The description has two hierarchical levels of structure: entity at the higher level and its I/O ports at the lower level.

Furthermore, each port has the following attributes: port name, type (input, output, etc.) and optional the dim for the dimension of the port.

When introducing the markup information into the model, each level of abstraction is replaced with the corresponding tag, which may be defined freely by the designer.

For example, the domain language level can be marked by <verilog> tag, the component interface level-by <module> tag, and port level-by <port> tag. Furthermore, the attributes of the abstraction are replaced by the property of the tag and its value. The property of tag can be defined freely, whereas the value of the property must be the same as the attribute of the abstraction. For example, module CRC can be marked as <module name= ”CRC”/>.

Using the XSLT transformation language we will transform the XMI file and generate a Verilog implementation.

```
  module  serial_crc_ccitt  (clk,  reset,  enable,
init, data_in, crc_out);
  input clk;
  input reset;
  input enable;
  input init;
  input data_in;
  output [15:0] crc_out;
  reg [15:0] lfsr;
  assign crc_out = lfsr;
  always @ (posedge clk)
  if (reset) begin lfsr <= 16'hFFFF; end
  else if (enable) begin
    if (init) begin lfsr <=  16'hFFFF; end
    else begin
      lfsr[0]  <= data_in ^ lfsr[15];
      lfsr[1]  <= lfsr[0];
      lfsr[2]  <= lfsr[1];
      lfsr[3]  <= lfsr[2];
      lfsr[4]  <= lfsr[3];
      lfsr[5]  <= lfsr[4] ^ data_in ^ lfsr[15];
      lfsr[6]  <= lfsr[5];
      lfsr[7]  <= lfsr[6];
      lfsr[8]  <= lfsr[7];
      lfsr[9]  <= lfsr[8];
      lfsr[10] <= lfsr[9];
      lfsr[11] <= lfsr[10];
      lfsr[12] <= lfsr[11] ^ data_in ^ lfsr[15];
      lfsr[13] <= lfsr[12];
      lfsr[14] <= lfsr[13];
    end
  end
  endmodule
```

Figure 5. A Verilog Module for a particular 0x1021 16-bit polynomial.

The design started with the creation of UML Diagrams that describe the essential elements of modulo P(x) data manipulation: shift register, XOR Matrix, register for memorizing the particular implemented P(x). This was done both structural and behavioral by the appropriate UML diagrams (Figure 2 and Figure 3). The process continued with the XMI generation and was finalized in a Verilog implementation as exemplified in Figure 5.

V. CONCLUSION

We have presented a methodology for designing ECC circuits using the UML meta-model to obtain the Verilog implementation for that specific circuit. The methodology includes the UML Class Diagram and State Machine Diagram conception, the XMI generation and XMI-XSLT translation to Verilog. This methodology can be applied for implementing various circuits with a diverse range of P(x) polynomial, code length, detecting and correcting capabilities, specified in the appropriate and detailed UML Class Diagrams.

REFERENCES

[1]J. Rumbaugh, I. Jacobson, and G. Booch, The Unified Modeling Language Reference Manual. Addison-Wesley, 1998

[2]M. Lajolo, IP-Based SOC Design in a C-based design methodology," in Proc. Of IP Based SoC Design 2003, pp. 203-208, Oct. 2003.

[3]A. Minosi, S. Mankan, A. Martinola, F. Balzarini, A. Kostadinov, and M. Prevostini, “UML-based Specifications of an Embedded Systems Oriented to HW/SW Partitioning: a Case Study" in FDL'03 Proceedings, pp. 226-237, Sep. 2003.

[4]Thomas, Donald, Moorby, Phillip "The Verilog Hardware Description Language", Springer, 5th edition (June 30, 2002).

[5]Janick Bergerdon, "Writing Testbenches: Functional Verification of HDL Models", 2000, (The HDL Testbench Bible).

[6]Kenny Chung Chung Wai, Dr. Shanchieh Jay Yang, “Field Programmable Gate Array Implementation of Reed-Solomon Code, RS(255,239)”, Integrated System Design Magazine, 2005.

[7]S. B. Wicker, Error control Systems for Digital Communication and Storage, Prentice Hall, 1995