Abstracting HW communications with channels for HDLRuby

Lovic Gauthier^{a,*}, Yōhei Ishikawa^a, Ryōta Sakai^a

Information System Course, National Institute of Technology, Ariake College
Higashi-Hagio 150, Omuta, Fukuoka 836-8585 (JAPAN)

*Corresponding Author: lovic@ariake-nct.ac.jp

Abstract

HDLRuby is an extensible hardware description language (HDL) created for improving the productivity of hardware (HW) designers. This paper presents an extension of the language with channels that decouple computation from communication and allow reusing components without modifying their HW description. As experiments, three data transmission protocols have been implemented as channels and seamlessly interchanged for the communication of HW components.

Keywords: Hardware Description Language, Transaction Level Model, IC communication.

1. Introduction

Current integrated circuits (IC) are often including multiple components that communicate with each other through complex protocols. The design of such devices is time-consuming, and a change in one component often requires redesigning the communication protocols as well as the other components interacting with it.

This paper proposes the concept of channel for decoupling the computation from the communication of HW components while limiting as much as possible the HW overhead. These channels are implemented as an extension of the HDLRuby^(1,2) hardware description language (HDL) and provide two high-level primitives for communicating, `read` and `write`, that can be used seamlessly within a HW process. At synthesis time, the HW corresponding to these primitives is blended into the target HW code to make it perform the communication through the channel. While such primitives have been used in several high-level design approaches, this is the first time to our knowledge that no high-level synthesis is required, nor any assumption is made for the implemented transmission protocol.

As experiments, the design of three channels for data transmission comprising a register, a handshake and a queue-based protocols are presented and used for the communication between a producer and a consumer HW component with several synchronization scenarios.

The rest of the paper is organized as follows: section 2 presents some related works, section 3 gives a quick introduction to the HDLRuby language and section 4 details the model, the usage and the implementation of the channels. Then section 5 describes the experiments before section 6 that concludes the paper.

2. Related works

Register transfer level (RTL) synthesis⁽³⁾, behavioral synthesis⁽⁴⁾ and high-level synthesis⁽⁵⁾ are commonly used methods for designing logic circuits. However, even with high-level synthesis, integrating multiple components is often tedious and time consuming. For tackling this problem, some approaches isolate inter-component communications through wrappers⁽⁶⁾ while other approaches are based on high-level communication protocols like the transaction-level model⁽⁷⁾ (TLM).

In details, the first kind of approaches consists in isolating each component from each other by adding HW components, called wrappers, that adapt the communication protocol of a specific component to the more general communication protocol used for the global IC. This approach allows to design each component independently, at its own pace while making the final integration easy. It makes it also easy to change a component or a communication protocol at any stage of the design by simply replacing the corresponding wrappers. However, this approach may suffer from the overhead of the additional HW brought by the wrapper. The second kind of approaches uses a global high-level description of the whole system,

```

1  system :counter { |max|
2    input :clk, :rst
3    [(max+1).width].output :q
4
5    par (clk.posedge) {
6      if (max+1).pow2? then
7        hif(rst | (q>max)) { q <= 0 }
8      else
9        hif(rst) { q <= 0 }
10     end
11     helse { q <= q+1 }
12   }
13 }

```

Fig. 1. HDLRuby code declaring a generic counter.

components and communications, and generates a low-level description of the global circuit. The high-level description can be standard high-level HDL like SystemC⁽⁸⁾, software code like C, or some tool-specific language like Parthenon⁽⁹⁾. For easy-to-change designs, high-level communication primitives are provided to the HW designer and advanced high-level synthesis algorithms are used to produce the resulting HW. However, efficient synthesis for higher level than RTL is still an open problem⁽¹⁰⁾.

The channels proposed in this paper are an enhancement of the first approach where the wrappers are replaced by the blending of the channels' description within the code of the HW component using the metaprogramming and reflection capability of HDLRuby⁽²⁾.

3. Presentation of the HDLRuby language

HDLRuby is an HW description language whose documentation and design tools' source code are available online⁽¹⁾. The goal is to increase the productivity of the HW designers by adapting to HW successful SW paradigms while ensuring the language remains synthesizable RTL. The main paradigms imported to HDLRuby are object-oriented programming, generic programming, reflection, and meta-programming. For that purpose, HDLRuby has been designed on top of the Ruby programming language⁽¹¹⁾. In term of syntax, HDLRuby includes all the syntactic constructs of the Ruby language with additional ones for handling HW-oriented descriptions. Synthesizability of the language is ensured by keeping an RTL model of computation, while the productivity-oriented features act only at the code description level.

Fig. 1 gives an example of HDLRuby code describing a generic counter. In the figure, the first line declares the HW module named `counter` with `max` as generic parameter. This parameter is the maximum value of the counter. Line 2 declares the clock and the reset of the counter (resp., `clk`

Fig. 2. The HDLRuby simulation and synthesis flows.

and `rst`). Line 3 declares the output signal (`q`). Its type is a bit vector whose width is given by generic parameter `max`. Lines 5-12 describe the process controlling the counter: this process is using non-blocking assignments as indicated by the keyword `par` (in case of blocking assignments the keyword `seq` is to be used instead) and is activated on rising edges of signal `clk`. Lines 6-11 use the hardware if control statements (`hif-helsif-helse`) to decide whether signal `q` is set to zero or incremented. There are actually two possible `hif` statements: the first one is for the case when maximum value `max` added to one will become zero due to an overflow, and the second one is for the other cases where the counter is to be set explicitly to zero when `max` is exceeded. The statement to use is selected during the synthesis with the Ruby language `if-else` statement.

This example also illustrates a few syntax specificities of HDLRuby compared to usual HDL: first, HDLRuby being fully object-oriented, its elements are all objects and are handled using methods, e.g., the `input` method of a type object for declaring a signal like in lines 2-3. Second, new objects are declared using a symbol for indicating their name, i.e., a colon followed by a string, but are then referred directly without any colon. Third, the Ruby language is fully supported within HDLRuby but this SW code is executed at synthesis time. This last feature is used for generating HW or adding extensions to HDLRuby for example.

HDLRuby code can then be processed by the HDLRuby compiler for simulation, or generation of Verilog HDL or VHDL code that can then be synthesized or simulated by conventional RTL design tools. The generated RTL code is synthesizable and can be integrated with other HW modules. Fig. 2 summarizes the possible simulation and synthesis flows from HDLRuby descriptions. More precisely, each succession of arrows from top to bottom represents a possible flow.

```

1  system :module3 {
2    lockstep :ch0
3    queue([8],256).(:ch1)
4    bus([32],4).(:ch2)
5    p3 = ch2.branch(3)
6  }

```

Fig. 3. Instantiation of three channels and a branch.

```

1  system :prodcons {
2    input :clkA, :clkB, :clkC, :rst
3    [4].output :counter
4    [8].output :odata
5    [8].inner :idata
6
7    queue(bit[8],clkA,rst).(:ch)
8
9    # Producer
10   par(clkB.posedge) {
11     hif(rst) { idata <= 0 }
12     helse {
13       ch.write(idata) {
14         idata <= idata + 1
15       } } }
16
17   # Consumer
18   par(clkC.posedge) {
19     hif(rst) { counter <= 0 }
20     helse {
21       ch.read(odata) {
22         counter <= counter + 1
23       } } }
24 }
25

```

Fig. 4. Communication through an internal channel.

4. Channels in HDLRuby

4.1 The model of a channel

Channels model arbitrarily complex communications between or within HW components. A channel comprises two ports, one for reading and one for writing, whose access HW is instantiated using the respective `read` and `write` primitives. These primitives take as argument an HW process that is executed when the access completes. Such a process can be for example used for starting some computations depending on the access result. There is no other constraint on the primitive so that any protocol, synchronous or not, can be supported. Depending on the kind of channel, one or several additional arguments may be required. For example, the signal whose value is to send in case of a write, the signal where to get the value from in case of a read or an address.

Like any other HW component, a channel is to be instantiated for being used. This instantiation actually inserts in the module the HW code for implementing the internals of the channel. However, the channel can also be accessed by other modules provided they have a reference to it: the required extra signals and ports for accessing the external channel are automatically added to the module.

```

1  system :module4 { |ch|
2    input :clk
3    [8].output :res
4    output :ack
5
6    par(clk.posedge) {
7      ack <= 0
8      ch.read(res) { ack <= 1 }
9    }
10 }
11
12 system :moduleCh {
13   input :clk, :rst
14
15   inner :ack
16   [8].inner :count, :res
17
18   queue([8]).(:ch)
19
20   module4(ch).(:my_m4).(clk,res,ack)
21
22   ...
23 }

```

Fig. 5. Communication through an external channel.

Sometimes, several read or write ports must be supported (e.g., for the buses). For that purpose, it is possible to define branches to a channel. A branch is an additional channel having access to the internal of the current channel but having its own read and write ports.

4.2 Using channels in an HW design

A channel is instantiated like a common module by giving the name of its description and the name of the resulting instance. Fig. 3 shows the instantiation of three channels: the first one is a channel named `ch0` of type `lockstep` and the second one is a channel named `ch1` of type `queue` and the third one is of type `bus`. The second channel requires additional generic parameters for specifying the queue: its data type (8-bit vector) and its depth (256) – please notice that due to syntactic constraints of the Ruby language, a channel with generic arguments requires putting the name in parentheses. The third channel takes as argument the data type (32-bit vector) and the number of branches (4). Line 5 shows the access to branch 3 of the bus channel and its assigning to variable `p3`.

After a channel is instantiated, the HW for accessing it can be instantiated by using the respective `read` and `write` primitives. Depending on the kind of channel, these primitives may have different arguments. For instance, a channel dedicated to synchronization may only need the process to execute when the access completes, whereas a queue channel also takes as argument the signal holding the data to transmit. Fig. 4 shows an example for the latter channel, where a first process transmits data to a second one via queue channel `ch`. In the figure, the first process, lines 10-15, sends the value of signal `idata` to channel `ch`,

```

1  channel :register { |typ|
2      typ.inner :buffer
3
4      reader_input :buffer
5      writer_output :buffer
6
7      reader { |blk, target|
8          target <= buffer
9          blk.call if blk
10     }
11     writer { |blk, target|
12         buffer <= target
13         blk.call if blk
14     }
15 }

```

Fig. 6. Implementation of a register-based channel.

incrementing it when the write completes. The second process, lines 18-23, reads it and outputs it through signal `odata` increasing signal `counter` when the read completes.

Even if a channel is declared outside the current module, it can be accessed provided it is passed as generic argument of the module. For example, Fig. 5 shows module `module4` accessing external channel `ch` whose reference is passed as generic argument. In the figure, this channel is declared in module `moduleCh` line 18. Then, line 20, it is passed as generic argument to the instance of `module4` named `my_m4` with connections to signals `clk`, `res` and `ack`.

4.3 Describing a channel

The implementation of a channel is described like a common module, but without the usual input-output-inout interface. Instead, additional processes dedicated to implementing the read and write primitives are to be described. Fig. 6 shows the description of a channel implementing a register-based communication. The first line tells that the name of the described implementation is `register`, and that it has a generic argument named `typ`. Any HW code can be put within a channel description, including the declaration of inner signals. For instance, at line 2 of Fig. 6, the buffer used for storing the transmitted data is declared as an inner signal whose type is given by generic argument `typ`.

The access primitives are described by two specific processes, `reader` for the read access and `writer` for the write access. Both are declared as shown Fig. 6 at respective lines 7-10 and 11-14. As seen in the figure, these processes can have arguments (`blk` and `target`). They are the arguments that will be used by the read and write primitives as explained section 4.2. In the example, argument `target` is the source signal of the read access and the destination signal of the write access, and `blk` is the process to execute when the corresponding access completes. Since the channel

```

1      writer { |blk, target|
2          top_block.unshift {
3              buffer <= 0
4          }
5          buffer <= target
6          blk.call if blk
7      }

```

Fig. 7. Writer with a default processing

is a simple register, accesses complete immediately and therefore the `blk` processes are executed at once using method `call`. In order to support the case where no such execution is necessary, the postfix `if` used lines 9 and 13 checks if `blk` does exists before using it. These processes are to be blended into the code of the modules performing the accesses and therefore are physically outside the channel. Hence, the inner signals of the channel they can access must be explicitly stated using the following commands:

- `reader_input`, `reader_output`, `reader_inout`: for assigning signals to the reader.
- `writer_input`, `writer_output`, `writer_inout`: for assigning signals to the writer.

The suffix of the commands (`input`, `output`, `inout`) indicates the direction of the signal when used in an external module. For example, the code of Fig. 6 specifies that signal `buffer` will be used as input for the reader process and as output for the writer process.

For some channels, the access primitives may require additional processing outside their instance HW, like

```

1  # Before instantiation
2  system :producer {
3      input :clk
4      input :run
5      [4].inner :idata : 0
6
7      register([4]).(:ch)
8
9      par(clk.posedge) {
10         hif(run) {
11             ch.write(idata) {
12                 idata <= idata + 1
13             } }
14     }
15
16 # After instantiation
17 system :producerI {
18     input :clk
19     [4].inner :idata : 0
20
21     sub(:ch) {
22         [4].inner :buffer
23     }
24
25     par(clk.posedge) {
26         ch.buffer <= 0
27         hif(run) {
28             ch.buffer <= idata
29             idata <= idata + 1
30         }
31     }
32 }

```

Fig. 8. Channel code blending

```

1  brancher(:bit) {
2    reader_input :buffer
3    reader { |blk, idx, target|
4      target <= buffer[idx]
5      blk.call if blk
6    }
7  }

```

Fig. 9. Description of a branch

```

1  output :outb
2  [3].inner :idx : 0
3
4  register :ch(bit[8])
5  chb = ch.branch(:bit)
6
7  par(clk.posedge) {
8    outb <= chb.read(idx)
9  }

```

Fig. 10. Access through a branch

resetting or setting default values to signals. The reflection capability of HDLRuby allows this by giving access for modification to any component of an HW description. For example, Fig. 7 shows a modification of the writer of the register channel presented Fig. 6 where `buffer` is set by default to 0. This is done by inserting “`buffer <= 0`” at the beginning of the process using the access primitive (line 3). This process does not exist while describing the channel (it will appear when compiled, i.e., where the accessed primitives are used), but command `top_block` of line 2 gives access to the top process being compiled, i.e., the very process using the `write` primitive. At that time, the command `unshift` inserts at the beginning of the process the HW passed as argument. This way, the first thing the process does when activated is to set `buffer` to 0, whether the write access is triggered or not.

It is at the initial stage of synthesis, when the components are compiled, that the code of the channels is blended into the HW they have been declared into, as well as their access code is blended from the corresponding `read` and `write` primitives. Fig. 8 illustrates such a blending. Its upper code describes a module, named `producer`. It includes a channel of the register kind (as described in Fig. 6 and 7) and it writes to it the increasing value of counter `idata` when input `run` is one. The lower code of the figure is the resulting module after instantiation. The channel declaration and access primitives have been replaced by the corresponding code: lines 6-8 declare channel’s buffer within a namespace (using keyword `sub`) to avoid name collision, line 11 sets the default value of the buffer (this is the line of code that have been inserted at the top of the process with the `topblock.unshift` command), and lines 13-14 implements the write access.

Lastly, a branch of channel is described as shown Fig. 9. The figure gives the code that, when added to the description of the register channel, defines a branch for reading a single bit of the register channel. The first line declares the branch and tells its identifier is `:bit`. Then its content is described like any other channel. In the case of the figure, this branch only allows reading a single bit of `buffer`. Fig. 10 gives an example of using such a branch for reading successively each bit of the channel and outputting it through signal `outb` at each rising edge of `clk`. In the figure, the accessed bit position is given by signal `idx` that is declared and initialized to 0 at line 2.

5. Experiment

As experiment, channels have been implemented and used for transferring data between two components. The first component generates a series of values and transmits them through a channel and the second one gets periodically the values from the channel. Previous Fig. 4 gives a simplified version of this code. The three channels are a register-based protocol whose code have been given Fig. 6, a handshake whose code is given Fig. 11 and a queue whose portion of code is given Fig. 12. They can be interchanged by modifying the instantiation of the channel line 7 of Fig. 4. Code modification can be avoided by providing them as a generic argument as it is the case for the code used for the

```

1  channel(:handshake) { |typ|
2    typ.inner :data
3    inner :req, :ack
4
5    reader_input :ack, :data
6    reader_output :req
7
8    reader { |blk, target|
9      top_block.unshift { req <= 0 }
10     hif(ack == 0) { req <= 1 }
11     helsif(req) {
12       target <= data ; req <= 0
13       blk.call if blk
14     } }
15
16    writer_input :req
17    writer_output :ack, :data
18
19    writer { |blk, target|
20      top_block.unshift { ack <= 0 }
21      hif(req) {
22        hif(~ack) {
23          data <= target
24          blk.call if blk
25        }
26        ack <= 1
27      } }
28  }

```

Fig. 11. Description of a handshake channel.

```

1 channel(:queue) { |typ,depth,clk,rst|
2   typ[-depth].inner :buffer
3   [depth.width].inner :rptr, :wptr
4   inner :rsync, :rreq, :rack
5   typ.inner :rdt
6
7   par(clk.posedge) {
8     hif(rst) { rptr <= 0 }
9     helse {
10      hif(~rsync) {
11        hif(~rreq) { rack <= 0 }
12        hif(rreq&~rack&(rptr!=wptr)) {
13          rdt <= buffer[rptr]
14          rptr <= (rptr + 1) % depth
15          rack <= 1
16        } } } }
17
18   reader_output :rreq,:rptr,:rsync
19   reader_input :rdt,:rack,:wptr,:buffer
20
21   reader { |blk,target|
22     if(cur_behavior.on_event?(
23       clk.posedge,clk.negedge)) then
24       top_block.unshift {
25         rsync <= 1 ; rreq <= 0
26       }
27       hif(rptr != wptr) {
28         target <= buffer[rptr]
29         rptr <= (rptr + 1) % depth
30         blk.call if blk
31       }
32     else
33       top_block.unshift {
34         rsync <= 0 ; rreq <= 0
35       }
36       hif(~rack) { rreq <= 1 }
37       helsif(rreq) {
38         rreq <= 0 ; target <= rdt
39         blk.call if blk
40       }
41     end
42   } }

```

Fig. 12. Part of a queue channel description.

experiments. It can be downloaded from: https://github.com/civol/HDLRuby/blob/master/lib/HDLRuby/hdr_samples/with_multi_channels.rb

The register channel uses a simple register named `buffer` to transmit the data. The handshake uses in addition a request/acknowledge-based protocol for ensuring that no data is lost. For that purpose, in its code given Fig. 11 the read primitive sets signal `req` to one and does nothing else as long as signal `ack` is 0 (lines 10-11). Conversely, the write access is inactive as long a signal `req` is zero (line 21), and then sets signal `ack` to one after writing to data (lines 22-26). The queue channel uses its own clock for handling the queuing memory. Its code adapts to the clocks of the processes reading and writing it: if the clocks are identical, it does not need any synchronization whereas in the other cases it uses a protocol identical to the handshake channel. Fig. 12 gives the code for the reading part of the channel. The first line begins the channel description and indicates four generic arguments: the type of the data to transmit `typ`, the depth of the queue `depth`, and the clock and reset. The second line

Table 1. The configurations used for the experiments.

Name	Channel	Producer clock	Consumer clock
rg_23	register	clk2	clk3
hs_32	handshake	clk3	clk2
qu_222	queue (clk2)	clk2	clk2
qu_213	queue (clk2)	clk1	clk3

Table 2. The Verilog generation results.

Name	Channel	Verilog code number of lines
rg_23	register	44
hs_32	handshake	73
qu_222	queue (sync)	104
qu_213	queue (async)	110

declares a queuing memory of depth size (`-depth` is a shortcut for a range from 0 to `depth-1`). Lines 3-5 declares the other signals implementing the queue including, the read and write pointers in the memory, `rptr` and `wptr`, the handshake synchronization signals, `rreq` and `rack`, and the read register `rdt` (the signals regarding the write accesses are omitted). The additional signal `rsync` is a flag indicating if the queue is synchronized on the same clock as the process performing the read access. When `rsync` is 0, the process given lines 7-16 handles the read access to the queue: in case of read request (`rreq` is one), and if the queue is not full, it reads the memory, updates the read pointer and acknowledges the transmission by setting `rack` to one (lines 13-15). Otherwise, this process handles the reset only. Lines 21-42 give the code of the read primitive. It takes as argument `target`, the signal to put the read access result to and `blk`, the process to execute when a read completes. It is first determined lines 22-23 if this access is used with the same clock as the queue. This check is performed at synthesis time since the Ruby `if` conditional is used. Reflection command `cur_behavior` gives access to the process the read is used in and method `on_event?` tells if signal `clk` positive or negative edges are used. If this is the case, no synchronization is required and the code lines 24-31 is used: it fixes flag `rsync` to one, and the access to the queue is performed directly without handshake. Otherwise, code lines 32-40 is used. This code does a handshake and delegates the access to the process of lines 7-16.

RTL simulations have been performed using the HDLRuby simulator for the configurations given in table 1 and Fig. 13 gives the corresponding simulation results as time charts. The simulations show that even though the code of the producer and consumer was left untouched, for each synchronization and each channel configuration, the

(a) rg_23

(b) hs_23

(c) qu_222

(d) qu_213

Fig. 13. Simulation time charts.

resulting circuits include different signals and behave with timing and synchronization tailored to each case.

Then the corresponding Verilog code have been generated for each configuration using the HDLRuby tool set. The size of the resulting code is given table 2 (the test bench part of the code has been omitted). By comparison, the size of the full HDLRuby code of the producer and consumer modules is 32 lines long when omitting the test bench and the scripts switching the configuration. Hence, the generated code is significantly longer than the initial code. Also, the important variation in the resulting code sizes is because for each configuration only the necessary HW code is present and fully integrated, i.e., the register protocol handling for rg_23, the handshake handling for hs_32 the synchronous queue for qu_222 and the asynchronous queue for qu_213. It should be noticed though that for the queue cases, the process controlling the queue memory – whose behavior depends on the value of signal `rsync` – is actually identical for both

circuits. In the current implementation, it is assumed that the RTL synthesis tool will simplify it out using dead code elimination optimizations. The resulting Verilog code can be downloaded from:

https://github.com/civol/HDLRuby/tree/master/lib/HDLRuby/hdr_samples/WithMultiChannelExpVerilog

The experiments show the flexibility and the ease of use of the channels. However, the verification of the components through HW simulation is more tedious that could be expected. The reason is that the resulting code after channel instantiation included multiple new signals that are not visible at the design level and are hard to interpret without the code of the channel that has intentionally be hidden. One can argue that such a problem is also present with the state of the art TLM and wrapper-based approaches. Nevertheless, we plan to extend the simulation tool with more information about the channels.

6. Conclusions

The paper presented a new approach for implementing an abstraction of HW communications that is both flexible and efficient. It introduces the concept of channels that embed RTL code describing some data transfer protocol and blends it into the communicating components and where the primitives called `read`, for reading a channel, and `write`, for writing a channel, are used. As illustrated by the experiments various communication protocols can be implemented through channels and they can adapt themselves to different synchronizing conditions. Replacing a channel by another only requires changing the declaration of the used channel, and the code will be automatically updated accordingly.

Acknowledgment

This research is founded by the Japanese Grants-in-Aid for Scientific Research (KAKENHI) No. 18K11284.

References

- (1) Lovic Gauthier : “HDLRuby Source Code and Documentation”, <https://github.com/civol/HDLRuby>
- (2) Lovic Gauthier, and Yōhei Ishikawa : “HDLRuby, a new High Productivity Hardware Description Language”, Proc. of the 5th International Conference on Business and Industrial Research (ICBIR), pp. 215-220, 2018
- (3) Louis Hafer, and Alice C. Parker : “A Formal Method for the Specification, Analysis, and Design of Register-Transfer Level Digital Logic”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 2, No. 1, pp. 4-18, 1983
- (4) Pierre Paulin, and John P. Knight : “Force-directed scheduling for the behavioral synthesis of ASICs”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 8, No. 6, pp. 661-679, 1989
- (5) Philippe Coussy, and Adam Moraweic : “High-Level Synthesis: from Algorithm to Digital Circuit”, Springer Science & Business Media, 2008
- (6) Luca de Alfaro Thomas A. Henzinger : “Interface Theories for Component-Based Design”, Proceedings of the 2001 International Workshop on Embedded Software (EMSOFT), pp 148-165, 2001
- (7) A. Donlin, "Transaction level modeling: flows and use models," Proc. of the International Conference on Hardware/Software Codesign and System Synthesis (CODES + ISS), pp. 75-80, 2004
- (8) O.S. Initiative : “Functional Specification for SystemC 2.0”, <http://www.systemc.org>
- (9) R. Nomura, K. Oguri, Y. Nakamura, and A. Nagoya : "The Strategy for Reliable ASIC Design in PARTHENON," Proc. of the 4th Int. Forum on ASIC and Transducer Technology (ASICTT), pp. 7-12, 1991
- (10) G.Martin, and G.Smith, “High-levelsynthesis : Past, present, and future,” IEEE Design & Test of Computers, vol. 26, no. 4, pp. 18–25, 2009
- (11) Yukihiro Matsumoto, and Kiju Ishituka : “The Ruby Programming Language”, Addison Wesley, 2002