International Journal of
Soft Computing And Software Engineering (JSCSE)
e-ISSN: 2251-7545
Vol.2,No.6, 2012
Published online: Jun 25, 2012

DOI: 10.7321/jscse.v2.n6.3

Modeling of a Memory Interface Using Modeling Language

*Akitoshi Matsuda[1], Shinichi Baba[2]

[1]*Dept. of Automotive Science, Kyushu University, Japan*
[2]*Kyushu Embedded Forum, Japan*
[1]*matsuda_aki@slrc.kyushu-u.ac.jp,*[2]*shinichi.baba@nifty.com*

Abstract. **In recent years, modeling languages have been widely used for algorithm development and verification in embedded system design methodologies. Such languages allow behavior descriptions or structure descriptions to be expressed in a specification that is defined by a consistent set of designers. It is expected that modeling language-based designs can reduce development times without sacrificing quality. This paper presents a case study of the design of a memory interface algorithm for peripheral memory circuits using a modeling language. The results of the case study demonstrate that the number of lines of source code of the modeling language-based design flow has been reduced by 86% and 78% compared to a traditional register transfer language (RTL) and the C language, respectively.**

Keywords: *modeling language, algorithm development, lines of source code*

1. Introduction

The design of current large-scale and highly functional digital home appliance frequently suffers from serious issues such as deterioration in quality and increase in the time and cost of developing the embedded system algorithm. The use of modeling languages in the design methodology focuses efforts to improve efficiency on the development of increasingly complex hardware and algorithm design in system-level design [1]. This design methodology involves a model-based design methodology and a modeling language-based design methodology. In this paper, a model-based design (Figure 1) shows that arithmetic element models (from multipliers, adders, and multiplexers to large-scale features such as filters) are provided as a combination of hardware circuitry [2]. Figure 2 shows a sorting circuit, which is a type of processing algorithm that can be realized by a modeling language-based design. This figure helps us to understand the operation processes in algorithm development [3]. In addition, these two design methodologies make automatic HDL simulation and generation possible [4]. The adoption of a modeling language facilitates a better understanding of hardware configurations and architectures, compared to both the C language and a register transfer language (RTL).

Thus, the adoption of a modeling language for hardware development will reduce the manual effort required for algorithmic level design, implementation, and verification compared to traditional methods, which are used in manually coded RTL. Visualization features can then be added to the whole algorithm structure in the system design to further reduce development time and improve quality.

In order to apply an efficient design approach, the modeling language-based design is adopted above the signal processing hardware level at the stage of algorithm development. Modeling language-based design flow is realized from the algorithmic level of the development stage to the auto−generation of RTL. We adopt the method for efficient hardware design, simplifying the hardware trade-off analysis and visualizing the overall structure of the algorithm. In this paper, we report the results of a case study applying the modeling language approach to memory interface development.

International Journal of
Soft Computing And Software Engineering (JSCSE)
e-ISSN: 2251-7545
Vol.2,No.6, 2012
Published online: Jun 25, 2012
DOI: 10.7321/jscse.v2.n6.3

Our paper is structured as follows: the next Section 2 gives an overview of related work in both C-based design and modeling language design. The outline of a design methodology for a modeling language is described in Section 3. In the same section we also present the design description methodology for the implementation and simulation of the modeling language. Section 4 describes the set of experiments we conducted on diverse real world applications on the DDR2 memory controller. After that, Section 5 gives a conclusion. Finally, Section 6 gives a future work.

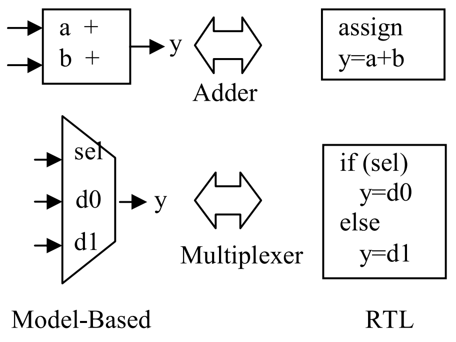

Figure 1. Arithmetic Element Models in Model-based Design

```
(* synthesize *)
module mkTb(Empty);
            Reg#(int) x <- mkReg(15);
            Reg#(int) y <- mkReg(6);
               rule swap ((x > y) && (y != 0));
                        x <= y;
                        y <= x;
               endrule
               rule subtract ((x <= y) && (y != 0));
                        y <= y - x;
               endrule
               rule fin (y == 0);
                        $display(x);
                        $finish(0);
               endrule
endmodule
```

Figure 2. Modeling Language Description Example

2. Related work

A variety of design automation techniques have been proposed to reduce the man-hours involved in the hardware design of embedded systems. The most general of these is a C language-based (C/C++, SystemC) design methodology to automatically convert to RTL by behavioral synthesis tools [5]. However, C-based languages are originally object oriented software that will be in the form of a sequential program description. In addition, for system-level design, C-based languages do not implement the operator and data structure needed for operation, so it is necessary to create the desired set of functions for planning through to implementation, which imposes additional costs [6].

For example, for behavioral synthesis from a C-based design, we need to use a Control Data Flow Graph (CDFG). In this case, some form of mapping is needed for a change from the description of the serial type format in the control flow graph to the description of the parallel type format. In contrast, using a modeling language, it is possible to use the description of the parallel type format because it is useful as a hardware model. In addition, it has been prepared in advance as a library module with storage or connectivity functions. Below, we list the main advantages of modeling language design compared to C-based design [7].

• Easy to build a test bench
• Simple concept of parallel programming
• Easy to describe interface between modules

In other words, in modeling language design, the simulation environment has been enhanced to include functional verification. Thus, if we take advantage of this design, we can accurately and quickly simulate a series design flow from algorithm development to hardware design using a single simulation tool. In addition, if we can automatically and reliably convert to RTL from a modeling language to implement the hardware from the algorithm level, while maintaining a high level of performance, then the designed hardware will exhibit high quality and high performance circuits.

3. Sorting algorithm using modeling language

We describe the outline of a design methodology for a modeling language. Specifically, using the modeling language, we have developed an algorithm (sort), which describes one type of numerical processing. Sorting is an operation in which a set of data is realigned, or sorted, using certain criteria. The main sorting algorithms are as follows.

```
(Module)
(* execution_order ="disp, fin" *)
(* preempts = "(swap_3, swap_2, swap_1, swap), fin" *)
(* synthesize *)
module mkBubSort (BubSort_IFC);
Vector#(5, Reg#(int)) x <- replicateM (mkReg(0));
Reg#(Bool) sorted <- mkDReg(False);

rule disp ;
$write("%2d : ", $time);
 for (Integer i=0; i<5; i=i+1)
 $write("x[%0d]=%2d, ", i, x[i]);
 $display("");
endrule

rule fin (x[0] != 0);
 sorted <= True;
endrule

for (Integer i=0; i<4; i=i+1) begin
 rule swap ((x[i] > x[i+1]));
  x[i] <= x[i+1];
  x[i+1] <= x[i];
 endrule
end

```

International Journal of
Soft Computing And Software Engineering (JSCSE)
e-ISSN: 2251-7545
Vol.2,No.6, 2012
Published online: Jun 25, 2012
DOI: 10.7321/jscse.v2.n6.3

```
(Module [Cont.])
method Action start(Vector#(5, int) a);
 writeVReg(x, a);
endmethod

method Vector#(5, int) result() if (sorted);
 return readVReg(x);
endmethod

endmodule
```

```
(Test bench)
import Vector::*;
import DReg::*;

(* synthesize *)
module mkTb(Empty);

let bsort <- mkBubSort();

int vals[5] = {23, 10, 5, 78, 16};
Reg#(int) cnt <- mkReg(0);

rule r1 (cnt == 0);
 bsort.start(arrayToVector(vals));
 cnt <= 1;
endrule

rule r2 (cnt == 1) ;
 $display("FINISHED");
 $write("%2d : ", $time);
 for (Integer i=0; i<5; i=i+1)
  $write("a[%0d]=%2d, ", i, bsort.result[i]);
  $display("");
 $finish(0);
endrule

endmodule
```

```
(Interface)
interface BubSort_IFC;
method Action start(Vector#(5, int) a);
method Vector#(5, int) result();
endinterface
```

Figure 3. **Sort Circuit for Modeling Language**

International Journal of
Soft Computing And Software Engineering (JSCSE)
e-ISSN: 2251-7545
Vol.2,No.6, 2012
Published online: Jun 25, 2012

DOI: 10.7321/jscse.v2.n6.3

1. Exchange sort
2. Selection sort
3. Insert sort
4. Heap sort
5. Shell sort
6. Quick sort

In this paper, we have chosen an exchange sort as a motif. This exchange sort is used in the process of sorting the five numbers of type “int” the algorithm. Figure 3 shows the description of the algorithm development for exchange sort by the modeling language. In this figure, it can be seen that the algorithm is composed of a sort circuit module (mkBubSort), a test bench module (mkTb), and a sort circuit interface (BubSort_IFC) [8].

There are two methods (“result” and “start”) in the sort circuit modules, at lines 26–28 and 30–32. The start method takes an argument of type Vector with five elements of type “int,” and assigns five registers (x[0]–x[4]) inside the module to each variable using the “writeVReg” function. These five registers are instantiated with an initial value of zero on line 5. The result method then outputs the value of the register to the outside when the sort is complete. The “sorted” condition of this method is the flag that indicates that the sort has completed. The flag is set when the rule “fin” has been executed on lines 15-17. Lines 19–24 also define four rules to perform the sorting. These rules execute a swap depending on the conditions under which adjacent values are compared in the registers. Rules generated by the “for” statement are named as swap, swap_1, swap_2, and swap_3. The pragma on line 2 dictates that, while one of the four rules to perform the swap is running, the “fin” rules still specify the criteria for exclusive execution so that it does not run. This pragma sets the “sorted” flag (i.e., sort is complete) when the rule of swap is not running and controls the execution condition of the “result” method.

A sort is started after passing a variable of type Vector to the “start” method in the sorting circuit at the rule “r1” of lines 46-49 in the test bench. When the value of “cnt” register is set to 1, the state moves to next, for which rule “r2” of lines 51-58 can be run at the same time. The rule “r2” condition is shown explicitly (cnt == 1), and it is not executed until the state for each action listed in the rule will be ready to run. In other words, the rule “r2” calls the “result” method of the sorting circuit module, the screen is displayed by the sorted result after sorting is complete, and the simulation is finished. Figure 4 shows the RTL code that is automatically generated from these modeling languages.

```
module mkBubSort(CLK, RST_N, start_a, EN_start,RDY_start, result,RDY_result);
        input  CLK, RST_N, EN_start;
        input  [159 : 0] start_a;
        output RDY_start, RDY_reset;
        output [159 : 0] result;
                    □
                    □
                    □
always@(negedge CLK)
  begin
      #0;
      if (RST_N)
        begin
          v__h663 = $time;
       #0;
          end
      if (RST_N) $write("%2d : ", v__h663);
      if (RST_N) $write("x[%0d]=%2d, ",
          $signed(32'd0), $signed(x));
      if (RST_N) $write("x[%0d]=%2d, ",
          $signed(32'd1), $signed(x_1));
      if (RST_N) $write("x[%0d]=%2d, ",
          $signed(32'd2), $signed(x_2));
      if (RST_N) $write("x[%0d]=%2d, ",
          $signed(32'd3), $signed(x_3));
      if (RST_N) $write("x[%0d]=%2d, ",
          $signed(32'd4), $signed(x_4));
      if (RST_N) $display("");
      end
endmodule
```

Figure 4. Auto Generation RTL by Modeling Language

4. Experiment results

In this section, we report a case study of applying the modeling language-based design to peripheral circuits with DDR2-SDRAM (DDR2) [9]. The modeling language description can describe the behavior model, and we are able to simulate it on Bluespec [10], which is a modeling language design tool. In this experimental process, a behavior of the specification is verified by the validity of the algorithm and executed by the RTL auto-generation. The RTL is automatically generated by the modeling language at the algorithm level. Finally, we compare the number of lines of code in the modeling language and the RTL and C++ language.

First, the DDR2 memory interface requires a bank address, column address, row address, and commands as input. In contrast to these inputs, they output the bank address, commands, and an address to the DDR2. Figure 5 shows an overview of these features. In addition, Figure 5 shows the list of input and output commands of the DDR2 memory interface.

Figure 6 represents the state transition, showing the relationship between the state and transition behavior of the DDR2 memory controller. Figure 6 represents the Finite State Machine (FSM) of Mealy type. The following represent the current state vector x, the input vector u, the next state vector y, and the output vector z:

$$x \in B^{i},\ \ u(k) \in B^{j},\ \ y(k) \in B^{i},\ \ z(k) \in B^{p}$$

(1)

where i, j, p represent the number of memory elements, external inputs, and external outputs, respectively. In addition, B = {-1, 0, 1} is a set of three Boolean signal values. However, -1, 0, 1, respectively, represent a logical 0, an undefined value, and a logical 1. k is the index of the input vector [11]. As shown in Figure 6, this FSM has 14 states and six external inputs of the condition of the transition state, which consist of the six input values of “Init,” “Refresh,” “Conflict,” “Wr,” “Rd” and “done.” The priority of these states is as follows: Refresh > Conflict > “Wr”/”Rd”; “Rd” and “Wr” are not to occur at the same time.

After completing the description of the modeling language, we could use it to perform simulations on the tool. In this process, the validity of the algorithms was verified. Once a result is obtained in accordance with the design specification, the next step is to automatically generate the RTL. Therefore, the RTL is automatically generated from the algorithm-level modeling language.

Next, we compared the number of lines of the modeling language with the number of lines of RTL, generated from the modeling language design tool. These results are shown in Table 1. We can see that the number of lines of code in the modeling language decreases by a factor of 7 compared with the RTL. The number of lines of code in the test-bench is reduced by more than a factor of 10. Traditional behavioral synthesis tools could compare the number of lines of code in the design, but could not compare the number of lines of the test-bench. These tools could not automatically generate the RTL for test-bench. Therefore, the ability to compare the test-bench is a notable feature. The test-bench description can also be defined easily by the “rule” description, similar to the design description in the case of the modeling language design.

We compared the number of lines of modeling language with the number of lines of C++ language, generated from the modeling language design tool. We can see that the number of lines of code in the modeling language decreases by a factor of 5 compared with the C++ language.

The man-hours of modeling language-based design and manual coding HDL design were compared. The modeling language-based design methodology, based on the system level of the modeling language, ran a simulation considering the hardware design. Thus, the test-benches of the HDL description were also generated automatically and simultaneously with the HDL generation, and the simulation verification times were reduced. These reduced design iterations will reduce the overall design man-hours. The design man-hours were reduced from 73 h to 48 h in this design case study, a reduction of approximately 34%.

International Journal of
Soft Computing And Software Engineering (JSCSE)
e-ISSN: 2251-7545
Vol.2,No.6, 2012
Published online: Jun 25, 2012

DOI: 10.7321/jscse.v2.n6.3

Input:
- Bank address (3bit)
- Column address (13bit)
- Row address (13bit)
- Command

Output:
- Bank address (3bit)
- Address (13bit)
- Command

Input command:
- Nop : No operation
- Init : Operation Instruction – Initial setup
- Refresh : Operation Instruction - Refresh setup
- Wr : Write instruction
- Rd : Read instruction

Output command:
- DDR2_LoadMode : Mode register set instruction
- DDR2_Refresh : Refresh instruction
- DDR2_Precharge : Pre-charge instruction
- DDR2_Active : Active instruction
- DDR2_Write : Write instruction
- DDR2_Read : Read instruction
- DDR2_Nop : No operation

Figure 5. Function Guideline of DDR2 Memory Interface

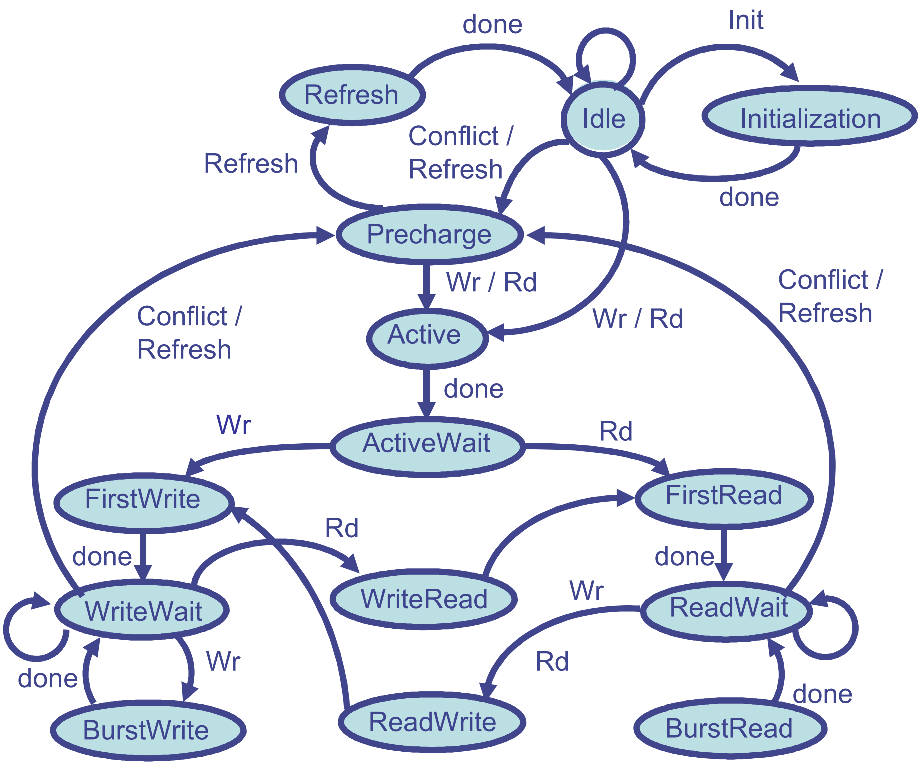

Figure 6. FSM of DDR2 Memory Controller

International Journal of
Soft Computing And Software Engineering (JSCSE)
e-ISSN: 2251-7545
Vol.2,No.6, 2012
Published online: Jun 25, 2012
DOI: 10.7321/jscse.v2.n6.3

Table 1. **Comparison of Lines of Source Code**

	Design	Test-bench
RTL	2929	434
C++	1852	N/A
Modeling	412	36

5. Conclusion

It was shown that we could consistently automate all processes, from algorithm verification to a system description and a detailed hardware design, using a modeling language-based design methodology and Bluespec. As a result, we found that it was possible to execute the system-level design quickly and with high quality. We believe that modeling language-based designs can be treated as block modules in a database, without too much dependence on a high-level synthesis tool. By adopting module visualization technology, we could prove that such algorithm modules will be accelerated by reusing old modules and properties.

6. Future work

Currently, we cannot adopt a modeling language-based approach for all hardware development. For example, advanced functions and/or complex control logic are not yet supported. We will continue to investigate such areas.

References

[1] W. Chen, X. Han, and R. Domer, "Out-of-order parallel simulation for ESL design," in Proc. of DATE'12, pp.141-146, 2012.
[2] M.D. Natale and H. Zeng, "Task implementation of synchronous finite state machines," in Proc. of DATE'12, pp.206-211, 2012.
[3] B. Safarinejadian, "Discrete event simulation and petri net modeling for reliability rnalysis," International Journal of Soft Computing and Software Engineering (JSCSE), Vol.2, No.5, pp.25-34, 2012.
[4] Y. Wang, P. Zhang, X. Cheng, and J. Cong, "An integrated and automated memory optimization flow for FPGA behavioral synthesis," in Proc. of ASP-DAC 2012, pp.257-262, 2012.
[5] G. Arnout, "C for System Level Design," in Proc. of DATE'99, pp.384-386, 1999.
[6] A. Ghosh, J. Kunkel, and S. Liao, "Hardware Synthesis from C/C++," in Proc. of DATE'99, pp.387-389, 1999.
[7] A. Yamada, K. Nishida, R. Sakurai, A. Kay, T. Nomura, and T. Kambe, "Hardware synthesis with the Bach system," in Proc. of IEEE ISCAS'99, Vol.VI, pp.366-369, 1999.
[8] A. Matsuda and M. Sugihara, "A case study of memory peripheral circuits using a modeling language design," in Pro. of ITC-CSCC2011, pp.786-787, 2011.
[9] M.D. Gomony, C. Weis, B. Akesson, N. Wehn, and K. Goossens, "DRAM selection and configuration for real-time mobile systems," in Proc. of DATE'12, pp.51-56, 2012.
[10] Bluespec Inc. Interra Systems' Benchmarking of Bluespec Compiler Uncovers No Compromises in Quality of Results (QoR), May 2004.
[11] S.S.S. Noori, S.A.S. Noori, and S.M.L. Baghal, "Optimization of routes in mobile ad hoc networks using artificial neural networks," International Journal of Soft Computing and Software Engineering (JSCSE), vol.2, No.4, pp.36-50, 2012.