Iterative Knowledge Based Code Generator for IEC 61499 Function Block

K. M. Goh
Singapore Institute of Manufacturing Technology
Singapore
kmgoh@SIMTech.a-star.edu.sg

W. Ding
Nanyang Technological University, Singapore
B. Tjahjono
Cranfield University
UK

Abstract—The IEC 61499 Function Block is an open standard for distributed control and automation. However, work related to using IEC61499 as code generator for various programming languages is still lacking. In this paper, a methodology for code generation based on XML and EBNF is discussed. Along with this code generator, an Iterative Knowledge Based Data Library (IKBDL) is proposed to improve the accuracy of the target codes. An example of converting XML codes generated from Function Block into SystemC language is demonstrated.

Keywords-IEC61499, Function Block, Code Generator, EBNF

I. INTRODUCTION

Embedded system development required various programming languages. Transforming codes from one language to another is a tedious work. The IEC61499 standard provides a modeling method that is target to encapsulate the low level source codes generation and make low level coding independent from design. In IEC61499 standard library elements definition [1], XML is chosen as one of the intermediate languages to represent the set of function blocks. As the growing use of XML, transformation between XML and other programming languages becomes necessary.

Chen et al [2] presents a tableau to give the corresponding between UML and SystemC. They are two specified languages’ transformation. It is time consuming to construct the tableau and to analysis each specified language’s grammar. Clark et al [3] employs a knowledge-based decision process and a generic mapping thesaurus to transform codes between languages. However, it is difficult to create the knowledge-based and to decide the search mechanism. CodeWorker [4] separates the source language and target language from the compiler which make the compiler independent. It will require proprietary script language to translate the source and target language. The advantage of this method is the accuracy of transformation, but the drawback user need to define and learn a new script language [5].

In this research, a new methodology aiming at code generation without additional script language. It will keep the configuration of the translation for reuse to improve the accuracy and automation of translation of the target language.

The structure of this paper is as follows: Section II gives the brief overview of IEC 61499 standard. Section III introduces the methodology based on XML and EBNF. Section IV presents the assumption of source XML. Section V discusses the mechanism of code generation. Section VI introduces the iterative knowledge based data library (IKBDL). Section VII gives an example on converting IEC61499 based XML document to SystemC for FPGA. Section VIII will conclude and discuss the future research.

II. IEC 61499 FUNCTION BLOCK STANDARD

IEC technical committee 65 (TC65) defines the IEC 61499 Function Block standard for distributed, programmable, dynamically configurable control. Function Blocks can be easily used for modeling a wide range of manufacturing problems and solutions ranging from small scale logic gates to large scale distributed systems. A Function Block is a ‘functional unit of software application’ that has its own internal data and algorithms. It allows functions to run on different resources so that distributed control systems can be constructed. Smaller elementary Function Blocks can be created and could be interconnected one another to form a composite Function Blocks for more complex problems. As shown in Figure 1, a Function Block consists of two parts: an algorithm part and an execution control part.

![Figure 1: IEC 61499 Function Block](image)

The top part of the block is the execution control, which is defined in terms of a state machine. Upon arrival of an event, the state machine will enter into a new state which will activate the algorithms within the block. Algorithms and internal data are hidden within the block which is represented in the lower part of the Function Block. A Function Block will have more than one algorithm sharing the same internal data. The Function Block will be mapped onto the actual resource, process interface and communication for deployments.

III. FRAMEWORK BASED ON XML AND EBNF

Iterative Knowledge Based Code Generator (IKBCG) consists of three parts: the XML document, the Extended...
Backus-Naur Form (EBNF) file and the Iterative Knowledge Base Data Library (IKBDL). The framework in Figure 2 shows the procedure of producing the desired code. Input data, the EBNF and XML documents are represented as expandable trees in a Graphic User Interface (GUI). IKBDL stores the operation on trees in an iterative way to improve the accuracy of target code.

**FIGURE 2: IKBCG FRAMEWORK**

**A. XML tree and function blocks**

XML is widely used as intermediate language. The hierarchy XML tree representation of data will match to the EBNF tree of the target language which makes the automatic code generation feasible.

In IEC61499, there are six kinds of function block types: FB Type, Device Type, Resource Type, System Type, Data Type and Adapter Type [6] corresponding to six kind of generated XML documents. In other words, the template for each function block type can be predefined. For each template, comparing with the customized XML document from IEC61499, the difference lies on the node’s attribute especially more nodes with the same name to be inserted. In the following piece of XML codes, more nodes named “VarDeclaration” can be inserted in the design and this name will not be changed. “VarDeclaration” is meaningless for the target codes’ generation. The valuable information lies in the nodes’ attributes like “name” and “type”.

```
<InterfaceList>
  <InputVars>
    <VarDeclaration Name="IN1" Type="BOOL" />
    <VarDeclaration Name="IN2" Type="BOOL" />
  </InputVars>
  <OutputVars>
    <VarDeclaration Name="OUT" Type="BOOL" Comment="Result" />
  </OutputVars>
</InterfaceList>
```

Therefore, the node’s attribute should be considered as a leaf node for further data extraction. In this paper, the XML tree’s leaf node is the attribute node. The XML tree is illustrated in Figure 3 below.

**B. EBNF tree construction**

Backus-Naur Form (BNF) and EBNF [7] is used as syntax definition of programming languages. EBNF is more convenient to express the syntax. It consists of four operators: ’|’ which indicates alternatives, ’[ ]’ stands for option, ‘{ }’ for repetition and ‘( )’ for grouping [8].

```
decl-begin ::= 'SC_MODULE' '(' module-name ')'
```

Basically, each language’s EBNF has a start symbol-the root of the EBNF Tree. Each production rule’s Left Hand Side (LHS) can be considered as a tree node and the Right Hand Side (RHS) can be thought as the children of this node. For the production rule:

```
Modifier ::= public | private | protected | static | final
```

The LHS “decl-begin” is a tree node and the RHS “SC_MODULE”, “(”, “module-name” and “)”) are its children.

IKBCG’s EBNF tree contains three kinds of leaf nodes: the language’s identifier node, the language’s keyword node and the EBNF keyword node. The leaf node (which is also the attribute node) in XML tree corresponds to the language’s identifier node in EBNF tree.

**C. Graphic User Interface**

Fully automatic language translation from XML to various other computer languages is not realistic [9]. IEC61499 will generate XML document and the target computer language EBNF definition can be found from the compiler suppliers. However, only these data is not enough to generate the desired code. EBNF only defines the syntax of the language but lack of semantic rules [9]. For example, production rules for Java’s modifier are defined as:

```
modifier ::= public | private | protected | static | final
```

The XML attribute corresponds to which modifier cannot be determined without additional semantic information. CodeWorker [4] uses script language to define additional semantic information of the target language. Comparing with script language, provide interactive GUI can simplify the user’s operations without using additional script languages. Figure 4 shows the IKBCG’s interactive GUI design framework.

**FIGURE 4: IKBCG WINDOWS**

IKBCG, can also specify this corresponding to achieve the same objective, which requires appropriate rules. The framework of code generator’s working process is shown in Figure 5. It is separated into two parts - the front part and the end part. In front part, intermediate XML (iXML) is formed by merging the two inputs, XML and EBNF. Then the iXML is used as input to the end part to produce the desired codes through an interpreter. This part is independent from the source XML and target language’s EBNF.

There are several benefits of the process. Firstly, it separates the whole process of code translation into two parts-the front part and end part. The front part merges EBNF and source XML together, unconcerned about what the target codes will looks like (or how to form the target codes). In this part, IKBCG forms by abstracting the useful information both in EBNF and XML. As a text based language, IKBCG is easy for users to read and understand. Secondly, the end part from
IKBCG to target language is independent from the EBNF and
source XML and only relates to IKBCG itself.

D. Basic Elements

Rule-based blocks (RBB) can be divided into two main
parts: the data type conversion part and the code generation
part both of which contains several basic tags and attributes
with fixed names, called basic elements. Code generation part
consists of several RBBs. A RBB can also be embedded in
another RBB. In this way, the hierarchy of the function blocks
can be represented. Each block has a header indicates the
assigned production rule for certain sub-structure or elements.
The body of RBB contains information on how to translate
such sub-structure or elements. Besides the basic elements,
there are also some tags and attributes exist in each RBB with
flexible name, name of which production rule being chosen. A
general structure of RBB file is shown in

E. Interpret Process

The whole interpreting process has two steps, the pre-
process and the parsing process. Pre-process makes
modifications on XML. Those modifications only refer to
change the structure of source XML and add the required
information to facilitate the parsing process. Parsing process
concerns how to generate the desired code. The whole
interpreting process is illustrated in Figure 7.

F. Pre-Process

Pre-process executes before the iXML being parsed. This
process is required mainly because the translation rule of data
types is unknown and the format of connections between
events, or ports, or states in source iXML does not enough
support IKBCG. In this section, four pre-treatment rules will be
given to facilitate the parsing process.

1) Pre-Process: Data Types

As mentioned before, different language has different data
type definition. Even though some data types have the same
meaning, the forms different languages express can be
difference.

Generally, there are two kinds of data types, one is those
exist both in IEC 61499 and target language, the other is those
exist either in IEC 61499 or target language. The latter one is
meaningless for language translation. Now we give the pre-
treatment rule for data types.

2) Pre-treatment Rule 1:

For the data types that exist both in IEC61499 and target
language, assume the data type in IEC61499 is A, the one with
the same meaning in target language is B. Insert the following
tag as the child of <DataType>.

<ReplacementRule Source="A" Destination="B" />

It means that during the translation process, any value
equals to “A” in the source XML will be translated into “B”.

Pre-Process: Ports Connecting To the Physical Devices

Any system or function block should indicate input, output
ports and its values.

3) Pre-treatment Rule 2:

Those ports and event, whose values are not assigned in
function blocks, will be considered as the output event
variables and output data variables and their tag names will be
changed into <Input>/<Output>.

Therefore, once the input port or output port are recognized,
a conversion will be taken. In the following example, “IN” is
recognized as the input, so conversion happens:

From:

<FB Name="switch_2" Type="IN_BOOL" x="650.0" y="1255.5555">
<Parameter Name="QI" Value="1" />
<Parameter Name="IN" />
</FB>

To:

<FB Name="switch_2" Type="IN_BOOL" x="650.0" y="1255.5555">
<Parameter Name="QI" Value="1" />
< Input Name="IN" Id ="in1" Type = "BOOL" />
</FB>

4) Pre-Process: Connections between Function Blocks:

<Connection> is a special tag in the XML definition of
IEC61499. It means an event connection, data connection or
adapter connection as defined in [1]. It is the only tag relates to
two function blocks as one connection from source to
destination. There are three kinds of connection:

Connection type 1: <Connection Source="S.a"
Destination="D.b" />, which means “a connection exists from
the variable a in function block S to the variable b in function
block D”. Both composite function block and system function block have this connection type.

Connection type 2: \(<\text{Connection Source}=“a” \text{Destination}=“b” />\), which means “a connection exists from the variable an in current composite function block to the variable b in its sub-function block D“. A will be considered as the input variable.

Connection type 3: \(<\text{Connection Source}=“a” \text{Destination}=“b” />\), which means “a connection exists from the variable a in function block S which belongs to current composite function block to the variable b in this composite function block”. b will be considered as the output variable.

Connection type 1 and 2 only exist in composite function block.

The expression of connections in IEC61499 is not well designed for code generation. The reason relates to the code generating process, and it will be discussed in section 5.5.

Here we give the Pre-treatment Rule 3:

5) Pre-treatment Rule 3:

a) For each tag in the form of \(<\text{Connection Source}=“a” \text{Destination}=“b” />\), add a child under the tag \(<\text{FB}>\) whose attribute “Name” is S:

\(<\text{VarDeclaration Name}=“a” \text{Type}=“T” \text{Id}=“X” \text{Dir}=“Source” />\)

b) In composite function block, for each tag in the form of \(<\text{Connection Source}=“a” \text{Destination}=“b” />\), add a child under the tag \(<\text{FB}>\) whose attribute “Name” is D:

\(<\text{VarDeclaration Name}=“a” \text{Type}=“T” \text{Id}=“X” \text{Dir}=“Destination” />\)

c) For each tag in the form of \(<\text{Connection Source}=“a” \text{Destination}=“b” />\), add an attribute of the child under the tag \(<\text{InputVars}>\) whose value of “Name” is a:

\(<\text{VarDeclaration Name}=“a” \text{Type}=“BOOL” \text{Id}=“X” \text{Dir}=“Source” />\)

The number of block name is not simply equal to the number of singal name. Figure out their relationship, that is, number of block name equals the sum of the value of attribute “Source” and “Destination” before “.” without reduplicate, number of singal name equals to the number of the value of attribute “Source” before “.” without reduplicate. Although this calculation can be done by \(<\text{ForEach}>\), besides that, assigning the correct singal_name into the port needs more work. Note that now there is only one channel, if the number is more than one, how the parser knows which channel corresponds to the correct port? And now only data connection emerges; event connection hasn’t been taken into account yet. Meanwhile port bindings in various languages may differ. All these evince \(<\text{Connection}>\) is not appropriated for parsing IKBCG. Since it has fixed format, only changing the structure of \(<\text{Connection}>\) but retaining the information as Pre-treatment Rule 3 does to facilitate the parsing process is more feasible.

V. CODE GENERATION

A. Parsing EBNF file

The main problem for EBNF parsing is the repetition. Repetition has the formats of EBNF syntax \{a\}, a{, a} and \{a, \}a respectively. The repeat times can be found by recognizing how many similar nodes there are for a specified EBNF node. The priority exists in this matching. When several similar nodes being found in the XML tree, EBNF node needs to be associated first is always the one outside the curly brackets. Take a{, a} for example, assume there is no similar XML node to be found which means there is only one node to match the corresponding EBNF node. If the first “a” is not assigned to this XML node, the result becomes “, a” while the expected result is “a”.

Searching similar nodes for specified XML node happens in repetition as identifying whether repeat EBNF nodes need to be added only occurs on the tree level which contains “(” and “)”. The value of this node’s corresponding EBNF node will be replaced but the other similar nodes’ corresponding to newly added EBNF nodes will not. The causation is that this EBNF

node itself cannot tell where its newly added EBNF nodes locate. Only the nodes on the level where “{ ” lies know that. One solution is appending a reference for this EBNF node, telling it where its “similar” nodes are. This is a recursive way starting from the nodes on the level which “{ ” lies to their children.

B. Similar attribute identification

When an attribute is assigned to the corresponding EBNF node, similar attribute can be recognized by searching the attribute node’s parent node. Figure 8 below illustrates the procedure,

If there is a specified node called “VarDeclaration”, and then any node named “VarDeclaration” will be regarded as the similar node. Here we introduce a concept named “similarity” which means how similar between two nodes. The usage will be discussed in the next section.

![Figure 8: Similar attribute identification](image)

VI. ITERATIVE KNOWLEDGE BASED DATA LIBRARY

Interactive GUI cannot guarantee the accuracy of the codes. The IKBDL is designed to makeup this drawback by adding the “experience” gradually into a knowledge base.

A. Similarity searching

Complex semantic rule cannot be handled automatically in the first. Take the previous piece of XML codes for example, the “IN1” and “IN2” is the input variable and “OUT” is the output variable. Different language will deal with them in different way. In SystemC, sensitive list should be like “sensitive <<IN1 <<IN2” while the first time using GUI will result “sensitive <<IN1 <<IN2 <<OUT”. The problem lies on how similar among these three nodes. On the level “VarDeclaration”, they seem to be the same. While forward to one more parent node, which is marked in dashed arrows in Figure 9 below, the difference appears. If forward for one more parent again, they are all convergence to “InterfaceList”. Until now, it is clear to see their similarity. “IN1” and “IN2” are the “most” similar – they will be handled in the same way while “OUT” is a “less” similar node which may be treated in a different way in certain place.

The algorithm to determine the nodes’ similarity is as follows:

1) Find those nodes whose parent’s parent node has the same name. Put them in a group and set their similarity to 0.
2) Look up one more parent node for each node and take each node itself as the reference. Compare this node itself with the rest nodes one by one. Those nodes which have the parents with the same name will stay their original similarity for this node. Those whose parents have different names will add their similarity by 1.

Repeat step 2 until all these nodes are convergence to the same parent.

B. Correspond similarity to EBNF

The similarity tells how “far” to look up in order to determine which node should be considered as the similar node. The similar node will be treated in the same way to meet the EBNF node. This is an iterative way. The first time when a new XML template is loaded, the parser will start from the attribute node’s first non-attribute parent to determine the similarity. In other words, it will only look up to the first non-attribute parent node. After user interactive configuration on the EBNF node, previous similar nodes group will be divided into several sub-groups. Forward searching one level by one level until these XML tree nodes also meet the same or the most similar grouping. Then the desired similarity is found. Next time when the same template is loaded, the parser will obey this similarity to determine the similar nodes for each node.

![Figure 9: Similarity searching in IKBDL](image)
data type definition in this language and correspond to those defined in IEC61499. Then Longest Substring matching (LSM) starts between EBNF data types’ nodes and IEC61499 data types’ nodes.

LSM only need to work from EBNF tree to IKBDL as the data types transfer from XML tree to the specified language. E.g. “int” in the EBNF tree can match “INT”, “SINT” and so on. It doesn’t matter what exactly it matches because this information already enough to know “INT” and “SINT” corresponds to “int” in this language.

The first time result may have some inaccurate matching. Modification is required but only needs once. Then the corresponding will store in IKBDL for further work. As different language has different data type definition, there may have several data types missed matching, this kind will need manual appending into IKBDL.

VII. IMPLEMENTATION IN SYSTEMC

In this Section, we use a XML document generated from an IEC 61499 development tool named Function Block Development Kit (FBDK) [11]. Figure 11 shows simple light switch examples with 2 switch inputs with 2 light outputs. The logic of control the light is as below.

Light 1 = Switch 1
Light 2 = Switch 1 AND ~Switch 2

The FBDK allows developer to test the system with graphical simulation as shown in Figure 12. The simulation shows the result of the applications and its operations. Once the application is tested correctly, then it can be converted to other embedded platform like FPGA.

Figure 13 demonstrated IKBCG code generation example. In this case, it shows the code generation from a Basic Function Block named “FB_AND.fbt” to SystemC codes. Developer could select the data type (Figure 13a), modify the name (Figure 13b) and simply generate the systemC code (Figure 13c). Figure 12 shows the result of configured code onto a XLINIX’s Spartan FPGA board. The result is the same as shown in the simulation on Figure 12.

VIII. CONCLUSION

IKBCG proposes a code generation principle combining with IKBDL. This is a generic way which can be used to generate various languages’ codes from XML document for IEC61499 standard. The core idea is using least human-computer interaction to produce highest accuracy codes. However, there are some issues needed to be solved. The Basic Function Block’s algorithm can be written in other languages mentioned in IEC61499. Those languages can be expressed as a diagram or a text. The diagram language can be auto-translated into XML so it can be handled in this methodology, while the texture languages, until now, have to design a special compiler for each of them. Whether there is a generic way to integrate those texture languages’ translator together will be put into further research.

REFERENCES