International Conference on Control, Engineering & Information Technology (CEIT'13)
Proceedings Engineering & Technology - Vol.3, pp. 39-44, 2013

A SysML and Clean based Methodology for digital Circuits Design and Simulation

Zakaria Lakhdara[#1], Salah Merniz[#2]

[#]*LIRE Laboratory, Computer Science Department, Constantine 2 University Constantine Algeria*

[1]lakhdara.z@umc.edu.dz

[2]s_merniz@ umc.edu.dz

***Abstract*— Due to the ever complexity of digital systems, there is a noticeable need for more abstract and structural mechanisms as well as design methodologies that systematically and formally derive low level concrete designs from high level abstract ones. To this aim, we present a methodological design approach that automatically generates a functional HDL code from SysML diagrams modeling hardware designs. The generated HDL code is both reliable and executable. While the first feature remains crucial for low level design refinements, the second one enables evaluating design performances at early stages. A case study involving the functional implementation of an ALU (Arithmetic Logic Unit) through Clean code generated from high level SysML diagrams is given, to practically show the potential features of the proposed approach.**

***Keywords*— Digital circuits design, SysML, Clean, Functional specification, Modeling, Simulation.**

I. INTRODUCTION

After the wide and successful application of UML in the area of software engineering, there is a growing interest in the use of UML at the high level of abstraction for modeling hardware designs. To this aim, researchers have created UML profiles such as UML-SystemC [2], UML-SoC [18], MARTE [8], or SysML [3] that extend UML with the appropriate constructs to hierarchically describe complex hardware designs and analyze their properties. Thereafter, they have created approaches to generate a system level implementation through imperative intermediate HDL code such as VHDL [3, 5], Verilog [5, 7] or SystemC [1, 2, 5, 6], to bridging the gap between UML high level description and the micro-architecture level description.

The main problem is that, imperative HDLs lack a well-defined semantic definition and consequently it becomes very hard to formally deriving thereafter low level design refinements that reflect the high level requirements (modeled by UML which still lacks a formal semantics). Therefore, it is necessary first, to validate the generated HDL code before to proceed toward low level implementations. While some approaches try to validate the generated HDL code by translating it to formal models [10], most others use simulation-based methods to validate such code. Both approaches present drawbacks. Formal models are mostly model checking-based and consequently they are still suffering from the state explosion problem, whereas simulation-based methods are very insufficient to cope with the growing complexity which according to the Moore's law, doubles almost every two years. At best, simulation methods which have also the disadvantage to lengthen the time-to-market, can only reduce the number of design faults, but never certify the design correctness.

This work proposes an alternative design methodology that uses the functional language Clean [13] as HDL at the system level design and use SysML [12] as a modeling language at the top of Clean. An automatic mapping from SysML diagrams represented as XMI file format to Clean HDL code is achieved using our proper code generator.

SysML (systems modeling language) is a UML profile (domain specific modeling language) for system engineering applications. It reuses a subset of UML 2.0, with extensions; including the notion of block (which replaces the notion of class), the modeling of requirements and the parametric constraints. The notion of block allows to hierarchically developing and modularizing complex designs, whereas the notions of requirements and parametric enable requirements engineering and performance analysis of such designs. Moreover SysML supports model and data interchange via XML Metadata Interchange (XMI) interface. It is a standard based approach with the aim to improve communications, tool interoperability, and design quality.

Clean has a graph rewriting semantics [11] which is very attractive for generating a reliable and executable code. This is very crucial at the system level, from which the synthesis process starts refining the low level design to produce physical circuits (which is the aim of this project). Beside its formal semantics which is also very important for formal reasoning (to optimize the code and prove its correctness), Clean provides powerful features that are very useful for dealing nicely with hardware de-signs. The key features are given bellow:

- *Function composition*: In hardware design, circuits are built hierarchically in terms of primitive components in the same way as function composition by application, abstraction and local naming.
- *Type-classes*: Provide a convenient mechanism for abstracting over circuit descriptions, and also provide means to generically deriving specific classes of circuits
- *Non-strict semantics*: Allows reasoning about descriptions involving unbounded structures. For example signals can be elegantly implemented as lazy lists (streams).

- *Lazy evaluation*: Combined with non-strict semantics, naturally supports the development and simulation of mutually dependent descriptions such as cyclic structures (feedbacks). Lazy evaluation is more amenable to termination than eager evaluation.

- *Parametric polymorphism*: Allows generic functional descriptions to be used in different contexts

- *Higher order functions*: This feature enables a designer to structure descriptions in elegant and concise way.

- *Functional simulation*: A functional specification is directly executable. It allows the circuit's behavior to be observed at early stages.

The remainder of this paper is organized as follows: Section two discusses the related works. Section three presents the proposed design methodology. Section four gives a typical example, and finally section five concludes this work.

II. RELATED WORK

Most of related works on hardware design using UML as a high level modeling language generate imperative HDL codes at the system level. In [1], the authors pro-pose an approach for automatic generation of SystemC code from UML diagrams at early stage of SoCs (Systems on Chip) designs, using two levels of abstraction: In the first level, they use UML sequence diagrams to generate the SystemC code which is dedicated to algorithmic space exploration and simulation. In the second level, they implement the messages occurring in sequence diagrams using UML activity diagrams where actions are expressed in the C++ Action Language (AL). The goal is to generate a full SystemC code for both simulation and synthesis. In [3], the authors propose a mechanical technique to map SysML models into SystemC code through XMI file format. The works developed in [4] and [9] present two similar views that generate a synthesizable code (VHDL for the former and Handel-C for the later) from UML models. In [5], the authors use UML-HD (UML profile, dedicated for hardware) for modeling asynchronous hardware designs and an approach that automatically generates a Haste HDL code from UML-HD models in XMI file format. In [6], the authors propose a methodology for modeling digital designs, using UML diagrams from which they automatically generate a SystemC code for verification, and VerilogHDL / VHDL for FPGA implementation through XMI file format. The research work presented in [7], describe a prototype tool that automatically generates a Verilog HDL code and the corresponding System Verilog assertions (for correctness purpose) from UML models.

All these approaches generate an imperative code which requires a complex verification step before generating a correct synthesizable one.

III. DESIGN METHODOLOGY

The design methodology proposed here follows the top down layered approach. It uses SysML at the high level to model the structural aspect of hardware designs, and uses Clean as a functional HDL at the system level to formally implement (to give a semantics in terms of Graph Rewriting) SysML models taken as XMI file format. The resulting functional HDL code which is both reliable (directly verifiable through Clean theorem prover) and executable, will be used for three purposes: For further design synthesis (In this work we limit our self only to the generation of the functional HDL code, although the complete project proceeds toward netlist generation using Functional Graph Rewriting Systems), for formal verification and for functional simulation. The transformational design flow is depicted in Fig. 1.

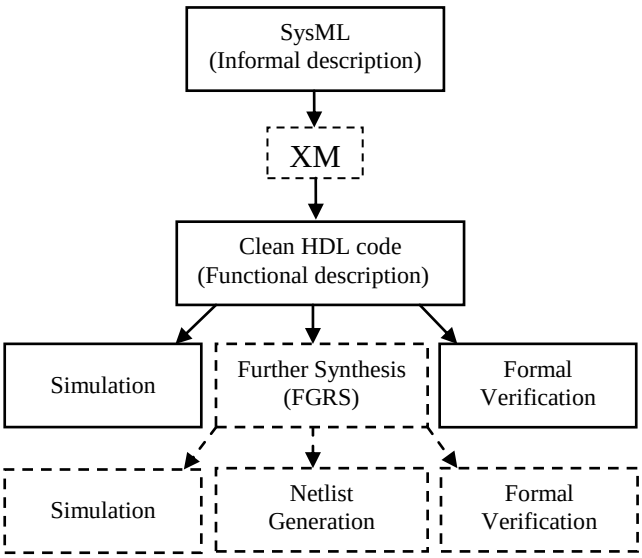

Fig. 1 Transformational design flow

A. Modeling with SysML

SysML provides attractive features for modeling both the behavior and the structure of hardware designs [14]. This work exploits only some structural diagrams (the Block Definition Diagram (BDD) and the Internal Block Diagram (IBD)) to describe the structural aspect of hardware designs. The notion of SysML Block implements perfectly the concept of module that performs a well-defined function and communicates with well-defined interfaces (defined by its inputs/outputs). Thereafter; the SysML block fits adequately the concept of modularity which is very important for developing complex designs.

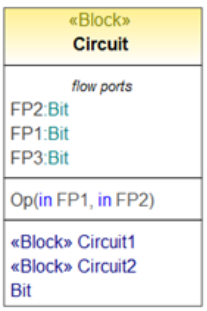

Fig. 2 SysML Block representation

A circuit is represented by a Block (stereotyped as <<block>>) whose Inputs/Outputs are specified using the SysML *FlowPorts* elements. The types of these *FlowPorts* are specified using the SysML *DataType* element. The task performed by the circuit is specified by the block's Operation which takes its parameters from the input *FlowPorts* and returns the result at the output *FlowPorts*. Fig. 2 shows a circuit description using SysML block representation. This circuit contains three *FlowPorts*: *FP1, FP2*, and *FP3*, with type *Bit* and performs an operation denoted *Op* that takes two inputs: *FP1* and *FP2*, and returns one output: *FP3*. Such circuit description declares also two sub-circuits denoted *Circuit1* and *Circuit2*, and a singleton *DataType* Bit indicating the type of the *FlowPorts* involved in both the circuit and its sub-circuits.

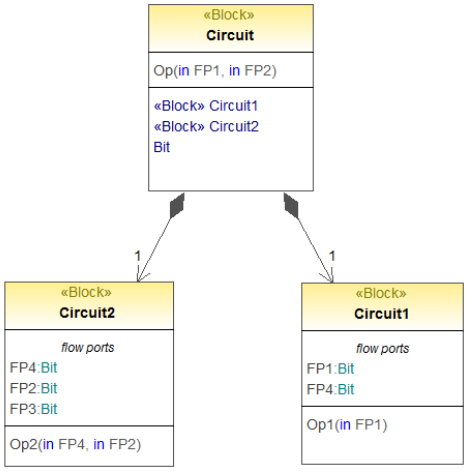

Fig. 3 Block Definition diagram of C*ircuit*

The BDD is used to hierarchically decompose a circuit using the notions of block and composition relationship (between a block and its components). Such hierarchical description is extremely important for developing complex digital circuits using the top down approach. Further sub-blocs decomposition is identical to that of the main block. Fig. 3 gives an example describing the circuit represented in Fig. 2 and its components.

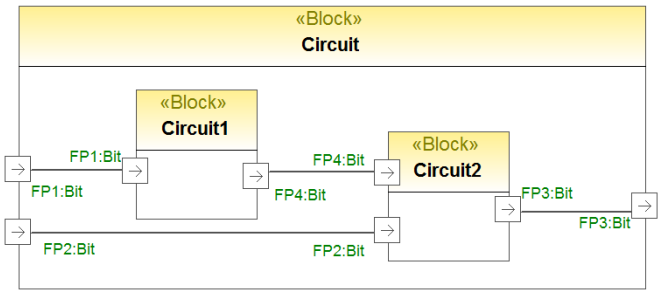

Fig. 4 Internal Block Diagram of Circuit

The details of the internal block structure could be shown using the IBD where the different internal sub-blocks are connected through their *FlowPorts* using the SysML *Connector* element. Fig. 4 shows the internal structure of the block *Circuit* represented in Fig. 3.

B. Mapping SysML Model to Clean Specification

The methodology derives a functional circuit specification from SysML models (using BDDs and IBDs) describing the structural aspect of a hardware design. The functional circuit specification is a clean module which contains a collection of function definitions and types. The generation process of the functional code from a high level SysML description is shown in Fig. 5; it involves three parts: A library, a parser and a code generator.

1) The Library: The library contains the functional description (structural aspect) of the primitive components (gates, registers, memories, etc), and their types. A complex circuit is built upon these primitive components using a top down decomposition and a bottom up design. Once built, a circuit is submitted to further activities such as simulation, verification or synthesis. Once simulated or verified, a circuit description is saved into the library as well. This enrichment helps in minimizing the design cost.

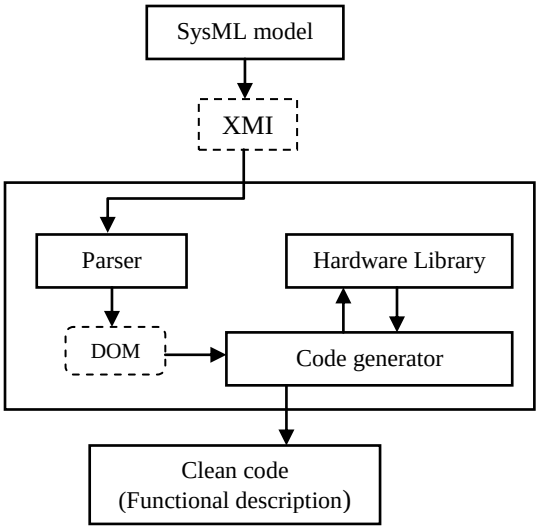

Fig. 5 Code generation with CleanSG

2) The Parser: The parser facilitates the process of manipulating an XMI file. It reads the XMI file and creates a document named DOM (Document Object Model) which contains a tree of object Nodes. The DOM delivers a complete parsed representation of the XMI file which can be used later by the code generator to access the XMI file components (entities, elements, attributes, etc). For more details about DOM parsers see [15].

3) *The Code Generator:* The SysML circuit model provides the necessary information for generating the corresponding Clean module. Initially, the code generator creates an empty module and progressively fills it using the functions describing the blocks involved in the circuit model and their types definition. The complete module construction consists of traversing the DOM (equivalently, decomposing

the main SysML block) till the lowest level of hierarchy. At each level, and for each block, the code generator launches a scanning task to search for a corresponding Clean function in the library. If so, then the target function and its type are inserted in the module, otherwise the corresponding Clean function is created automatically using the information provided by the SysML block representation shown in Fig. 2 (the block/sub-blocks Operation and the FlowPorts with their types). The main block will be interpreted as a function composition by application, abstraction and local naming; using the Clean let expression described in Fig. 6.

TABLE I
Correspondences between SysML and Clean

SysML Block representation	**Clean**
Block	Function
Input FlowPorts	Function arguments
Output FlowPorts	Function results
DataType	Type definition
Blocks composition	Functions composition

Functional HDLs have been efficiently used in the context of hardware design. In this work, we will propose a template function defined based on the correspondences between SysML and Clean (see Table 1). We use the Clean let expression as a powerful mechanism for capturing most of the important structural aspects of a digital circuit such as composition, parallelism, and hierarchy. The full definition is given below.

```
FuncName :: {arg_type} → result _type
FuncName {arg} = let
    Val1 = exp1
    ⋮
    Valn = expn
 in (exp)
```

Fig. 6 Function template

IV. Case Study

This section discusses a case study involving the functional implementation of an ALU (Arithmetic Logic Unit) through Clean code which is generated from high level SysML diagrams.

The modeling phase is performed using Altova UModel framework. This later enables designing application models in both UML and SysML, and exporting them as XMI file format (for more details see [17]).

The ALU being designed operates on words (of an arbitrary size). It takes as input two words: *as* and *bs*, a carry in bit *ci*, and two bits *s0* and *s1* for selecting the operation to be executed. It performs four operations: *Addition*, *And*, *Or* and *Not*. Fig. 7 represents the BDD of the ALU circuit; it is composed of three parts: a logic unit (LU), a word adder and a multiplexer four-to-one.

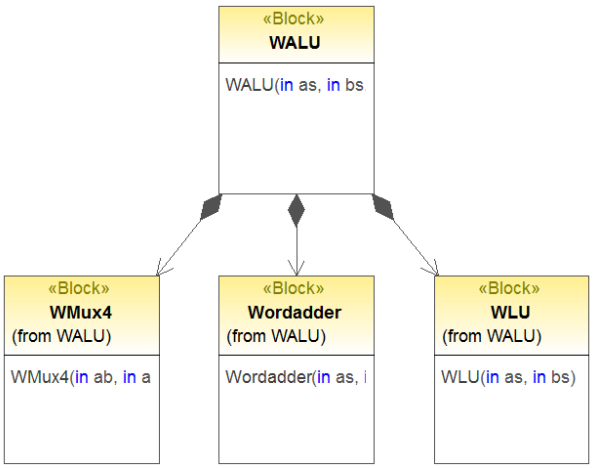

Fig. 7 IBD of the circuit WALU

After decomposing the circuit into several blocks, the internal structure of the main block and its sub-blocks is shown using the IBD. Fig. 8 represents the IBD of the block WALU. The IBDs of the other blocks are done by the same way.

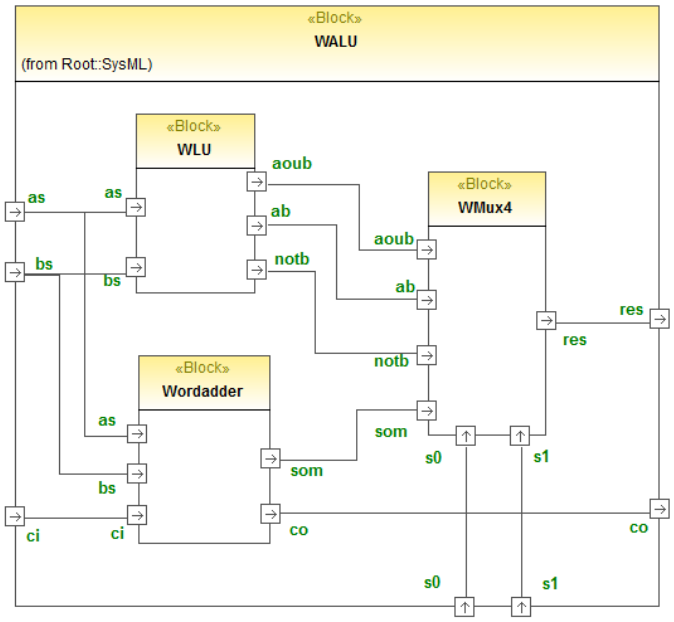

Fig. 8 IBD of the circuit WALU

After completing the modeling phase, the circuit model is exported as an XMI file format. Our tool CleanSG uses this file to generate the corresponding functional circuit specification. In this case study the circuit model shows that the circuit is composed of: *Xor*, *Not*, *WNot*, *And*, *WAnd*, *Or*, *WOr*, *Mux2*, *WMux2*, *WMux4*, *Halfadder*, *Fulladder*, *Wordadder*, *WLU*, and uses two data types *Bit* and *Word*. The library of CleanSG contains the definitions of these types and the functional specifications of these digital components except for the blocks *Wordadder*, *WLU* and the *WALU*. To generate the functional circuit specification, the code generator creates an empty Clean module and fills it from the library with the definitions of the types and the

functional specifications of the components involved in the circuit model. The missing ones are generated automatically, inserted in the module and saved in the library.

Here is the complete functional specification generated for the circuit ALU:

```
module WALU // module name

// types definitions
:: Bit :== Int
Bit = 0
Bit = 1

:: Word :== [Bit]

//functions describing the circuit's
     components
Xor :: Bit Bit -> Bit
Xor a 0 => a
Xor a 1 => Not a

Not :: Bit -> Bit
Not 0 => 1
Not 1 => 0

WNot :: Word -> Word
WNot [] = []
WNot [x:xs] => let
    s = Not x
    ss = WNot xs
in([s:ss])

And :: Bit Bit -> Bit
And a 0 => 0
And a 1 => a

WAnd :: Word Word -> Word
WAnd [] [] = []
WAnd [x:xs] [y:ys] => let
    s = And x y
    ss = WAnd xs ys
in([s:ss])

Or :: Bit Bit -> Bit
Or a 0 => a
Or a 1 => 1

WOr :: Word Word -> Word
WOr [] [] = []
WOr [x:xs] [y:ys] => let
    s = Or x y
    ss = WOr xs ys
in([s:ss])

Mux2 :: Bit Bit Bit -> Bit
Mux2 a b c => let
   s = Or (And a (Not c)) (And b c)
in(s)

WMux2 :: Word Word Bit -> Word
WMux2 [] [] c => []
WMux2 [x:xs] [y:ys] c => let
    s = Mux2 x y c
    ss = WMux2 xs ys c
in([s:ss])
```

```
WMux4 ::Word Word Word Word Bit Bit -> Word
WMux4 [] [] [] [] c1 c2 => []
WMux4 a b c d c1 c2 => let
 s = WMux2 (WMux2 a c c1)(WMux2 b d c1) c2
in(s)

Halfadder :: Bit Bit -> (Bit,Bit)
Halfadder a b => let
    s = Xor a b
    c = And a b
in (c,s)

Fulladder :: Bit Bit Bit -> (Bit,Bit)
Fulladder x y ci => let
    (c1,s1) = Halfadder x y
    (c2,s) = Halfadder s1 ci
    co = Xor c1 c2
in (co,s)

Wordadder :: Word Word Bit -> (Bit,Word)
Wordadder [] [] ci => (0,[])
Wordadder [x:xs] [y:ys] ci => let
    (co,s) = Fulladder x y c1
    (c1,ss) = Wordadder xs ys ci
in (co,[s:ss])

WLU :: Word Word -> (Word,Word,Word)
WLU [] [] => ([],[],[])
WLU a b => let
    notb = WNot b
    ab = WAnd a b
    aoub = WOr a b
in (ab,aoub,notb)

// the main function
WALU :: Word Word Bit Bit Bit ->(Word,Bit)
WALU [] [] ci f0 f1 => ([],0)
WALU a b ci f0 f1 => let
    (co,som) = Wordadder a b ci
    (ab,aoub,notb) = WLU a b
    res = WMux4 ab aoub notb som f0 f1
in (res,co)

Start = WALU as bs ci s0 s1
```

To simulate the designed circuit, we simply execute its functional specification by invoking the main function *WALU* using the following code:

```
Start = WALU as bs ci s0 s1
    Where
        as = [0,0,0,0,0,0,1,1]
        bs = [0,0,0,0,0,0,0,1]
        ci = 0
        s1 = 1
        s2 = 1
```

To verify the four operations performed by the ALU, for each operation, we set the corresponding operation code and we invoke the function *WALU*.

In order to perform several tests, we have defined a higher-order function called *test*. This function applies the function *WALU* on a list of different arguments. The definition of the function *test* is given bellow.

```
Test :: (Word Word Bit Bit Bit ->(Bit,Word))
         [Word] [Word] [Bit] [Bit] [Bit] ->
         [(Bit,Word)]
Test f [] [] [] [] [] = []
Test f [a:as] [b:bs] [c:cs] [f:fs] [s:ss] =
      [f a b c f s : Test f as bs cs fs ss]
```

To execute the function test, we invoke it using the following code:

```
Start = Test  WALU  as bs ci s1 s2
  where
   as =[[0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,1],
        [0,0,0,0,0,0,1,0],[0,0,0,0,0,0,1,1]]
   bs =[[0,0,0,0,0,0,1,1],[0,0,0,0,0,0,1,0],
        [0,0,0,0,0,0,0,1],[0,0,0,0,0,0,0,1]]
   ci = [0,0,1,0]
   s1 = [0,1,0,1]
   s2 = [0,0,1,1]
```

Invoking the function *test* with these arguments enables the execution of the function *WALU* four times. At each time, the function *WALU* performs a different operation on different values of the two data words *as* and *bs*. As a result, we get a list containing the result of the four operations performed by the function *WALU*.

```
(co,res) = [(0,[0,0,0,0,0,0,1,1]),
            (0,[0,0,0,0,0,0,1,1]),
            (1,[1,1,1,1,1,1,1,0]),
            (0,[0,0,0,0,0,1,0,0])]
```

V. Conclusion

In this paper we have presented a methodology for digital circuits design based on SysML and the functional language Clean. It involves a modeling technique based on two SysML structural diagrams BDD and IBD and a technique for automatic Clean code generation from a SysML model (in XMI file format). This transformation is performed by our proper tool CleanSG. The design methodology has been evaluated by means of a typical case study, involving a digital circuit say an ALU where a Clean code has been derived from SysML model describing this ALU.

This work gives a first contribution towards a research topic that has not been investigated so far, namely UML to functional HDLs. It combines the powerful features of SysML and the functional language Clean. Beside the availability of SysML documentation and tools, our modeling technic is simple and easy to use. The automatic code generation from an electronic schema and the reuse of the saved specifications in the library minimize the circuit design time. Saving the generated specifications in the library enriches it every time we design a circuit. The generated functional circuit specification gives the possibility to simulate the circuit.

Our future work will focus on enhancements toward modeling with SysML and code generation, extensions for formal verification, although the complete project proceeds toward Functional Graph Rewriting Systems.

References

[1] F. Boutekkouk. "Automatic SystemC Code Generation from UML Models at Early Stages of Systems on Chip Design", International Journal of Computer Applications (0975 – 8887) Vol. 8 – No.6, October. 2010.

[2] F. Mischkalla, D. He, W. Mueller. "A UML Profile for SysML-Based Comodeling for Embedded Systems Simulation and Synthesis", in *Proc*. 1st Workshop on Model Based Engineering for Embedded Systems Design (M-BED), Dresden, Germany,Mar. 2010.

[3] Mauro Prevostini, Elena Zamsa. "SysML Profile for SoC Design and SystemC Transformation", ALaRI, Faculty of Informatics University of Lugano via G. Buffi 13, CH-6904 Lugano, May 11, 2007

[4] Tomás G. Moreira, Marco A. Wehrmeister, Carlos E. Pereira, Jean-François Pétin, and Eric Levrat. "Generating VHDL Source Code from UML Models of Embedded Systems",IFIP Advances in Information and Communication Technology Volume 329, pp 125-136. 2010.

[5] Kim Sandström and Ian Oliver. "A UML Profile for Asynchronous Hardware Design", Lecture Notes in Computer Science Volume 4017, pp 15-26. 2006.

[6] N. Shimizu, M. Ikura, W. Wiriya, and S. Chivapreecha, "A new logic circuit design methodology with uml", in *Proc*. ITC-CSCC 2009, pp.62-65.

[7] Lun Li, Frank P Coyle, and Mitchell A Thornton. "UML to systemverilog synthesis for embedded system models with support for assertion generation". In Proc. ECSI Forum on Design Languages 2007, paper 10 on CD-ROM.

[8] Aulagnier D., Koudri A., Lecomte S., Soulard P., Champeau J., Vidal G., Perrouin G., and Leray P. "Soc/sopc development using mdd and marte profile". In Model Driven Engineering for Distributed Real-time Embedded Systems. Hermes, 2009.

[9] Tim Schattkowsky, Jan Hendrik Hausmann, and Gregor Engels. "Using UML activities for System-on-Chip Design and Synthesis", O. Nierstrasz et al. (Eds.): MoDELS 2006, LNCS 4199, pp. 737 – 752, 2006.

[10] Salah MERNIZ. " Méthodologie de Vérification Formelle Pour les Microarchitectures RISC Approche Fonctionnelle", Phd. Thesis, Computer science Department, Constantine 2 University, Constantine Algérie, 2008.

[11] R.Plasmeijer, M.V. Eekelen, "Functional programming and parallel graph rewriting". Addiso Wesley 1993.

[12] The OMG website. [Online]. Available: http://www.omgsysml.org/.

[13] The Clean website. [Online]. Available: http://wiki.clean.cs.ru.nl/Clean.

[14] Systems Modeling Language (SysML) Specification. OMG document: ad/2006-03-08-01, version 1. Draft, April 2006.

[15] Timothy R. Fisher. LE GUIDE DE SURVIE Java L'ESSENTIEL DU CODE ET DES COMMANDES. ISBN : 978-2-7440-4004-7. Copyright© CampusPress, 2009.

[16] Rinus Plasmeijer, Marko van Eekelen, "Clean language repport ", Version 2.1, November 2002.

[17] The Altova UModel website. [Online]. Available: http://www.altova.com/umodel.html.

[18] UML Profile for System on a Chip (SOC). OMG Available Specification, version 1.0.1 formal /06-08-01, August 2006.