Send Orders of Reprints at reprints@benthamscience.net

XML Specification for Complex Digital Logic Components

Fulong Chen*, Wen Zhou, Yunxiang Sun, Yonglong Luo

Department of Computer Science & Technology, Anhui Normal University, Wuhu, Anhui 241002, P. R. China

Abstract: Considering that the digital system design is a complex task, in order to meet some requirements such as effective performance, low costs and high reliability, and improve the development quality and efficiency, design activities need to be carried out under a precise description for completing the design cycle. This paper presents the XML specification method, and defines a subset of XML syntax for the complex digital logic component model. In this method, the logic mapping table is used to describe combinational logic components, state-oriented finite state machine to describe the behavior models of sequential logic components, and structure-oriented connector to describe structure models of composite hardware components.

Keywords: Complex digital logic components, XML, specification.

1. INTRODUCTION

In the development of digital logic systems, before we submit our design result for manufacturing, usually we have to face two key questions. The first one is how to specify what we want and the second one is how to make sure that what we specify is what we want. It has been widely estimated that over 70% of the design time for circuits is spent in performing various kinds of verification tasks and the effort devoted to this process eclipses all other aspects of the design process. Specification and verification have become the most important two tasks in current design flows and can have major impact on the timely delivery of a functionally correct product.

To get the final product, designers need to specify its functionalities and simulate its behaviors under the help of some simulation and verification tools. To save costly engineering effort, much of the effort of designing large logic machines needs to be automated.

Over the past two decades, with the rapid increase of complex digital logic systems' scale, the designers have leveraged technology scaling and rising power budgets to rapidly scale performance [1]. This fact showed that low level design methodologies would not be effective for designing complex digital logic systems in future, and new, efficient high abstract level design techniques need to be developed and adopted. These techniques should be used to model in different levels for designers. Hardware Description Languages (HDLs) can be used for model and synthesis with related tools. HDLs are based on the software programming languages and extend these languages with hardware signals, modules and interfaces.

*Address correspondence to this author at the Department of Computer Science and Technology, Anhui Normal University, 189 Jiuhua East Rd., Wuhu, Anhui Province 241002, P. R. China; Tel: (+86-553) 5910757; E-mail:long005@mail.ahnu.edu.cn

The function of HDL descriptions is to specify the structure and behavior of hardware. These specifications are mainly used to simulate the behaviors of description, synthesize lower level descriptions such as bit files and generate the real devices. Some descriptions in HDLs can be used for simulation, but cannot be synthesized into circuits.

1.1. Related Works

The dominant traditional HDLs are Verilog and VHDL (Very high speed integrated circuit HDL) which are both developed for simulation originally. Their grammar and style are similar to C. However, these HDLs have no enough system-level support for complex digital logic circuit design. They do not fully support the algorithm-level specification. Due to the low level of abstraction, designing Verilog /VHDL modules manually needs very skilled engineering and a significant time investment. Writing in HDLs can be more tedious and time consuming than writing a software program to do the same thing. After the design is complete, verification takes even more time. Currently, there are some expansions in traditional HDLs for enhancing their describing capability. Moreover, most of their description can only be used for simulation and cannot be synthesized [2].

In order to solve these problems, many late-model HDLs are developed. For example, Chisel can be embedded in Scala programming language and raises the level of hardware design abstraction [3]. MyHDL uses Python as a low level description language [4]. Genesis2 uses Perl to provide more flexible parameterization and elaboration of hardware blocks written in System Verilog [1]. The language called Verischemelog [5] provides Scheme syntax for specifying modules in a similar format to Verilog. JHDL (Java HDL) [6] regards Java classes as modules. HML (Hardware Meta Language) [7] uses standard ML (Meta Language) functions to connect circuits together. These approaches combine the poor abstraction facilities of the underlying HDLs with the high-level programming model that does not understand hardware constructs.

1874-4443/13

Therefore, directly programming in these HDLs is time consuming because they are in low abstraction level. Some descriptions in HDLs are only suitable for simulation purposes and cannot be synthesized into circuits. This means, we can simulate them, but we cannot get the real product.

XML (Extensible Markup Language) has been introduced as a standard of new generation network data expression, transmission and exchange by the W3C (World Wide Web Consortium), belongs to the definition of electronic document structure, describes the content of the international Standard language SGML (Standard Generalized Markup Language), and is a technology depending on the content of cross-platform on the Internet. It is a kind of simple data storage language and widely used in the computer system and other systems of data description [8]. Recently, XML has been implemented in simulator software called Ptolemy II. The primary persistent file format for Ptolemy II models is MoML (which stands for Modeling Markup Language), an XML schema language. In this simulator software, designers can use Vergil to graphically construct models. Vergil stores models in ASCII files using MoML [9].

1.2. Contributions

Our contribution is to introduce an efficient simple method to model embedded system quickly and make the specification and verification process practical for real designs. This paper presents the XML specification for these reasons:

(1) It has a simple and flexible text format;

(2) It is widely used for the representation of arbitrary data structures;

(3) The construct is similar to hardware (structure style).

This paper defines a subset of XML syntax for the component model. In this method, we regard circuits as components and every component has input ports for getting input signals and output ports for generating output signals. Big component contains small ones, basic gates: OR, AND, NOT as the atomic components. Logic mapping table is used to describe combinational logic components, state-oriented finite state machine is used to describe the behavioral models of sequential logic components, and structure-oriented connector is used to describe composite components.

2. COMPLEX DIGITAL LOGIC COMPONENTS

Complex digital logic circuit can be divided into combinational digital systems and sequential digital systems, corresponding to combinational logic components and sequential logic components which belong to atomic components, and namely cannot be divided into smaller components. A bigger component which is a compound logic component is constructed from those smaller components connecting with connectors (wires).

2.1. Combinational Logic Components

Combinational logic component is a kind of logic circuit in which the steady output is only related to the input variables at any time. The combinational logic component is made up of all kinds of logic circuit gate without memory components or feedback lines.

Definition 1: A combinational logic component is defined as

$$c =< IP,\ OP,\ T >$$

Where IP is a set of input ports, OP is a set of output ports, and T is a logic mapping table specified in ***LM*** (see Section 4).

2.2. Sequential Logic Components

Sequential logic system includes synchronous and asynchronous systems. The former changes all states when triggered by a clock signal. And the latter propagates changes whenever inputs change. Currently, only synchronous sequential systems are supported in our specification.

A Mealy logic component generates an output based on its current state and inputs. In contrast, the output of a Moore logic component depends only on its current state transitions without direct dependence on the inputs.

Definition 2: A sequential logic component is defined as

$$c =< IP,\ OP,\ M >$$

Where IP is a set of input ports, OP is a set of output ports, and M is a Mealy machine or Moore machine specified in ***MlM*** or ***MrM*** (see Section 5).

2.3. Compound Logic Components

A compound logic component is constructed from combinational logic components, sequential logic components and some smaller components connecting with connectors (wires).

Definition 3: A compound logic component is defined as

$$c =< IP,\ OP,\ C,\ L >$$

Where IP is a set of input ports, OP is a set of output ports, C is a set of sub-components specified in ***SCs***, and L is a set of connectors specified in ***Cns*** (see Section 6).

3. XML SPECIFICATION SUMMARIZATION

The XML description grammar of components is defined as Fig. (**1**). All logic components are specified in XML. It has the following characteristics:

(1) Each document statement includes four parts-a head for encoding declaration, included documents (not necessary), component definitions and comments.

(2) Each comment can start with <!-- and end with -->.

(3) The included documents are used to define if the current document includes other documents like the head files in C language.

(4) A document contains a series of component definitions specified in the marks <components> and </components>, and each component definition is a prototype component specified in the marks <component> and </component>. If the current document needs to use for instance prototype defined in other documents, the related document name should be in the mark <include> and </include>.

(5) Each component has a name, a list of input ports, a list of output ports and a body. Input ports are used for re-

Document	*D*	::=	*H Incls Cs* | *H Cs*
Head	*H*	::=	<?xml version=”1.0” Encoding =*Cd* standalone=O*pt*?>
Code	*Cd*	::=	“UTF-8” | “GB2312”
Option	*Opt*	::=	“yes” | “no”
Include Documents	*Incls*	::=	<includes> *Incl*$^+$ </includes>
Include	*Incl*	::=	<include> *F* </include>
Components	*Cs*	::=	<components> *C*$^+$ </components>
Component	*C*	::=	<component name=”*ID*”> *Ps B* </component>
Ports	*Ps*	::=	<ports>*P*$^+$</ports>
Type	*Tp*	::=	“bool”|”int”|”real”|”clock”
Port	*P*	::=	<input type=*Tp*> *ID* </input>
			| <output type=*Tp* > *ID* </output>
			| <inout type=*Tp* > *ID* </inout>
Body	*B*	::=	<body> *LM* </body>
			| <body> *MlM* </body>
			| <body> *MrM*</body>
			| <body> *SCs Cns* </body>
File Name	*F*	∈	String
Identifier	*ID*	∈	String

Fig. (1). Grammar for components.

ceiving input signals, output ports are used for generating output signals to other connectors. The port type includes bool, int, real and clock. The body specifies the component type-combinational logic component, sequential logic component or compound logic component.

4. LOGIC MAPPING

The first task of logic circuit design is to abstract logic relationships from the design requirements. For combinational components, the key work is to abstract logic relationships for descriptions.

In the combinational components, the logic mapping table is used to describe the logic relationships of the input variables and the output variables. Their XML description grammar is defined as Fig. (**2**). In the grammar, only **TRUE** (1), **FALSE** (0), **UNKNOWN**(*x*, *X*) and **HIGN-IMPEDANCE** (*z*, *Z*) are used to specify logic values.

Example 1: The following XML descriptions specify a full adder which is a combinational logic component as shown in Fig. (**3**) and its logic expressions are:

$$sum = a_{in} \oplus b_{in} \oplus c_{in}, c_{out} = (a \oplus b)c_{in} + ab .$$

```
<component name=” FullAdder”>
 <ports>
```

Logic Mapping	*LM*	::=	<logicmap {*ID*=”*E*”}* > *T*$^+$</logicmap>
Term	*T*	::=	<term {*ID*=”*E*”}> <in {*ID*=”*E*”}*> *ID*=”*E*”{;*ID*=”*E*”}*</in> <out {*ID*=”*E*”}*> *ID*=”*E*”{;*ID*=”*E*”}*</out> </term>
Value	*V*	::=	{0, 1, *x*, *X* , *z*, *Z*} ∪ Integer ∪ Real ∪ {*ID*}
Binary Operator	*BOp*	::=	{&, |, ^,+,-,*,/,%}
Unary Operator	*UOp*	::=	~
Expression	*E*	::=	*E BOp E* | *UOp E* | *V* | (*E*)

Fig. (2). Grammar for logic mapping tables.

```
  <input type=”bool”>ain</input>
  <input type=”bool”>bin</input>
  <input type=”bool”>cin</input>
  <output type=”bool”>sum</output>
  <output type=”bool”>cout</output>
 </ports>
 <body>
  <logicmap>
   <term>
    <in>a=”ain”;b=”bin”;
    c=”cout”</in>
    <out f1=”a^b^c” f2= “(a^b)&c|a&b”>
    sum=”f1”;cout=”f2”</out>
   </term>
  </logicmap>
 </body>
</component>
```

In the above descriptions, a_{in}, b_{in} and c_{in} are three input ports which will transmit three signals to generate the values of the signal f_1 and the signal f_2 according to their respective expressions, and *sum* and c_{out} are two output ports which will carry the values of the signal f_1 and the signal f_2. Obviously, this is a full adder.

Fig. (3). Full Adder Component.

5. STATE MACHINE

Finite State Machines (FSMs) or state tables are used to describe sequential logic components. Sequential logic com-

Mealy Machine	*MlM*	::=	<mealy {*ID*="*E*"}* Trigger="*Trig*"> *Init MlT*+ </mealy>
Moore Machine	*MrM*	::=	<moore {*ID*="*E*"}* Trigger="*Trig*"> *Init MrT*+ *MrO*+ </moore>
Initial State	*Init*	::=	<initial> State </initial>
Mealy Transition	*MlT*	::=	<transition {*ID*="*E*"}*> <current> *State* </current> <in {*ID*="*E*"}*> *ID*="*E*"{;*ID*="*E*"}*</in> <next> *State* <next><out {*ID*="*E*"}*> *ID*="*E*"{;*ID*="*E*"}* </out> </transition>
Moore Transition	*MrT*	::=	<transition {*ID*="*E*"}*> <current> *State* </current> <in {*ID*="*E*"}*> *ID*="*E*"{;*ID*="*E*"}*</in> <next> *State* <next></transition>
Moore Output	*MrO*	::=	<mooreoutput> <current> *State* </current> <out {*ID*="*E*"}*> *ID*="*E*"{;*ID*="*E*"}* </out> </mooreoutput>
State	*State*	∈	String
Trigger	*Trig*	::=	Negative | Positive | High | Low

Fig. (4). Grammar for Mealy and Moore FSMs.

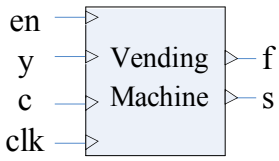

Fig. (5): Vending machine component.

ponents contain Mealy or Moore state machine, and each state machine has an initial state. The XML description grammar for FSM of sequential logic components is defined as Fig. (**4**).

Example 2: The following description fragment models for controlling the logic of a vending machine as shown in Fig. (**5**) used to charge \$1 (*y*=1) or 50 cents (*c*=1) so as to sell a piece of merchandise (*s*=1) or give changes (*f*=1), whose Mealy finite state machine is shown in Fig. (**6**).

```
<component name="VendingMachine">
 <ports>
  <input type="bool">en</input>
  <input type="bool">y</input>
  <input type="bool">c</input>
  <input type="clock">clk</input>
```

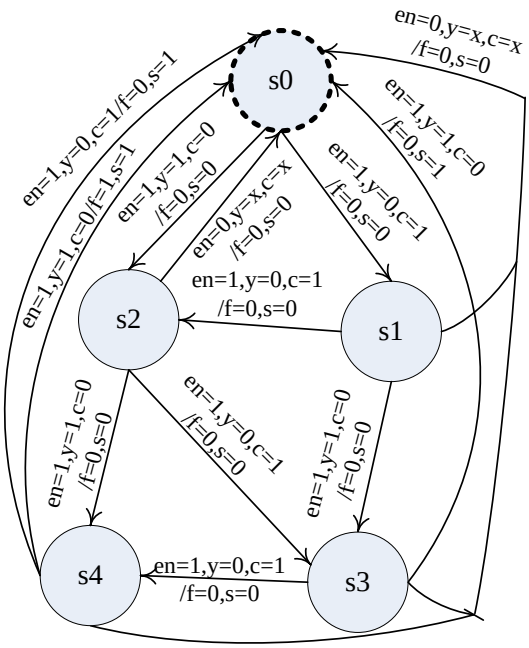

Fig. (6). State machine.

```
  <output type="bool">f</output>
  <output type="bool">s</output>
 </ports>
 <body>
  <mealy trigger="Positive">
  <!--finite state machine-->
   <initial>s0</initial>
   <transition>
    <current>s0</current>
    <in>en="0";y="0";c="1"</in>
    <next>s1</next>
    <out>f="0";s="0"</out>
   </transition>
   <transition>
    <current>s0</current>
    <in>en="1";y="1";c="0"</in>
    <next>s2</next>
    <out>f="0";s="0"</out>
   </transition>
   …<!-- part of the code omitted here -->
   <transition>
    <current>s4</current>
    <in>en="1";y="0";c="1"</in>
    <next>s0</next>
```

Sub Components	***SCs***	**::=**	**<subcomponents>** ***Inst*$^+$** **</subcomponents>**
Instance	*Inst*	::=	<instance name=" *ID*"> *ID* </instance>
Connectors	*Cns*	::=	<connectors> *Cn*$^+$ </connectors>
Connector	*Cn*	::=	<connector> {*ID.*}*ID* -> {*ID.*}*ID*</connector> | <connector> *V* -> {*ID.*}*ID*</connector>

Fig. (7). Grammar for compound components.

```
      <out>f="0";s="1"</out>
    </transition>
    <transition>
      <current>s4</current>
      <in>en="1";y="1";c="0"</in>
      <next>s0</next>
      <out>f="1";s="1"</out>
    </transition>
    <transition>
      <current>s4</current>
      <in>en="0";y="x";c="x"</in>
      <next>s0</next>
      <out>f="0";s="0"</out>
    </transition>
  </mealy>
 </body>
</component>
```

In this sequential logic component, there is a special input port- *clk* whose type is "clock" used to trigger the component to transform the input signal *en*, *y* and *c* into the output signal *f* and *s* according to its Mealy state machine. In Fig. (**6**), s_0 stands as the initial state. If *en*=1, *y*=0 and *c*=1, s_0 will be transferred to the state of s_1, and will generate the output signals of *f*=0 and *s*=0. When the state machine is in the state s_4, and captures the input signals of *en*=1, *y*=0 and *c*=1, it will migrate to the state s_0 and generate the output signals of *f*=0 and *s*=1 so as to sell a piece of merchandise. In any state, if *en*=0, the state machine will be transferred to the state s_0 regardless of *y* or *c*.

6. STRUCTURAL MODELLING

The structure of a compound logic component is hierarchical and composed of some smaller components, namely sub-components, connected with connectors. Fig. (**7**) shows its grammar.

Example 3: The following descriptions are used to specify a compound logic component which counts for the sale of goods in a vending machine as shown in Fig. (**8**).

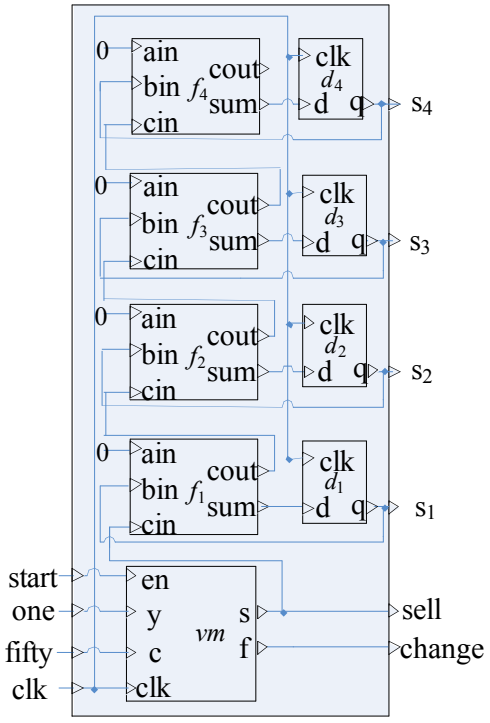

Fig. (8). Sale counter component.

```
<component name="SaleCounter">
 <ports>
  <input type="bool">start</input>
   <input type="bool">one</input>
   <input type="bool">fifty</input>
   <input type="clock">clk</intput>
   <output type="bool">s1</output>
   <output type="bool">s2</output>
   <output type="bool">s3</output>
   <output type="bool">s4</output>
   <output type="bool">sell</output>
   <output type="bool">
      change</output>
 </ports>
 <body>
   <subcomponents>
    <instance name="f1">
      FullAdder</instance>
    <instance name="f2">
      FullAdder</instance>
    <instance name="f3">
      FullAdder</instance>
    <instance name="f4">
      FullAdder</instance>
```

```
    <instance name="d1">
      DFlipFlop</instance>
    <instance name="d2">
      DFlipFlop</instance>
    <instance name="d3">
      DFlipFlop</instance>
    <instance name="d4">
      DFlipFlop</instance>
    <instance name="vm">
      VendingMachine</instance>
  </subcomponents>
<connectors>
    <connector>
      f3.cout->f4.cin</connector>
<connector>
      0->f4.ain</connector>
<connector>
      d4.q->f4.bin</connector>
    <connector>
      f2.cout->f3.cin</connector>
<connector>
      0->f3.ain</connector>
<connector>
      d3.q->f3.bin</connector>
    <connector>
      f1.cout->f2.cin</connector>
<connector>
      0->f2.ain</connector>
<connector>
      d2.q->f2.bin</connector>
    <connector>
      vm.s->f1.cin</connector>
<connector>
      0->f1.ain</connector>
<connector>
      d1.q->f1.bin</connector>
<connector>
      start->vm.en</connector>
<connector>
      one->vm.y</connector>
<connector>
      fifty->vm.c</connector>
<connector>
      clk->vm.clk</connector>
  <connector>
      vm.f->change</connector>
    <connector>
      vm.s->sell</connector>
    <connector>
      clk->d1.clk</connector>
    <connector>
      clk->d2.clk</connector>
    <connector>
      clk->d3.clk</connector>
    <connector>
      clk->d4.clk</connector>
    <connector>d1.q->s1</connector>
    <connector>d2.q->s2</connector>
    <connector>d3.q->s3</connector>
    <connector>d4.q->s4</connector>
  </connectors>
 </body>
</component>
```

In Fig. (**8**), the sale counter component is a compound logic component composed of 7 sub-components. Four FullAdder components –f_4, f_3, f_2 and f_1 constitute a 4-bit ripple adder which is used to add *vm.s* to $d_4.q$ $d_3.q$ $d_2.q$ $d_1.q$, four D flip-flop components –d_4, d_3, d_2 and d_1 constitute a 4-bit data register, and a vending machine component *vm* is used to receive the input signal *one* and *fifty* through the port *vm.y* and *vm.c* so as to generate the counting signal *vm.s*.

7. CONCLUSIONS AND FUTURE WORKS

To our best knowledge, this paper presents a practical way to describe circuit by providing specialized XML grammar. Our design will enable the quick development and data exchange of new IC (Integrated Circuit) product. Hierarchical structure helps to divide and conquer verification. After those jobs, the next step is to convert XML descriptions to low level description languages such as Verilog or VHDL so as to generate ASIC net or FPGA bit stream files under the help of the existing EDA synthesis tools. However, XML description is more lengthy and tedious. In order to build complex digital logic systems efficiently, it is necessary to model in a graphical development environment. Another important challenge is to verify the correctness and accuracy of the modeling components. Currently, a model named XModel GUI tool is being developed for all above aims. As noted in [10-15], more work needs to be done in the future.

CONFLICT OF INTEREST

The authors confirm that this article content has no conflicts of interest.

ACKNOWLEDGEMENT

The authors would like to thank the editors and anonymous reviewers for their valuable comments. This work is supported by the project of Anhui Provincial Natural Science Foundation under Grant No. 308085QF118, Anhui Normal University Innovation Fund (Grant No. 2011cxjj01), Anhui Normal University Scientific Research Foundation (Grant No. 132-151201).

REFERENCES

[1] O. Shacham, O. Azizi, M. Wachs, W. Qadeer, Z. Asgar, K. Kelley, J. P. Stevenson, S. Richardson, M. Horowitz, and B. Lee, "Rethinking digital design: Why design must change", *IEEE Micro* ,vol. 30, no. 6, pp. 9-24, 2010.

[2] F. Chen, Z. Zhu, and X. Fan, "XML Component-Based Modeling for Digital Circuits", In: Available at *2nd Pacific-Asia Conference on Circuits, Communications and System (PACCS)*, 2010, pp. 148-151.

[3] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic,"Chisel: Constructing hardware in a Scala embedded language", In : Available at *49th Design Automation Conference (DAC),* 2012, pp. 1212-1221.

[4] J. Villar, J. Juan, M. Bellido, J. Viejo, D. Guerrero, and J. Decaluwe, "Python as a hardware description language: A case study", In: *7th Southern Conference on Programmable Logic (SPL),* 2011, pp. 117-122.

[5] J. Jennings, and E. Beuscher, "Verischemelog: Verilog embedded in scheme", In: *2nd Conference on Domain Specific Languages*, 2000, pp. 123-134.

[6] P. Bellows, and B. Hutchings, "JHDL-an HDL for reconfigurable systems", In: *IEEE Symposium on FPGAs for Custom Computing Machines,* 1998, pp. 175-184.

[7] Y. Li, and M. Leeser, "HML, a novel hardware description language and its translation to VHDL", In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 8, pp. 1-8, 2000.

[8] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, H. Zheng, S. S. Bhattacharyya, E. Cheong, I. Davis, and M. Goel, "Heterogeneous concurrent modeling and design in java (volume 1: Introduction to ptolemy ii) ", DTIC Document, 2008.

[9] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng, "Heterogeneous Concurrent Modeling and Design in Java (Volume 1: Introduction to PtolemyII)", Available at: http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-28.html, Technical Report No.UCB/EECS-2008-28, April 1, 2008.

[10] Z. K. Huang, and K. W. Chau,"A New Image Thresholding Method Based on Gaussian Mixture Model", *Applied Mathematics and Computation*, vol. 205, no. 2, pp. 899-907, 2008.

[11] R. Taormina, K. W. Chau, R.Sethi, "Artificial Neural Network simulation of hourly groundwater levels in a coastal aquifer system of the Venice lagoon", *Engineering Applications of Artificial Intelligence*, vol. 25, no. 8, pp.1670-1676, 2012.

[12] C. L. Wu, K. W. Chau, and Y.S.Li,"Predicting monthly stream flow using data-driven models coupled with data-preprocessing techniques", *Water Resources Research*, vol. 45, W08432, 2009.

[13] J. Zhang, and K. W. Chau, "Multilayer Ensemble Pruning via Novel Multi-sub-swarm Particle Swarm Optimization", *Journal of Universal Computer Science*, vol.15, no. 4, pp. 840-858, 2009.

[14] C. Cheng, K. W.Chaw,Y. Sun, and J.Lin, "Long-term prediction of discharges in Manwan Reservoir using artificial neural network models", *Lecture Notes in Computer Science*, vol., 3498, pp.1040-1045, 2005.

[15] K. W. Chau, "Application of a PSO-based neural network in analysis of outcomes of construction claims", *Automation in Construction*, vol., 16, no. 5, pp.642-646, 2007.

Received: July 14, 2013 Revised: August 22, 2013 Accepted: August 26, 2013