Title:	The XML Generator and DFIR Compiler for LabVIEW Communications
Major:	Electrical Information Science and Technology
Name:	Jiaxin Chen
Student ID:	12346007
Supervisor:	Professor Brent Nelson and Professor Dong Zhang

Abstract

The National Instruments provides an API to access the intermediate representation (IR) called DFIR in LabVIEW Communications, which bridge the gap between the diagram in LabVIEW and other programming language. In my thesis, I focus on printing out IR and designing compiler to parse the XML file as well as converting it into C code.

Firstly, I learn about how to design a diagram in LabVIEW Communications. Meanwhile, I grasp the organization and namespaces of DFIR, which enables me to activate the API to deploy DFIR in to my C# project.

Furthermore, I choose the useful information in DFIR to design IR and invoke the nodes' properties and methods in my C# project to output DFIR's representation as the XML file. And I also establish the corresponding XML schema to verify the validation our XML file.

Last but not least, I parse my XML file by the built-in XML DOM Parser in C# of Visual Studio. I build a list of classes to store the representation from the XML file. With the help of topologic sorting thoughts, I create the code converter, which can generate C code from the XML representation.

Accordingly, with the DFIR project and C# compiler, I converter the diagram in LabVIEW Communications into C Code by the bridge XML representation.

Keywords: LabVIEW Communications, DFIR, Intermediate Representation, XML Parser, Code Converter

Content

Chapter 1 The Introduction

1.1 Background and Significance

The LabVIEW Communications System Design Suite is developed by National Instruments Company. It provides the design environment which is compactly integrated with NI software based on radio (SDR) hardware for developing prototyping communications systems swiftly. And due to the applicability of LabVIEW Communications, the developers can apply it to develop processors, FPGAs, jump-start research LTE and some other device easily.

However, on account of the obvious graphical programming characteristics of LabVIEW Communications, it lacks the versatility to graft it in other integrated development environment. Consequently, National Instruments Company offers an API, which can allow users to access a read-only approximation of the LabVIEW intermediate representation of G and MRD files. The intermediate representation, called Data Flow Intermediate Representation (DFIR), provides us a graph-based view of a source GVI or GMRD.

Accordingly, we can utilize DFIR to build the bridge between the diagram in LabVIEW Communications and other text-based programming language. The key is to print out all the nodes and connections information in XML file. As a result, it can provide the developers who know nothing about LabVIEW Communications understanding LabVIEW Communications' diagram and compiling it into another programming language. More significantly, on the principle of the XML file and validated schema file we provide, anyone can write their own complier to convert LabVIEW Communications into another programming language they need. It increases the commonality of LabVIEW Communications' usage and the field of its application drastically.

1.2 Current Situation Internally and Abroad and Relevant Research

Currently the requirement of the converter between two source code is increasing tremendously. And the research just starts in China. The leading research production about converting one programming language into another programming language is dominated abroad nowadays. For example, Tangible Software Solutions have developed a series of the most accurate and reliable source code converters, such as Instant VB, Java to C++ Converter and so on.

Analogously and particularly, because C dominates the embedded market, the National Instruments also developed the tool named LabVIEW C Code Generator, which enable experts to develop algorithms graphically in LabVIEW and create the C code that represents the algorithm. It can ensure domain experts, who may not have any embedded or C programming experience, to apply design methodologies to implementation directly. Therefore, the LabVIEW C Code Generator bridges the gap between design and implementation, which means it can also help domain experts save the time to focus on designing algorithms and prototype without worrying about producing the domain-expert code. Accordingly, owing to the example of the LabVIEW C Code Generator, we can also develop the tool for converting graphical diagram in the LabVIEW Communications into C code, which is my research focuses on.

1.3 The Content of the Research

In my research, the main job is printing XML file of DFIR and writing compiler to process XML file and generate C code.

In the beginning, I learned about the programming basics of LabVIEW Communications and the configuration of VI as well as how to design their diagram in LabVIEW Communications to prepare for the XML file I need print out. Moreover, I can grasp the model and the namespaces of DFIR I'll use in my research, which eventually deploy it into my C# project and compiler.

Next, we design the detailed information we need in the Intermediate Representation and take advantage of DFIR to output all the nodes' representation as the XML file. And we create the XML schema to validate our XML file to make sure the validity of our representation.

Last but not least, I utilize the built-in XML DOM Parser in C# of Visual Studio to process the XML file. And then I create a list of classes to memorize the information of all the nodes in my C# compiler. I make the best of topologic sorting thoughts to establish the code converter and generate C code from the XML representation.

1.4 The Structure and Chapters of the Thesis

The thesis has five chapters totally. The arrangement of the thesis in following sequence:

Chapter 1: Introduce the background, significance and basic content of my research, as well as current research situation abroad.

Chapter 2: Introduce the usage of LabVIEW Communications and represent the introduction, the structure and the namespaces of DFIR.

Chapter 3: Output XML file about the graphical programming diagram in LabVIEW Communications by C# and design XML schema to validate the XML file.

Chapter 4: Construct the data representation of the XML file contents in memory by processing XML file using DOM. And compile the C# compiler to convert it into C code.

Chapter 5: Summary my research and predict its prospect in ulterior research.

Chapter 2 DFIR in LabVIEW Communications

In my research, I use the LabVIEW Communications to design the diagram. And with the help of DFIR API in the C# program, we can obtain the intermediate representation of all the nodes in the diagram.

2.1 The Usage of LabVIEW Communications

The LabVIEW Communications offers all of the tools, which can interact with hardware, store data, capture data and compile code to process data. The users can open and or create a document to edit their LabVIEW Communications project.

2.1.1 Source Code Documents in LabVIEW Communications

Because we need to process the data in LabVIEW Communications, we have to compile the code on the diagram of a source code document type. Accordingly, LabVIEW Communications provides users the following types of source code documents:

- VI Diagram: The basic diagram in LabVIEW Communications, which can execute G dataflow. And the VI diagram can run on the host device or an FPGA target.
- Multirate Diagram: It can process signals on streams of data and transform the code to call from a VI targeted to an FPGA by simulating on a PC or host.
- Clock-Driven Logic: It need be used as a sub CDL within a Clock-Driven Loop from a VI targeted to an FPGA. And it will compile the code within a single cycle of the clock that users appoint or the FPGA target clock.
- FPGA IP VI: It's a VI diagram which is specifically for creating and transferring the algorithms in the diagram to an FPGA target. The users can design their diagram in G dataflow, specify the clock rate and output any results they want if they assign the VI diagram as an FPGA IP VI.

2.1.2 The Configuration of a VI

In my research, I designate a VI diagram as an FPGA IP VI to analyze and process my data. In order to obtain the information of every node and their connections, we need design a valid diagram. The basic configuration of a document can be edited as Figure 2-1:

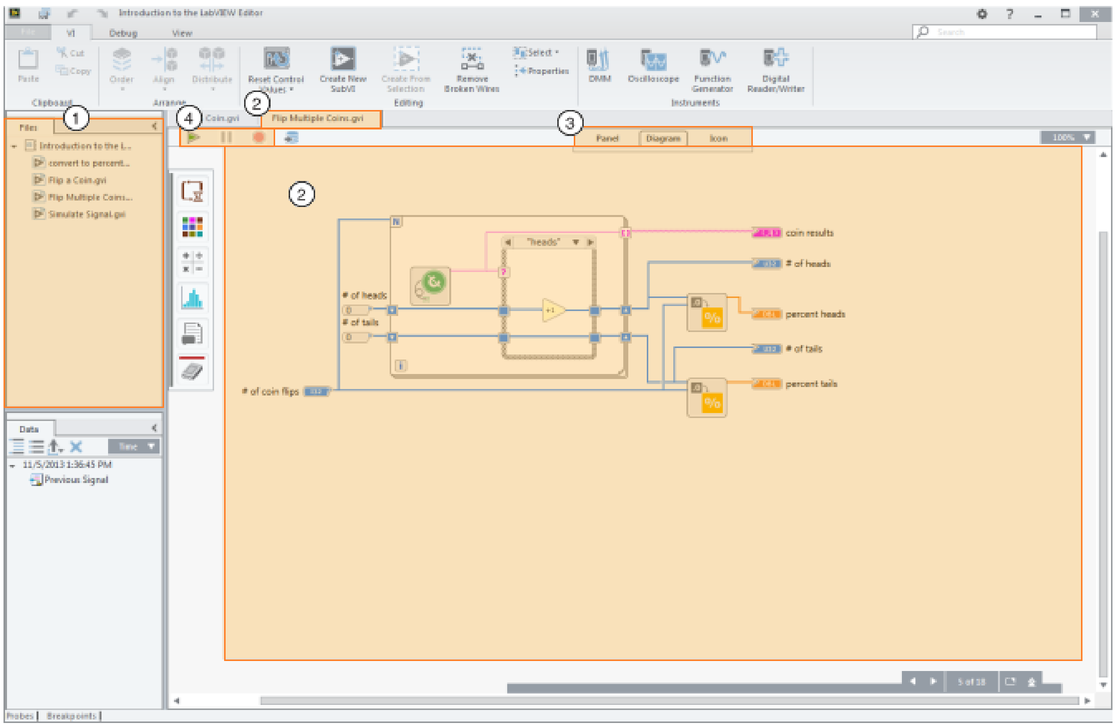


Figure 2-1 The Configuration of a VI

① Files pane: Open, create, and organize all the documents within a single project.

② Document: Access the associated content if you open one or more documents.

③ View selector: Access the different parts of a single document:

- Panel:

Panel Selector: Provide a spot for receiving the information for the users and indicating the inputs and outputs results of a VI, which enable to display the user interface of a VI in the workspace.

Controls and Indicators: Allow the users to either edit or view the data if they put them in the panel. Controls can both accept and display a single type of data, and the indicators can only display data. The data can be a number, a Boolean value, a text string, an array and so on.

Panel Palette: The hierarchical organization of all the controls and that you can add to the panel. If the users want to add an object from the palette to the panel, they can drag the object from the palette.

- Diagram:

Diagram Selector: Design the code in G dataflow, display the diagram and run it when the VI executes in the workspace.

Diagram Objects: Include all the nodes, wires, and other programming objects and combine them together to execute the code of the diagram.

Diagram Palette: The hierarchical organization of all the nodes you can add to the diagram. If the users want to add an object from the palette to the diagram, they can drag the object from the palette. The diagram palette is quite different from the palette that appears for the panel because the diagram palette displays the objects that must be useful in the diagram to create the code, which means some of them may not appear in the panel.

- Icon: Customize the VI to serve as a subVI in the workspace.

④ Run, pause, and abort buttons: Control the behavior of a VI diagram as running, pausing and aborting. And output the results of executed VI on the panel.

2.1.3 The FPGA IP VI Diagram Palette

My research is mainly based on the design of diagram part. As a matter of fact, due to the targeted FPGA IP VI I used, the nodes that are available for the diagram are less than the basic VI on a PC. Consequently, I need print out the information of nodes in FPGA IP VI Diagram Palette and its connections into XML file.

The nodes in FPGA IP VI Diagram Palette:

- Project Items: Source Code.
- Program Flow: For Loop, Case Structure, Select, Feedback Node and Comment.
- Numeric: Numeric Constant, Numeric Terminal, Complex, Dara Manipulation, Multiply, Decrement and some other numeric operating nodes.
- Boolean: Boolean Terminal, True Constant, False Constant, AND, OR, EXCLUSIVE OR and some other Boolean operating nodes.
- Array: 1D Array Constant, 1D Boolean Terminal, 1D Integer Terminal, Build Array, Array Size and some other array operating nodes.
- Cluster: Build Cluster, Cluster Properties and Cluster Constant.
- Comparison: Equal, Not Equal to 0, Greater than 0, Less than 0, Less or Equal to 0 and some other comparison operating nodes.
- Math: Add, Subtract, Divide, Increment, Absolute Value, Trigonometric and some other math operating nodes.

2.2 The Introduction of DFIR

Data Flow Intermediate Representation (DFIR) can provide users a graphical view of a source GVI or GMRD, which can represent all the VI constructs. Meanwhile, National Instruments offers their users an API, which equip the ability to read-only access to the LabVIEW Communications intermediate representation of VIs. The API is as a part of a special build of LabVIEW. Once the API is activated, the users can use it to build C# applications in Visual Studio platform. For a full understanding of API features, the users are encouraged to use Visual Studio with ReSharper to explore the API. For any class, the interface definition, inherited classes, and base classes can be viewed using the right-click context menu's Navigate section.

The establishment of DFIR C# application is in following steps:

- **Create A New C# Visual Studio Project**

When the users create a New Project, they can choose to establish the Class Library project as the project type according to "Installed - Templates - Visual C#". This operation will help the users create a C# .dll file, that can be connected with LabVIEW Communications actually.
And the users can customize the name and the solution name of the class by themselves. In addition, they can add multiple projects to the solution. In my research, I have built two C# projects: DotStringPrinter.cs and XMLGenerator.cs. Correspondingly, every project will match their own .dll library file. And each project can instantiate multiple buttons in the LabVIEW Communications' s UI.

- **Add A Class to the DFIR Project**

This class is used to represent one button in the LabVIEW GUI. The class is renamed to DotStringPrinter and XMLGenerator, a more sensible name, and then is refactored its name by ReSharper. Once this update has been done, other references in this solution will also be updated automatically.

- **Configure the DFIR Project to Export a LabVIEW GUI Button**

When the users need export a LabVIEW GUI Button, because the class must implement an object of *NationalInstruments.LabVIEW.DfirExport.IDfirRootAcceptor* interface, they need use the .NET Managed Extensibility Framework and some National Instruments components to achieve it.
Hence, the users need to add the following three lines on the top of the class definition in their C# project:

```
[Export(typeof(IDfirRootAcceptor))]
[PartMetadata("ExportIdentifier", "ProductLevel:Base")]
{PartCreationPolicy(CreationPolicy.NonShared)]
```

Furthermore, when the users need to find their class in the .dll file, they can add the following using directives to their class to make it possible for LabVIEW:

```
using NationalInstruments.LabVIEW.DfirExport;
using NationalInstruments.LabVIEW.DfirExport.Interfaces.IDfirBase;
using System.ComponentModel.Composition;
using System.Windows.Media.Imaging;
```

Eventually, the users need utilize the following sentences to replace the code in the

```
using NationalInstruments.Composition;
[assembly: ParticipatesInComposition]
```

"AssemblyInfo.cs" file in the Properties folder of the C# project:
Therefore, the users can edit the declaration of the class for the DotStringPrinter and XMLGenerator C# projects to make the IDfirRootAcceptor interface implemented:

```
public class DotStringPrinter : IDfirRootAcceptor
public class XmlGenerator : IDfirRootAcceptor
```

- **Add API Reference to the DFIR Project**

In the Solution Explorer window, the users can right click on Reference and Add Reference. In the browse section of Reference Manager, select the following two files:

NationalInstruments.LabVIEW.DfirExport.dll
NationalInstruments.Composition.dll files and select them.

- **Add Relevant System References to the DFIR Project**

Similarly, in the Framework, Assemblies of Reference Manager, the users need navigate and add the following system references:

System.CompositionModel.Composition
PresentationCore
WindowsBase
System.Xaml

- **Add a Name for the Button**

If the users want the button to display its name in the LabVIEW Communications GUI, they can make the *get* accessor of the *ButtonName* property return a string that contains the name. In my project, I designate the ButtonName returns *Print DotString* and *XMLGenerator*.

```
public string ButtonName
{
    get { return "Print DotString"; }
}
```

```
public string ButtonName
{
    get { return "XmlGenerator"; }
}
```

- **Add an Image for the Button**

The users can also add a .png image for their button in the LabVIEW Communications GUI. The image can be either the stock image or customized image. For the customized image, it should be 32x32 pixels without any transparency effects. In my research, I assign the DotStringPrinter and XMLGenerator C# projects to make the *get* accessor return null, which means that both of them use the default image.

```
public BitmapImage ButtonImage
{
    get
    {
        // Use the default image
        return null;
    }
}
```

- **Add the Button Action to the DFIR Project**

If the users want to add any action for LabVIEW to generate, they can add their code to the *AcceptDfirRoot* method. When the button is clicked in the LabVIEW Communications GUI, the code in the method will be executed and become the entry point of the C# application correspondingly. The class object will be instantiated when the VI is loaded by LabVIEW.
For my project, when I click the Print DotString Button, it will produce the .dot file to indicate the connections of the nodes in a graph. When I click the XMLGenerator Button, it will produce XML file to display the information of every node and their connections.

2.3 The Structure of DFIR

Top level abstract interface in the basic hierarchy of the IDfirRoot structure is the INode. And all of the other classes derive from the INode. The IDfirRoot organization is as following Figure 2-2:

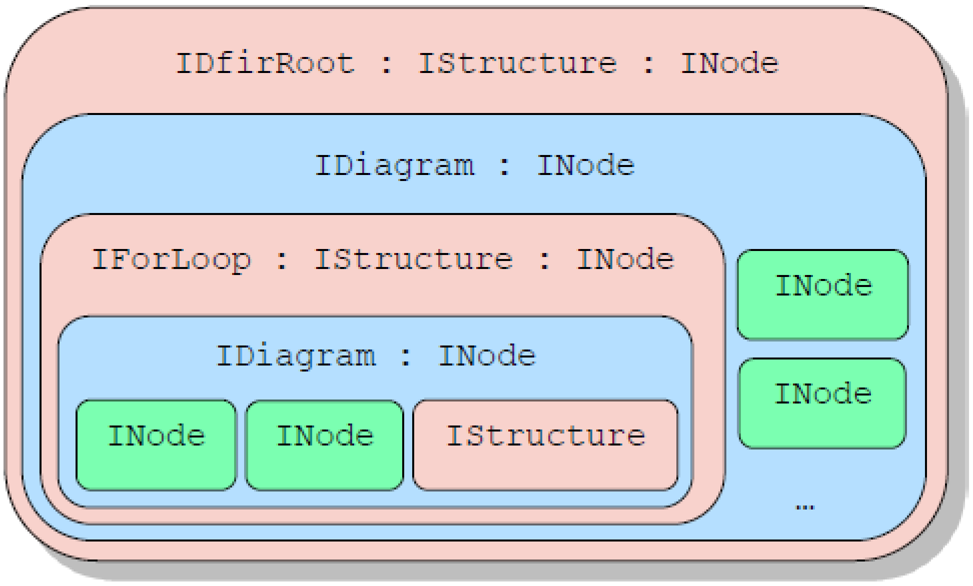


Figure 2-2 IDfirRoot Organization

The IDfirRoot is a special kind of IStructure, which contains all the INodes in the diagram. Besides, every GVI/GMRD has a single IDfirRoot. And an IStructure is an abstract interface that can have one or more IDiagrams and the IDfirRoot has only one IDiagram.

The IDiagram in an IStructure encapsulate a collection of the INodes. On account of the INodes in the IDiagrams that can implement the IStructure interface by themselves, it is possible for the DFIR to recursive the nesting IDiagrams. Take the IForLoop as an example. It will encapsulate the other nodes inside of the IForLoop, which enable DRIF to produce inside IDiagram that contains multiple nodes.

Moreover, the IStructure contains IBorderNodes besides IDiagrams. The IBorderNodes are nodes that are seen and have special functions on the border of a loop or case structure. The ICaseSelector node, ITunnel, IRightShiftRegister, ILeftShiftRegister, ILoopIndex node and ILoopMax node are some nodes that derive from IBorderNode.

In addition, the Wires are represented by the IWire node, which can connect two or more nodes together. Likewise, the IWire also derives from INode. However, considering all of the information in the IWire is existent in other nodes I will print out to the XML file, I omit outputting IWire in my C# project.

The ITerminal interface can display a terminal on an INode. On its left it's the source terminals, which is connected with the node's upstream node. And on its right it's the sink terminals, which is connected with the node's downstream node. ITerminal does not implement the INode interface.

2.4 The Namespace of DFIR

2.4.1 The NationalInstruments.LabVIEW.DfirExport Namespace

The IDfirRootAccepter interface is the only one for the NationalInstruments.LabVIEW.DfirExport namespace. Therefore, the users need program at least one object to implement the interface.

2.4.2 The NationalInstruments.LabVIEW.DfirExport.Interfaces Namespace

The NationalInstruments.LabVIEW.DfirExport.Interfaces namespace is the root namespace for all of the IDfir interfaces. In my project, I need use the IDfirBase Namespace, The IDfirBase.ITypes Namespace, the IProgramFlow Namespace, the IDataType Namespace and the IMath Namespace.

- **The IDfirBase Namespace**

The IDfirBase Namespace can display the basic blocks of a GVI. They include the interfaces, which can implement basic features of the IDfir graph. They are listed in following Table 2-1:

Table 2-1 The Interfaces in IDfirBase Namespace

Element	**Description**
INode	An element of the DFIR graph. This is the base abstract class from which all other nodes in the DFIR graph are created.
IDfirRoot	The root structure of a GVI/GMRD.
IDiagram	Holds a collection of INodes within an IStructure.
IStructure	Frames one or more IDiagrams. It has IBorderNodes that allow data to flow between the IDiagram outside of an IStructure and

	the IDiagrams within the IStructure.
ITerminal	A data connection on an INode. Is connected to an IWire.
IWire	A data connection between nodes. Is also represented as an INode.
IPrimitive	An operator on the graph. Provides a common template for many other nodes.
IBlackBoxNode	An unsupported node.
IConstant	A node with a constant value.
IDataAccessor	An input or output terminal node on a GVI’s DfirRoot, or a Data Port on a GMRD’s DfirRoot.
IBorderNode	An INode that exists on the border of an IStructure. It has input and output terminals that connect to INodes in the IDiagrams inside and outside the IStructure.

- **The IDfirBase.ITypes Namespace**

The IDfirBase.ITypes Namespace represents the data types exported by the DfirExport tool. Furthermore, the IDataType is the root datatype for all the other datatypes. They are listed in following Table 2-2:

Table 2-2 The Data Type in the IDfirBase.ITypes Namespaces

IArray	IBit	IBoolean
IComplex	IDataType	IDouble
IFixedSizeArray	IIncorrect	ISignedFixedPoint
ISignedInt	ISingle	IString
IUnknown	IUnsignedFixedPoint	IUnsignedInt
IUnsupported	IVariableSizedArray	IVoid

- **The IProgramFlow Namespace**

The IProgramFlow namespace includes the nodes from the Program Flow palette according to the location as well as the associated nodes with program flow nodes. They are listed in following Table 2-3:

Table 2-3 The Node in the IProgramFlow Namespace

ICaseSelector	ICaseSelectorRange	ICaseStructure	ICoercionNode
IDataRange	IFeedbackInitNode	IFeedbackInputNode	IFeedbackOutputNode
IFlatSequenceFrame	IForLoop	IFrame	ILeftShiftRegister
ILoop	ILoopCondition	ILoopIndex	ILoopMax
IMethodCall	IRightShiftRegister	ISelectPrimitive	ISiblingTunnel
ITickCountPrimitive	ITunnel	IWaitPrimitive	IWhileLoop

- **The IDataTypes Namespaces**

The IDataTypes namespaces represent nodes in the Data Type palette. They are listed in following Table 2-4:

Table 2-4 The IDataTypes Namespaces

Namespace	**Description**
IDataTypes.IArray	Array operations.
IDataTypes.IBoolean	Boolean operations.
IDataTypes.IComparison	Comparison operations.
IDataTypes.IString	String operations.
IDataTypes.IStreamManipulation	Stream manipulation operations in GMRDs.
IDataTypes.INumeric	Numeric operations.
IDataTypes.INumeric.IComplex	Complex math operations.
IDataTypes.INumeric.IConversion	Numeric conversion operations.
IDataTypes.INumeric.IDataManipulation	Numeric data manipulation operations.

- **The IMath Namespaces**

The IMath Namespaces represent nodes in the Math palette. They are listed in following Table 2-5:

Table 2-5 The IMath Namespaces

Namespace	**Description**
IMath.IExponential	Exponential operations on numeric data types.
IMath.ITrigonometric	Trigonometric operations on numeric data types.
IMath.IHyperbolicTrigonometry	Hyperbolic trigonometric operations on numeric data types.

2.5 The Summary of Chapter 2

In this chapter, I represent about the usage of LabVIEW Communications. The users can know the configuration of VI and how to design their diagram in LabVIEW Communications. Besides, I mentioned the nodes we need use later. In addition, I talk about the introduction, the model and the namespaces of DFIR I'll use in my research, which can equip us the ability to understand DFIR better and apply it into C# project and compiler better.

Chapter 3 The Intermediate Representation for DFIR

We build the two buttons in LabVIEW Communications to output the DOT file and XML file. The DOT file can provide us the visual representation while the XML file has the intermediate representation for the pending compiler. And the XML file is such an essential representation for the developers who need them to parse with their own compiler in a target programming language.

3.1 The Output of the DOT File

I use the “Print DotString” button in the LabVIEW Communications’ External Research Plug-Ins Ribbon, in order to observe how the GVI/GMRD maps to an DFIR. As I mentioned in the previous chapter, this button can generate the Dot file, which can provide the users the visual representation of the IDfir nodes generated. In addition, I use Graphviz to view the Dot file of a diagram.

The “DotStringPrinter” class extends the “IDfirRootAcceptor” class. The button parses the IDfirRoot and writes the dotstring to a file in the AcceptDfirRoot(IDfirRoot dfirRoot) function. The parameter name is “difrRoot” and the “IDfirRoot” is provided by the DfirExport tool. Besides, it uses writer.Write(dfirRoot.DotString) function to write the dotstring to the file.

The Dot file will print out the basic information of every node, such as NodeId, ParentId, TerminalId, TerminalIndex, Data Type and so on. Besides, the information of the wires that connect with each node will also be displayed.

For instance, in the basic “add.gvi”, the diagram in LabVIEW Communications is in the Figure 3-1:

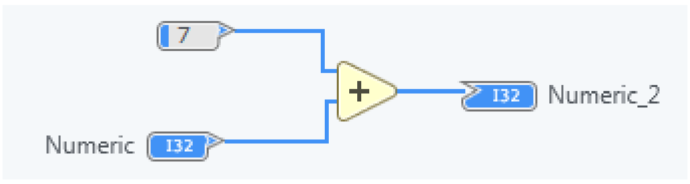


Figure 3-1 The “add.gvi” Diagram

When we press the “Print DotString” Button, we can print out the visual representation of the “add.gvi” diagram in Figure 3-2:

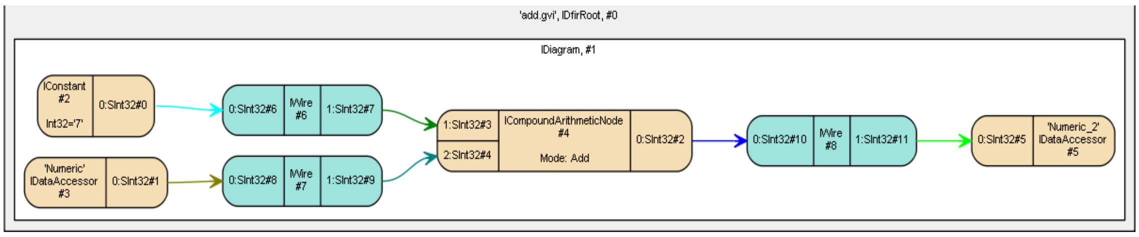


Figure 3-2 The “add.gvi” DOT file

And we can obtain the useful representation of every node from the above Dot file graph:

- IConstant:

  NodeId: 2 ParentId: 1 Value:7 DataType: ISignedInt

  TerminalId: 0 TerminalIndex: 0 TerminalDataType: ISignedInt
- IDataAccessor:

  NodeId: 3 ParentId: 1 Name: Numeric

  TerminalId: 1 TerminalIndex: 0 TerminalDataType: ISignedInt
- ICompoundArithmeticNode:

  NodeId: 4 ParentId: 1 Mode: Add

  InputTerminals: TerminalId: 3 TerminalIndex: 1 TerminalDataType: ISignedInt

  TerminalId: 4 TerminalIndex: 3

  TerminalDataType: ISignedInt

  OutputTerminals: TerminalId: 2 TerminalIndex: 0 TerminalDataType: ISignedInt
- IDataAccessor:

  NodeId: 5 ParentId: 1 Name: Numeric_2

  TerminalId: 5 TerminalIndex: 0 TermianlDataType: ISignedInt

Although the Dot file have IWire’s representation of IWire #6, IWire #7 and IWire #8, I just omit them because their information have the repetition with other nodes’ information. We can utilize the connection between the two nodes to match each other without IWire’s representation. As a result, we don’t have to use them in later XML output and compiler.

3.2 Print Out XML File as the Intermediate Representation

In order to output the intermediate representation for later compiler to parse it, we choose XML file to record the information of every node and their connection. When we press the “XMLGenerator” button in the External Research Plug-Ins of LabVIEW Communications, it will generate the XML file, which have the text-based representation of the IDfir nodes generated. The “XMLGenerator” class also extends from the “IDfirRootAcceptor” class. Similarly, the button parses the IDfirRoot and writes the XML to a file in the AcceptDfirRoot(IDfirRoot dfirRoot) function.

The IDfirRoot has a single IDiagram, called the IDfirRoot.BlockDiagram. In order to call the XML Generator with the top level block diagram, we use following statement:

```
IDiagram diag = dfirRoot.BlockDiagram;
```

And due to the XML prolog we need in the first line of an XML file, we print out it as following statement. To avoid errors, we should specify the encoding used, for example, to save the XML file as UTF-8:

```
writer.Write("<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n");
```

For the purpose of printing out the representation of every node, we call the function xg to output the various structures found recursively.

```
xg(0, writer, (dynamic)diag, "");
```

The “dynamic” means that the datatype of the diag can be IDiagram, IConstant, IDataAccessor and any other interfaces in DfirExport. And we utilized the properties and methods of every interface in the “National Instruments DFIR API Help.chm” to establish multiple xg function for all kinds of nodes. And in consideration of the readability and conveniences, we adopt the name of interface in DFIR directly as the name or the attribute of the element of the XML file.

- **XML Generator IDiagram:**

```
public void xg(int indnt, StreamWriter writer, IDiagram node, string msg)
```

The parameter indnt is used for indentation in XML file. And the writer declared by StreamWriter is for printing out representation into the XML file.

As the root element of the XML file, the IDiagram element has three attributes:

(1) NodeId: Output the unique own Id of current IDiagram by calling node.UniqueId.
(2) ParentId: Output the unique Id of current IDiagram’s parent node, and call the property node.ParentNode.UniqueId
(3) DiagramIndex: Print out the index of current IDiagram by calling node.Index.

For the representation of IDiagram in XML file, it looks like the following statement:

```
<IDiagram NodeId="1" ParentId="0" DiagramIndex="0">
```

Since a diagram contains a collection of nodes, even sub diagrams, we need GetAllNodes() to return all the nodes inside the diagram, not just the direct child node, and the use foreach to traverse them to call xg() function for every node to output their representation in XML file.

Furthermore, if we need the entire representation of every node, we need prepare for outputting their IDataType, connection and ITerminal firstly. And some of them need call their own particular representation function to print out some special properties.

- **Print IDataType**

```
public void PrintDataType(int indnt, StreamWriter writer, IDataType nodedatatype)
```

The IDataType is the root datatype for all the 18 datatypes in the IDfirBase.ITypes Namespace we mentioned in chapter 2.4.2. Generally speaking, the IDataType is the child element of ITerminal, which indicates the datatype of the terminal in the node. Especially, IConstant has the IDataType as its child element directly because it has its own value and needs to represent both its datatype and its terminal's datatype.

We utilize the IDataType as the element name and assign the specific datatype as the subelements inside of it. And in order to represent the specific datatype in detail, we declare the following PrintSubDataType() function to print the concrete information:

```
public void PrintSubDataType(int indnt, StreamWriter writer, IDataType nodedatatype)
```

According to the subelements of the specific datatype, we can divide the datatype into 5 parts to introduce their child elements:

(1) ISignedInt, IUnsignedInt
WordLength: Output the word length of the terminal by calling nodedatatype.WordLength.

(2) ISignedFixedPoint, IUnsignedFixedPoint
LeftLength: Represent the left length of the terminal by calling nodedatatype.LeftLength.
RightLength: Output the right length of the terminal by calling nodedatatype.RightLength.

(3) IArray, IVariableSizedArray, IFixedSizeArray
Dimensions: Print out the array's dimensions by calling nodedatatype.Dimensions.
Element: Indicate the datatype of elements in an array by invoking the PrintDataType() function to print it out.
ArraySize: Particularly for IFixedSizeArray to represent the array size in a fixed size array by calling nodedatatype.ArraySize[0].

(4) IComplex
Element: Indicate the datatype of elements in a complex by invoking the PrintDataType( ) function to print it out.

(5) IBit, IBoolean, IDouble, IIncorrect, ISingle, IString, IUnknown, IUnsupported, IVoid
These datatypes don't have special subelements.

- **Print Connections**

```
public void PrintConnections(int indnt, StreamWriter writer, ITerminal t)
```

In consideration of the direction of the terminal, we invoke the IsInput() and IsOutput() function to determine. If the terminal has input connection, call the GetImmediateSourceTerminal() function to obtain input ITerminal and print it out. If the terminal has output connections, call the GetSinkTerminals() to obtain a collection of output ITerminals and traverse them to represent.

The Connections element has the child element Connection, which has two attributes:
(1) TerminalId: Output the connected terminal Id of current ITerminal, call terminal.UniqueId
(2) NodeId: Represent the connected parent node Id of current ITerminal, call the property terminal.ParentNode.UniqueId

- **Print ITerminal**

```
public void PrintTerminals(int indnt, StreamWriter writer, ReadOnlyCollection<ITerminal> terminals)
```

The PrintTerminals() function output two element: InputTerminals and OutputTerminals. We use the terminal.Direction to distinguish them. They contain the list of relevant ITerminal as the subelements by calling the following PrintTerminal() function:

```
public void PrintTerminal(int indnt, StreamWriter writer, ITerminal terminal)
```

The ITerminal element itself has two attributes to print out:
(1) TerminalId: Output the terminal Id of current ITerminal, call the property terminal.UniqueId
(2) TerminalIndex: Represent the terminal index of current ITerminal by calling terminal.Index

After that it invokes the PrintDataType() and PrintConnections() function orderly to print out the datatype and connections of every ITerminal.

- **XML Generator INode**

```
public void xg(int indnt, StreamWriter writer, INode node, string msg)
```

As we mentioned before, INode is the root node interface for other nodes. The INode element contains two attributes, which ought to be represented by all of the other nodes for their attributes as well:

(1) NodeId: Print out the unique own Id of current INode by calling node.UniqueId.
(2) ParentId: Output the unique Id of current INode's parent node, and call the property node.ParentNode.UniqueId

And then it call the PrintTerminals() function to print out the representation of its terminals. Last but not least, each of the versions of xg() below we'll demonstrate handle extended types of INodes, which means that the default xg() for INode will be executed when there is not a specialized version.

- **XML Generator IDataAccessor**

```
public void xg(int indnt, StreamWriter writer, IDataAccessor node, string msg)
```

The xg() for IDataAccessor is executed with the purpose of representing controls and indicators in the diagram of LabVIEW Communications. Usually it's in possession of two attributes, which are NodeId and ParentId like in INode, as well as three subelements:

(1) Name: Output the name of current IDataAccessor node by calling node.Name.
(2) Direction: Represent the direction of current IDataAccessor node by establishing the PrintDirection( ) function and calling terminals[0].IsInput() and terminals[0].IsOutput() to judge the direction of IDataAccessor. And we stipulated OUTPUT for indicators and INPUT for controls.

```
public void PrintDirection(int indnt, StreamWriter writer, ReadOnlyCollection<ITerminal> terminals)
```

(3) InputTerminals or OutputTerminals: Output the ITerminal's representation by invoking PrintTerminals() function. And being dependent on the direction of controls for Input and indicators for Output, the IDataAccessor can only contain one of InputTerminals and OutputTerminals element content.

- **XML Generator IConstant**

```
public void xg(int indnt, StreamWriter writer, IConstant node, string msg)
```

The IConstant is used for expressing the constant value in a diagram. The representation of IConstant element is relatively similar with IDataAccessor element, which also has NodeId and ParentId as attributes and three subelements:

(1) Value: Print out the value of current IConstant node by calling node.Value.
(2) IDataType: Represent the datatype of IConstant's value by invoking the PrintDataType( ).
(3) OutputTerminals: Because the IConstant node can only output its value to other nodes, it doesn't have InputTerminals as its child element. Accordingly, we invoke PrintTerminals( ) function to obatint the ITerminal information of IConstant node.

- **XML Generator ICompoundArithmeticNode**

```
public void xg(int indnt, StreamWriter writer, ICompoundArithmeticNode node, string msg)
```

The ICompoundArithmeticNode is Add, Multiply, And, Or, Xor node in the diagram of LabVIEW Communications. The ICompoundArithmeticNode element shares the same attributes NodeId and ParentId with the INode element. Moreover, it contains the third attribute Mode as well as four element contents:

(1) Mode: Print out the mode of current ICompoundArithmeticNode by calling node.Mode.
(2) InvertedInputs: Owing to the inverted property of ICompoundArithmeticNode's every terminal as well as its mutiple input terminals, we declare the PrintInvertedInputs( ) function by calling node.InvertedInputs and traverse every input terminal to print out the list of the InvertedInput child element for every input termial:

```
public void PrintInvertedInputs(int indnt, StreamWriter writer, ICompoundArithmeticNode node)
```

(3) InvertedOutput: The ICompoundArithmeticNode has the exclusive output terminal, which determines the unicity of the inverted output. Hence, calling node.InvertedOutput to represent this element.
(4) InputTerminals and OutputTerminals: Every ICompoundArithmeticNode must have both InputTerminals and OutputTerminals as element contents. We can print out the representation of ITerminal element by invoking the PrintTerminals( ) function.

- **XML Generator IPrimitive**

```
public void xg(int indnt, StreamWriter writer, IPrimitive node, string msg)
```

IPrimitive includes most of operating nodes in DFIR, such as Subtract, Divide, Select, some Array's operating nodes, the nodes in IMath Namespaces and so on. To a certain exten, IPirmitive element is semblable to ICompoundArithmeticNode, which also has the third attrbute Mode apart from NodeId and ParentId and but quite simple element contents:

(1) Mode: Print out the mode of current IPrimitive node by calling node.Mode
(2) InputTerminals and OutTerminals: Output the representation of ITerminal element by invoking the PrintTerminals( ) function for both InputTerminals and OutputTerminals as requisite subelements.

- **XML Generator ITunnel**

```
public void xg(int indnt, StreamWriter writer, ITunnel node, string msg)
```

The tunnel node can point through which data enters or exits a structure. The ITunnel element has the two fixed attributes NodeId and ParentId and six subelements as usual:

(1) IsInput: Print out the direction of the ITunnel to judge if it's input or not by calling the method node.IsInput( ).
(2) GetInnerTerminal: Output the the terminal Id of the tunnel node which is in the inner diagram side. And we can obtain its attribute named TerminalId by calling the

method node.GetInnerTerminal( ).UniqueId.
(3) GetOuterTermianl: Output the terminal Id of the tunnel node which is in the outer diagram side. Similarly, we can receive its attribute named TerminalId by calling the method node.GetOuterTerminal( ).UniqueId.
(4) IndexingMode: The tunnel node can assume two functionalities:
Non-indexing tunnel: Pass the data through the loop border.
Auto-indexing: The auto-indexing input tunnel node can process one element of an array for each iteration of the loop. And the auto-indexing output tunnel node can append a piece of data from a single loop iteration to an accumulating arrat of data. Therefore, when we call node.IndexingMode to obtain the IndexingMode element's representation, Indexing in the representation stands for Auto-indexing and NonIndexing stands for Indexing.
(5) InputTerminals and OutputTerminals: Every ITunnel node may have both InputTerminals and OutputTerminals or only one of them as element contents, which depends on the connection of ITunnel on the diagram. We can print out the representation of ITerminal subelements by invoking the PrintTerminals( ) function.

- **XML Generator ILoopIndex**

```
public void xg(int indnt, StreamWriter writer, ILoopIndex node, string msg)
```

ILoopIndex indicates the iteration, namely the current loop iteration count, in a for loop or while loop. And the loop count always begins at 0 for the first iteration's executation. The ILoopIndex has the two attributes NodeId and ParentId and only one OutputTermianls as the child element:

OutputTerminals: The ILoopIndex owns only output terminals. So we can invoke the PrintTerminals( ) to output the representation of ITerminal subelements.

- **XML Generator ILoopMax**

```
public void xg(int indnt, StreamWriter writer, ILoopMax node, string msg)
```

ILoopMax is the Count for the number of times to execute the code inside the For loop or While loop. Likewise, the ILoopMax has NodeId and ParentId as its two attributes and both InputTerminals and OutputTermianls as the child elements:

InputTerminals and OutputTerminals: We can set the count to the ILoopMax's input terminal and get the number of times the loop executes for the output terminals. Accrodingly, it can print out the representation of ITerminal subelements for InputTerminals and OutputTerminals by invoking the PrintTerminals( ).

- **XML Generator ILeftShiftRegister**

```
public void xg(int indnt, StreamWriter writer, ILeftShiftRegister node, string msg)
```

The ILeftShiftRegister is on the left side of a loop, which can pass a value from current iteration of a loop to the next iteration through its a series of terminals. After the initial loop iteration, the left shift register in the pair returns the value it receives from the right shift register from the previous iteration. The ILeftShifRegister shares the same attributes NodeId and ParentId and can represent two or three subelements:

(1) AssociatedRightShiftRegister: Output the attributes for NodeId and ParentId of associted right shift register in the same loop by calling node.AssociatedRightShiftRegister.UniqueId and node.AssociatedRightShiftRegister.ParentNode.UniqueId.
(2) InputTerminals and OutputTerminals: Dependent on the terminals' connection of ILeftShiftRegister in the diagram, it may have both InputTerminals and OutputTerminals or only one of them as subelements. We can print out the representation of ITerminal subelements by invoking the PrintTerminals( ) function.

- **XML Generator IRightShiftRegister**

```
public void xg(int indnt, StreamWriter writer, IRightShiftRegister node, string msg)
```

The IRightShiftRegister node's function and representation is extremly similar with the ILeftShiftRegister. It's located on the right side of a loop and can pass the value it receives from the current iteration to the next iteration of left shift register in the pair after the initialization. The IRightShiftRegister has attributes NodeId and ParentId and can also represent two or three subelements:

(1) AssociatedLeftShiftRegister: Represent the attributes for NodeId and ParentId of associted left shift register in the same loop by calling node.AssociatedLeftShiftRegister.UniqueId and node.AssociatedLeftShiftRegister.ParentNode.UniqueId.
(2) InputTerminals and OutputTerminals: Share the same terminals' situation with ILeftShiftRegister by invoking the PrintTerminals( ) function to print out the representation of ITerminal subelements.

- **XML Generator IForLoop**

```
public void xg(int indnt, StreamWriter writer, IForLoop node, string msg)
```

IForLoop can execute its sub diagram n times in the light of loop iteration count, which ranges from 0 to n-1. And the child nodes for IForLoop can be divided into two parts: Diagrams, which is the sub diagram in the for loop and a list of BorderNodes, namely ILoopIndex, ILoopMax, ITunnel, ILeftShiftRegister, IRightShiftRegister. Importantly, the ILeftShiftRegister and IRightShiftRegister always exist in the pair on account of their characteristics mentioned before. Consequently, the IForLoop element has two attributes NodeId and ParentId just like other nodes and is able to traverse its node.BorderNodes and node.Diagrams to output six types of subelements at most:

(1) ILoopIndex: Represent the loop index of current IForLoop by invoking the ILoopIndex's xg( ).
(2) ILoopMax: Output the loop count of current IForLoop by calling ILoopMax's xg( ).
(3) ITunnel: Output the tunnel's information of current For loop by calling the ITunnel's xg( ).
(4) ILeftShiftRegister: Print out the ILeftShiftRegister of current IForLoop by calling its xg( ), which may be not necessary if it doesn't exist in the IForLoop.
(5) IRightShiftRegister: Output the right shift register of current For loop by invoking the IRightShiftRegister's xg( ), which may also not necessary if the ILeftShiftRegister doesn't appear.
(6) IDiagram: Demonstrate the nested sub diagrams of current For loop by calling the IDiagram's xg( ) function.

- **XML Generator ICaseSelector**

```
public void xg(int indnt, StreamWriter writer, ICaseSelector node, string msg)
```

The ICaseSelector can determine which diagram in the case should be executed through passing the data, which can be a Boolean, string, error cluster, integer and enumerated type. The ICaseSelector element has NodeId and ParentId as its attributes and two subelements:
(1) DefaultDiagramIndex: Print out the default diagram index in the ranges of case selector by calling node.DefaultDiagramIndex.
(2) Ranges:

```
public void PrintCaseRanges(int indnt, StreamWriter writer, ReadOnlyCollection<ICaseSelectorRange> ranges)
```

Ranges is the data field of the ICaseSelector. We can obtain the collection of ranges by calling node.Ranges. Hence, the Ranges element can be possessed of multiple Range as its child elements. Indeed, we establish the above PrintCaseRanges( ) function to traverse every Range element content in the Ranges and print out the representation of them sequentially, which has DiagramIndex as the attribute and one or two subelements:

(1) DiagramIndex: Indicate the diagram index corresponded to the diagram executed, by calling the property node.DiagramIndex
(2) SingleValue: Represent the transitive data to determine which case should be executed by calling node.Range.LowValue.
(3) LowValue and HighValue: Represent the range of the transitive data by calling the property node.Range.LowValue and node.Range.HighValue. And they always appear in the pair at the same time because of the range.

- **XML Generator ICaseStructure**

```
public void xg(int indnt, StreamWriter writer, ICaseStructure node, string msg)
```

The case structure contains one or more sub diagrams, which will be decided to execute by the value wired to the case selector. Just as the IForLoop node, the

ICaseStructure node also has two kinds of child nodes: BorderNodes, especially for ICaseSelector and ITunnel, and Diagrams, which is the different sub diagrams in terms of different ranges. As a result, the ICaseStructure element has the same fixed two attributes NodeId and ParentId as well as traverse node.BorderNodes and node.Diagrams separately to obtain three kinds of subelements:

(1) ICaseSelector: Represent the situation of every range in an order in current case structure by invoking ICaseSelector's xg( ) function.
(2) ITunnel: Output ITunnel's information of ICaseStructure by calling ITunnel's xg( ).
(3) IDiagram: Demonstrate the nested sub diagrams of current case sturcture by calling the IDiagram's xg( ) function.

- **XML Generator IFeedbackInitNode**

```
public void xg(int indnt, StreamWriter writer, IFeedbackInitNode node, string msg)
```

The IFeedbackInitNode can transmit the initial value for the first execution or loop iteration. And it need combine with IFeedbackInputNode and IFeedbackOutputNode to form the complete feedback node, which operates similarly to shift registers in a loop. Accordingly, it has the two attributes NodeId and ParentId and three requisite subelements:

(1) AssociatedFeedbackInputNode: Output NodeId and ParentId as the attributes of associated feedback input in feedback node by calling node.AssociatedFeedbackInputNode.UniqueId and node.AssociatedFeedbackInputNode.ParentNode.UniqueId
(2) AssociatedFeedbackOutputNode: Output NodeId and ParentId as attributes of associated feedback output in feedback node by calling node.AssociatedFeedbackOutputNode.UniqueId and node.AssociatedFeedbackOutputNode.ParentNode.UniqueId
(3) InputTerminals: Represent the input terminals which is connected to initial value's terminal by invoking the PrintTerminals( ) function.

- **XML Generator IFeedbackInputNode**

```
public void xg(int indnt, StreamWriter writer, IFeedbackInputNode node, string msg)
```

The IFeedbackInputNode is used for input the data and store n samples of data by delaying the output of the feedback node for multiple executions or iterations. The IFeedbackInputNode element is possessed of the two attributes NodeId and ParentId as well as four subelements as usual:

(1) Delay: The feedback node only outputs the initial value until the Delay is complete when we increase Delay to more than one execution or iteration. Represent it by calling node.Delay.
(2) OutputNode: Print out the NodeId and ParentId of the associated output feedback node by calling node.OutputNode.UniqueId and

node.OutputNode.ParentNode.UniqueId.
(3) InputTerminals: The IFeedbackInputNode has only InputTerminals connected with other nodes. Print out by invoking PrintTerminals( ) function.

- **XML Generator IFeedbackOutputNode**

```
public void xg(int indnt, StreamWriter writer, IFeedbackOutputNode node, string msg)
```

The IFeedbackOutputNode can pass the data stored from the previous execution or iteration. We need obtain its two attributes NodeId and ParentId and 4 kinds of subelements as the representation:

(1) InitTerminal: Represent the terminal Id of the initial IFeedbackOutputNode's input terminal as its attribute by calling node.InitTerminal.UniqueId.
(2) InputNode: Output the NodeId and ParentId of the associated input feedback node by calling node.InputNode.UniqueId and node.InputNode.ParentNode.UniqueId.
(3) InputTerminals and OutputTerminals: The IFeedbackOutputNode has both of them.

- **XML Generator IMethodCall**

```
public void xg(int indnt, StreamWriter writer, IMethodCall node, string msg)
```

When we invoke the IMethodCall's function xg( ) above, it can not only display the list of SubVI but also create a list of new XML file for printing out the every subdiagram in detail. Generally speaking, the IMethodCall has the fixed two attributes NodeId and ParentId as well as 4 types' subelements:
(1) TargetName: Record the corresponding name of SubVI by calling node.TargetName.
(2) SubVI: Pair subelement wraps into the SubVI element, which can represent the corresponding relationship between IMethodCall's TerminalId and IDataAccessor's NodeId inside it for the two attributes TerminalId and SubVIDataAccessor by invoking PrintSubVI( ) function.
(3) InputTerminals and OutputTerminals: Record the connected terminals by invoking the function PrintTerminals( ).

3.3 The Validation of XML Schema

When we generate the XML Intermediate Representation for all the nodes, it's difficult for us to overstate the significance of XML's validation. That's the reason why the XML Schema is tremendously necessary for us, which is the validation file for XML. And I attached my DFIR.xsd in the Appendix A.

In our XML schema, we need make sure the IDiagram element is always the first element occurrence. In addition, we also design some restrictions for certain elements, such showing sequence, occurring times and specific text contents. If we pass incompatible text content to this field, the XML schema can detect the antipathy and throw us an error message. For example:

(1) IDataAccessor: For the Direction element in IDataAccessor, we utilize the simpleType to restrict the pattern value, which must be the options of INPUT and OUTPUT.
(2) ICompoundArithmeticNode: For the Mode attribute in ICompoundArithmeticNode, we restrict the pattern value to be the options among Add, Multiply, Or, And, Xor.

3.4 The Summary of Chapter 3

In this chapter, we talk about how to design the intermediate representation. We can output both DOT file and XML file by the C# project. And I explain the meaning of subelements and attributes for every node element in detail. The Table 3-1 is the summary of the IR in the XML file (* represent the range of it occurrence from 0 to unbounded times):

Table 3-1 The Summary of the IR

Node Type	SubElement	Attribute
IDiagram	ICompoundArithmeticNode* IDataAccessor* IConstant* IForLoop* IWhileLoop* ICaseStructure* IMethodCall* IPrimitive* {IFeedbackInputNode,IFeedbackOutputNode}* IBlackBoxNode* IArrayIndexNode* IReplaceArraySubsetNode* IInsertIntoArrayNode* IDeleteFromArrayNode* IInitializeArrayNode* IBuildArrayNode* IArraySubsetNode*	NodeId (byte) ParentId (byte) DiagramIndex (byte)

IDataType	ISignedInt IUnsignedInt ISignedFixedPoint IUnsignedFixedPoint IArray IVariableSizedArray IComplex IFixedSizeArray IBit IBoolean IDouble IIncorrect ISingle IString IUnknown IUnsupported IVoid	NodeId (byte) ParentId (byte)
ISignedInt IUnsignedInt	WordLength (byte)	
ISignedFixedPoint IUnsignedFixedPoint	LeftLength (byte) RightLength (byte)	
IArray IVariableSizedArray	Dimensions (byte) Element	
IComplex	Element	
Element	IDataType	
Connections	Connection(TerminalId, NodeId)*	
ITerminal	IDataType Connections	TerminalId (byte) TerminalIndex (byte)
InputTerminals	ITerminal*	
OutputTerminals	ITerminal*	
IDataAccessor	Name (String) Direction (INPUT || UTPUT) InputTerminals || OutputTerminals	NodeId (byte) ParentId (byte)

IConstant	Value (String) IDataType OutputTerminals	NodeId (byte) ParentId (byte)
ICompoundArithmeticNode	InvertedInputs InvertedOutput InputTerminals OutputTerminals	NodeId (byte) ParentId (byte) Mode (Add || Multiply || Or || And || Xor)
IPrimitive	InputTerminals OutputTerminals	NodeId (byte) ParentId (byte) Mode
ITunnel	IsInput (True || False) GetInnerTerminal (TerminalId) GetOuterTerminal (TerminalId) IndexingMode(NonIndexing|| Indexing) InputTerminals OutputTerminals	NodeId (byte) ParentId (byte)
ILoopIndex	OutputTerminals	NodeId (byte) ParentId (byte)
ILoopMax	InputTerminals OutputTerminals	NodeId (byte) ParentId (byte)
ILeftShiftRegister	AssociatedRightShiftRegister(NodeId, ParentId) InputTerminals OutputTerminals	NodeId (byte) ParentId (byte)
IRightShiftRegister	AssociatedLeftShiftRegister(NodeId, ParentId) InputTerminals OutputTerminals	NodeId (byte) ParentId (byte)
IForLoop	ILoopIndex ILoopMax ITunnel* {ILeftShiftRegister,IRightShiftRegister}* IDiagram	NodeId (byte) ParentId (byte)
ICaseSelector	DefaultDiagramIndex (byte) Ranges {Range (LowValue, HighValue) || SingleValue } InputTerminals	NodeId (byte) ParentId (byte)

	OutputTerminals	
ICaseStructure	ICaseSelector ITunnel* IDiagram	NodeId (byte) ParentId (byte)
IFeedbackInputNode	Delay (byte) FeedbackTerminal(TerminalId) OutputNode(NodeId, ParentId) InputTerminals OutputTerminals	NodeId (byte) ParentId (byte)

IFeedbackOutputNode	InitTerminal(TerminalId)
	InputNode(NodeId, ParentId)
	InputTerminals
	OutputTerminals
		NodeId (byte)
		ParentId (byte)

SubVI	Pair<TerminalId, SubVIDataAccessorId >	NodeId (byte) ParentId (byte)
IMethodCall	TargetName (String) SubVI InputTerminals OutputTerminals	NodeId (byte) ParentId (byte)
IBlackBoxNode IArrayIndexNode IReplaceArraySubsetNode IInsertIntoArrayNode IDeleteFromArrayNode IInitializeArrayNode IBuildArrayNode IArraySubsetNode	InputTerminals OutputTerminals	NodeId (byte) ParentId (byte)

Chapter 4 The C Code Generation by C# Compiler

In my project, I choose C# in Visual Studio to design my compiler to convert XML file into C code. Because Microsoft's .NET Framework has the built-in XML DOM parser, I can utilize the convenient parser in C# to process the XML file effectively, which is necessary for me to establish the code converter. Furthermore, I build a series of nodes classes in different .cs files, similar with the organized classes in the DFIR, to help me create and invoke my C# program to generate C code more efficiently.

In my C# compiler, I build a list of classes to store, access, process the information of every node, which follows the definition of classes' name. These classes can be divided into five parts in DFIR in the Table 4-1:

Table 4-1 The Compiler Classes

Class Classification	**Description**
DFIR Representation	INode class is the base class which has a collection of derived classes: IDataAccessor, IConstant, IDiagram, ITunnel, ILoopIndex, ILoopMax, ILeftShiftRegister, IRightShiftRegister, ICompoundArithmeticNode, IPrimitive, IForLoop, ICaseStructure And these classes need invoke some functional classes: IDataType, IDataTypeBuilder, Connection, AssociatedLeftShiftRegister, AssociatedRightShiftRegister, ITerminal, ICaseSelector, Range
Enum	The public enum types invoked by the other classes: ICompoundArithmeticNodeMode, BaseDataType, IPrimitiveMode, IsInputMode, IndexingMode, Direction.
DomParser	Process the XML file and store the information of nodes.
CodeConverter	Generate C code by using topologic sorting.
main SchemaValidation	Load the XML file and execute the whole C# program. Validate the XML file with XML Schema

4.1 DFIR Representation

As the DFIR Representation in Table 4-1, there are many classes to achieve different functions in my compiler. And the following classes I'll explain are the most important classes, which can fulfill the main functions in my C code.

4.1.1 INode Class

Among the DFIR Representation classes, due to the base class INode, other nodes' classes can inherit its properties and override its methods. The INode in my compiler is quite different from the INode in DFIR. I package the INode's properties as the protected internal members and declare the corresponding public Get methods to provide access from the invocation outside. Accordingly, other class can inherit INode's protected members as the private members and the public methods, which equip the ability to offer the program better encapsulation, confidentiality and security.

The Table 4-2 is the description of INode's method:

Table 4-2 The Method of INode Class

Method	**Description**
INode	Construct function to declare the new INode variable
GetNodeId	Get the NodeId of this INode
GetParentId	Get the ParentId of this INode
GetIsInputConnected	Get if the INode's InputTerminals connect with other INodes
GetIsOutputConnected	Get if the INode's OutputTerminals connect with other INodes
GetParentINode	Get the Parent INode through the IDictionary reflection from ParentId to the target INode.
GetINodeType	Get the INode type of this INode
Occupied	Get if this INode is used or set its Boolean value
IsDeclared	Get if this INode is declared or set its Boolean value
GetInputTerminals	Get the list of InputTerminals of this INode
GetOutputTerminals	Get the list of OutputTerminals of this INode
GetInputConnectionIndex	Get the corresponding input connection index of this INode
GetOutputConnectionIndex	Get the corresponding output connection index of this INode
ConnectedInputTerminals	Get the list of input terminals of this INode

ConnectedOutputTermianls	Get the list of output terminals of this INode
GetIdToINode	Get the IDictionary reflection from NodeId to this INode
GetTerminalIndexToINodeId	Get the IDictionary reflection from the input TerminalIndex of this INode to its connected INode's NodeId
GetInputTerminalIDatatype	Get the datatype of this input terminal given the specific index
GetOutputTerminalIDatatype	Get the datatype of this output terminal given the specific index
GetInputINodes	Get the list of input INodes of this INode
GetOutputINodes	Get the list of output INodes of this INode
Indegree	Get the count of the input connections of this INode
DecIndegree	Decrease the count of input connections of this INode once
IncIndegree	Increase the count of input connections of this INode once
GetFrontINode	Get the list of the front INodes of this INode
AddInputTerminals	Add the ITerminal to the InputTerminals of this INode
AddOutputTerminals	Add the ITerminal to the OutputTerminals of this INode
GetName	Get the name of this INode
Print	Print out the necessary representation of this INode

4.1.2 IDataType Class

In my C# Compiler, I create a public Enum class named BaseDataType to contain the IDataType in DFIR. So I create the IDataType class and IDataTypeBuilder class, which can get the field of every BaseDataType to declare the variable's datatype in C code. Furthermore, in order to convert the datatype in FPGA LabVIEW Communications into the declared variables' datatype in C, I assign some rules to figure out the corresponding datatype as Table 4-3:

Table 4-3 Declared Datatype in C to Correspond the IDataType in DFIR

BaseDataType	**Field**	**Declared Datatype in C Code**
ISignedInt	WordLength	short, int or long
IUnsignedInt	WordLength	unsigned short, unsigned int or unsigned long
ISignedFixedPoint, IUnsignedFixedPoint	LeftLength, RightLength	double: Because C don't support fixedpoints

IArray, IVariableSizedArray IFixedSizeArray	Dimensions, ElementType	Create an Array class to declare and represent the array.
	ArraySize	
IComplex	ElementType	Invoking complex in C Library
IBoolean	Null	bool
IDouble, ISingle,	Null	double
IBit, IString, IUnknown, IIncorrect, IUnsupported, IVoid	Null	Null

4.1.3 IDiagram Class

The IDiagram class, which can be considered as the container for all other nodes, derives from the INode class. Although IDiagram can be nested if there are IForLoop or ICaseStructure in it, the VI in LabVIEW Communications has only one top IDiagram. Consequently, the IDiagram contains many different characteristics even if it inherits the INode class. For instance, it has its particular field named diagramIndex as well as without any input terminals and output terminals. It can contain zero or multiple nodes, such as ICompoundArithmeticNode, IDataAccessor, IConstant, IForLoop, ICaseStructure, IMethodCall, IPrimitive, IFeedbackOutputNode and so on.

4.1.4 IConstant Class

The IConstant class derives from the INode class as well. Because IConstant node has value itself, which means both its terminal and itself have datatype. Accordingly, it has value and iDataType as its private fields. And in order to obtain its information, I create the specific methods as Table 4-4:

Table 4-4 The Specific Method of IConstant Class

Method	**Description**
GetIDataType	Get the value's datatype of this IConstant
GetValue	Get value of this IConstant, especially processing the output of value for IArray, IVariableSizedArray, IFixedSizeArray and IBoolean.
GetReal	Get the real part if this IConstant value's BaseDataType is IComplex
GetImag	Get the imaginary part if this IConstant value's BaseDataType is IComplex

4.1.5 IDataAccessor Class

The IDataAccessor class also inherits the INode class. It has its own private fields, which are name and direction. The IDataAccessor node in LabVIEW Communications can be divided into control and indicator. Accordingly, I create the public Enum Direction class, which has INPUT and OUTPUT element to indicate the control and indicator separately and invoke the GetDirection() function to obtain its direction.

4.1.6 ICompoundArithmeticNode Class

The ICompoundArithmeticNode can have more than two multiple input terminals and only one output terminal. The ICompoundArithmeticNode class, which has own private members: mode, InvertedInputs and InvertedOutput, derives from the INode class. Importantly, the mode member is declared by the public Enum class ICompoundArithmeticNodeMode: Add, Multiply, Or, And, Xor. Hence, I build the following specific methods to process ICompoundArithmeticNode as Table 4-5:

Table 4-5 The Specific Method of ICompoundArithmeticNode Class

Method	**Description**
GetMode	Get the ICompoundArithmeticNode mode from the Enum class ICompoundArithmeticNodeMode
GetInvertedInputs	Get the list of inverted input terminals’ Boolean value
GetInvertedOutput	Get the Boolean value of inverted output terminal

4.1.7 IForLoop Class

The IForLoop class inherits the INode class. In the VI diagram, the IForLoop node can contain multiple nested IDiagram class and IBorderNode class: one ILoopIndex, one ILoopMax, a list of ITunnel, ILeftShiftRegister and IRightShiftRegister.

Due to the direction of ITunnel, I divide the IBorderNodes into two groups: inputBorderNodes, including input ITunnel and ILeftShiftRegister, and outputBorderNodes, namely output ITunnel and IRightShiftRegister. Accordingly, I declare eight private members in IForLoop class: iDiagram, iLoopIndex, iLoopMax, iTunnels, iLeftShiftRegisters, iRightShiftRegisters, inputBorderNodes, outputBorderNodes as well as corresponding Get method to obtain them.

- **ITunnel Class**

The ITunnel class inherits the IBorderNode class, which derives from the INode class. I declare the IndexingMode as private member from the public Enum class IndexingMode: Indexing and NonIndexing. And other important method to process ITunnel class is in following Table 4-6:

Table 4-6 The Specific Method of ITunnel Class

Method	**Description**
IsIndexingITunnelInsideLoop	Judge if there's the indexing ITunnel inside for loop
IsOutputCaseTunnel	Judge if it's output ITunnel in the ICaseStructure
GetIsInput	Get the Boolean value of this ITunnel's direction
GetInsideTerminal	Get the specific terminal connected to inside nodes
GetOutsideTerminal	Get the specific terminal connected to outside nodes

- **ILeftShiftRegister Class and IRightShiftRegister Class**

Because the ILeftShiftRegister and IRightShiftRegister always occur in pair, they have similar classes derived from IBorderNode. Consequently, the private members are the AssociatedRightShiftRegister for ILeftShiftRegister and the AssociatedLeftShiftRegister for IRightShiftRegister, which related them together as a pair.

4.1.8 ICaseStructure Class

The ICaseStructure class can contain one ICaseSelector, a list of ITunnels as its border nodes and a list of IDiagrams corresponding to the Range the case has. Therefore, I declare three private members: iTunnels, iDiagrams, iCaseSelector as well as corresponding Get and Set method to obtain and set them. Besides, in order to obtain more detailed information of ICaseStructure, I create the following specific method as Table 4-7:

Table 4-7 The Specific Method of ICaseStructure Class

Method	**Description**
NonDefaultCaseIndex	Get the index for the no default case
NonDefaultCaseCode	Convert the range into the C code for the no default case

- **ICaseSelector Class**

The ICaseSelector class derives from the IBorderNode class. And it has two field: defaultDiagramIndex, which can indicate which IDiagram is default, and ranges, namely a list of Range to store the mapping index information.

- **Range Class**

I declare the lowValue, highValue, singleValue, diagramIndex as the private members in the Range class. Attentively, the lowValue and highValue always occur in a pair. Hence, I can get and set these fields by declaring corresponding method if my compiler accesses the ICaseStructure class.

4.1.9 IPrimitive Class

The IPrimitive class is very similar to the INode class apart from its specific field: mode, which can identify its function in the VI diagram. The number of IPrimitives is huge, which means the significance of the mode to distinguish them from each other. As a result, I build a collection of certain function for some of the IPrimitives when the code converter need invoke it.

4.2 The DomParser for XML File

4.2.1 The System.Xml Namespace

In the Microsoft's .NET Framework Library, it provides me the System.Xml as well as its four supportive namespaces to define the functionality to process with XML file to build the DOM tree. These supportive namespaces are System.Xml.Schema, System.Xml.Serialization, System.Xml.Xpath, and System.Xml.Xsl.

Among these namespaces, I use System.Xml and System.Xml.Schema namespace mainly. The System.Xml namespace defines the basic and major XML functionality as well as the classes for XML 1.0 XML namespace and schemas. And the System.Xml.Schema includes many classes to work with XML schemas, which can support XML schemas for structures and data types.

4.2.2 The DOM Interfaces

Document Object Model (DOM) is a platform-neutral interface that allows programs and scripts to access and update XML and HTML documents dynamically. Furthermore, DOM defines the logical structure of a document's data. The program can process the document in a structured format, generally a DOM tree format. Besides, DOM can also help developers to complete building, adding, modifying, deleting, navigating as well as other manipulation in the XML document. Therefore, the various contents in a DOM tree can represent the document's data as Figure 4-1:

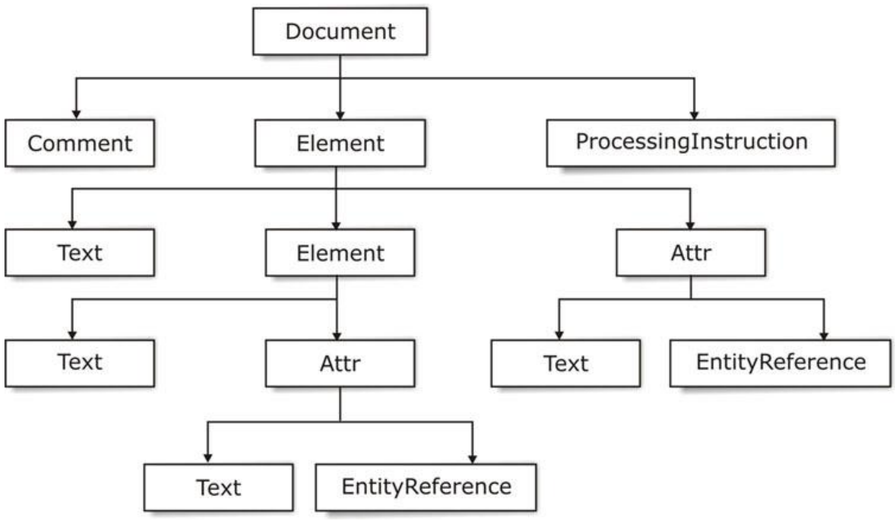


Figure 4-1 The DOM Tree

Microsoft .NET provides a brilliant wrapper around the DOM interface: the DOM API, which has a class for almost every interface. These classes offer a high-level programming model for developers with hiding all the complexity of interface programming. Accordingly, I create a DomParser.cs to process my XML file in a tree structure using DOM interfaces and objects in Table 4-3:

Table 4-3 The Method in the DomParser.cs

Method	**Description**
GetIDataType	Get the IDataType of current XmlNode node.
GetITerminal	Get the ITerminal of current XmlElement node.
GetIDiagram	Get the IDiagram of current XmlNode node and build the collection of all kinds of nodes' XmlNodeList.

The DOM implementation provided by the .NET Framework has three core classes: XmlDocument, XmlNode, XmlElement. In the methods of my DomParser.cs, I use these three classes mainly to access and traverse through my XML documents.

- **The XmlNode Class**

The XmlNode class is the abstract base class. It can represent a tree node, which can be the entire document. This base class defines enough methods and properties to access and traverse the document node. The following Table 4-4 is the properties I invoke mostly:

Table 4-4 The Property of the XmlNode Class

Property	**Description**

InnerText	Gets or sets the concatenated values of the node and all its child nodes.
Name	Gets the qualified name of the node, when overridden in a derived class.
NodeType	Gets the type of the current node, when overridden in a derived class.

- **The XmlDocument Class**

The XmlDocument class is derived from the XmlNode class. It can also represent an XML document through its plenty of properties and methods just like XmlNode class. DOM is a cache tree representation of an XML document. In my main.cs file, I utilize the XmlDocument class to focus on the Load Method from a string filename to obtain the DOM tree:

```
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.Load(filename);
xmlDoc.DocumentElement.Normalize();
```

- **The XmlElement Class**

The XmlElement class is derived from the XmlLinkedNode class, which comes from XmlNode. This class object can represent an element in a document. It inherits and overrides lots of properties and methods from XmlNode class. Besides, it inherits the two useful properties: NextSibling and PreviousSibling from the XmlLinkedNode class. In my DomParser.cs, I often to specify an XmlNode node into an XmlElement node in order to invoke its useful properties and methods as Table 4-5:

Table 4-5 The Method of the XmlElement Class

Method	**Description**
GetElementsByTagName	Returns an XmlNodeList containing a list of all descendant elements that match the specified Name.
GetAttribute	Returns the value for the attribute with the specified name.

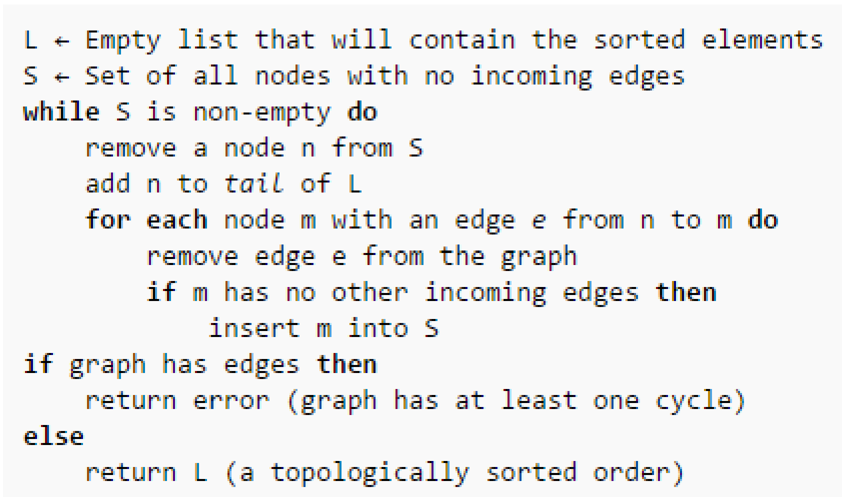
4.3 The Code Converter

The Code Converter can convert the information stored in IDiagram into C code, which is the most challenging part of the whole compiler. Because the C language is the graphical programming language in LabVIEW Communications, which tremendously differs from the textual programming language such as C. For example, the node in LabVIEW will only execute the diagram when all required data inputs it, which cannot be found in the logic of C.

As a result, it’s hard to overstate the significance of the C code’s logical sequence. According to the logic in G, I choose the topological sorting algorithm, which is the classical graph algorithms similar with G, to process the order of all nodes. Consequently, all the methods in my compiler can be divided into two groups: topological sorting and processing methods to generate C code.

4.3.1 Topological Sorting

The topological sorting of a directed graph is a linear ordering of its vertices such that for every directed edge uv from vertex u to vertex v, u comes before v in the ordering. My converter utilizes the thoughts of Kahn’s algorithm, which chooses vertices in the same order as the eventual topological sort. It inserts a list of nodes which have no incoming edges into the set S, and then executes and obtains the target topological sorting order L as the following Figure 4-1:



```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edges
while S is non-empty do
    remove a node n from S
    add n to tail of L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S
if graph has edges then
    return error (graph has at least one cycle)
else
    return L (a topologically sorted order)
```

Figure 4-1 The Kahn’s algorithm

Correspondingly, in terms of the Kahn’s algorithm, the nodes without input terminals can be regarded as zero indegree, such as IConstant, input IDataAccessor and ILoopIndex inside the IForLoop. My converter can get the list of them and sort them topologically.

- **Get the List of Zero Indegree Nodes**

On account of the existence of nested IDiagram, I declare a Boolean type isTop to distinguish whether it’s the top level IDiagram or the nested IDiagram.

For the top level IDiagram, only the IConstant and input IDataAccessor can be the list of zero Indegree nodes. Even if the IForLoop and ICaseStructure don’t have any input terminals, they cannot be regarded as the zero Indegree nodes because their border nodes have the input terminals. In other words, they need appear dependent on other nodes.

As for the nested IDiagram, we should consider more zero indegree nodes inside except the zero Indegree node in the top level IDiagram. For the IDiagram in the ICaseStructure, we should connect the ICaseSelector and input ITunnels with this IDiagram together. Because they may not link with the nodes inside the IDiagram sometimes. In addition, for the IDiagram in the IForLoop, we should add the input ITunnels and ILeftShiftRegister into the list of zero indegree nodes. And for the ILoopIndex and ILoopMax, if they have the output connections, we also need put them into the list.

- **Sort the List of Zero Indegree Nodes Topologically**

When we obtain the list of zero indegree nodes, we can exert the topological sorting we talked above to order the sequence of all the nodes. Different nodes have different processing method. For the nodes except IDataAccessor and IConstant, I apply particular sorting method to process them.

First of all, for the input IDataAccessor and IConstant, on account of the zero indegree they have, they will be always processed to appear in the front of the sorted list.

Furthermore, as for the IForLoop, ICaseStructure, although they don't have any indegree, the indegree's sum of their border nodes such as ILeftShiftRegister, IRightShiftRegister, ITunnel and ICaseSelector can be regarded as their virtual indegree. Therefore, I utilize the DecIndegree method to decrease their indegree when their border nodes be put into the sorted list. When their indegree becomes zero, they also will be added into the sorted list. Moreover, for the input border nodes, their indegree can only be decreased when they are out of the IForLoop and ICaseStructure. On the contrary, their indegree cannot be processed to decrease before entering into the IForLoop and ICaseStructure, because they belong to the IDiagram in the IForLoop and ICaseStructure. Analogically, for the output border nodes we can only decrease their indegree when they're in the IDiagram of IForLoop and ICaseStructure.

Particularly, for the output ITunnel in ICaseStructure, it usually has one output terminals and multiple input terminals, which connected with certain case. Accordingly, the indegree of the output ITunnel is the same with the number of cases in ICaseStructure. And it divided its indegree averagely into every case, which means that it has one indegree for every IDiagram in ICaseStructure.

In addition, the feedback node in LabVIEW Communications splits into IFeedbackInputNode, IFeedbackOutputNode and IFeedbackInitNode(inaccessible) in DFIR. Hence we ought to assume the virtual connection between the IFeedbackInputNode and IFeedbackOutputNode. For example, if we add the IFeedbackInputNode into the sorted list, we should decrease one indegree of IFeedbackOutputNode. Likely, when we consider the indegree of

IFeedbackOutputNode, we should add one indegree automatically due to the virtual connection of IFeedbackInputNode.

4.3.2 C Code Rules

In the converter, we need stipulate some rules for the output C code, which enables us to organize and manage the C code more efficiently and effectively. They're naming system, declaration system and organization system.

- **Naming System**

In my naming system, on consideration of the uniqueness of INodeType and NodeId in DFIR representation, I use the GetName() method in INode class, which is the INodeType + NodeId of this node, to output INode's name as the corresponding variables' name in my C code . Consequently, almost all of the nodes can derive this naming system from the GetName() method except some special nodes in Table 4-6:

Table 4-6 The Naming Method of Special Nodes

Node	**Naming Method**	**Description**
IDataAccessor	IDataAccessor.name	Get the name field of this IDataAccessor
ILeftShiftRegister	GetINodeType() + GetNodeId() + '_' + index	The index is the list's index of InputTerminals and OutputTerminals because of the terminals' extensibility of ILeftShiftRegister
Indexing ITunnel	GetName() + [GetILoopIndex(). GetName()]	The datatype's dimension of the indexing ITunnel's inner ITerminal and outer ITerminal is different when it's inside the IForLoop
Other nodes	GetName()	Get the INodeType + NodeId of this node

- **Declaration System**

On consideration of the declaration's necessity of the variable, we have to declare it before using it. In my compiler, I prefer designing to declare the variables when I need use them rather than declaring all of them in the beginning. Because for some local temporary variables, we needn't declare them as the global variables. Moreover, for the border nodes in IForLoop and ICaseStructure, we also ought to declare them before processing the IDiagram in IForLoop and ICaseStructure.

Therefore, I create the Declaration() method to output the declaration according to the terminals' datatype and its corresponding index and direction in related INode. In addition, I create two subroutines inside the Declaration() to be invoked owing to the existence of array declaration. They're the SubIConstantDeclaration() for

IConstant specifically and the SubDeclaration() for other variables due to different situation of their input and output terminals.

The Table 4-7 below is the declaration sample for different nodes. And we assume all of the datatypes in the sample is int.

Table 4-7 The Declaration Sample of Nodes

Node	**Declaration Sample**
ICompoundArithmeticNode	int variable;
array variable	int variable[]={0,1,2};
Fixedsize array variable	int variable[3]={1,2,3};
array IConstant	int IConstant[]={0,1,2};
Fixedsize array IConstant	int IConstant[3]={1,2,3};
ILeftShiftRegister	int ILeftShiftRegister1_0, ILeftShiftRegister1_1…
Indexing ITunnel	int ITunnel[];

- **Organization System**

For the purpose of completing the expression as the executable expression in C code, we need add the semicolon in the rear of every expression. Hence, I design the AddSemicolon() method to fulfill this task.

Furthermore, in order to improve the readability of C code, for every executable expression we obtain, we need add the appropriate indentation. Therefore, I create the Indentation() method to provide necessary indentation in front of every expression, which equips my converted code the ability to look the same with the real C code.

4.3.3 Processing Methods

Once we fulfill the topological sorting order and declaration design, we can create a series of processing methods to convert this order into C code. On account of the processing logical similarity of some nodes, we can split them into several groups to handle specifically as Table 4-8.

Table 4-8 The Classification of Processing Methods

Classification	**Class**
Only Right Identifier	IConstant, ILoopIndex
Both Left and Right	IDataAccessor, ILoopMax, ILeftShiftRegister, IRightShiftRegister,

Identifier	ITunnel, ICaseSelector
Operator	ICompoundArithmeticNode, IPrimitive(mode= ExSubtractPrimitive, ExDividePrimitive)
For Loop	IForLoop
Case Structure	ICaseStructure
Packaged Function	IPrimitive (except mode= ExSubtractPrimitive, ExDividePrimitive), Array Operation Nodes(IArrayIndexNode, IInsertIntoArrayNode, IReplaceArraySubsetNode, IBuildArrayNode, IArraySubsetNode IInitializeArrayNode, IDeleteFromArrayNode,)

- **Only Right Identifier**

The IConstant and ILoopIndex have only output terminals, which means that they can only be the right identifier in C code. They cannot be used until their sink connected nodes are processed.

- **Both Left and Right Identifier**

The nodes in this classification can be used as both left and right identifiers. Generally speaking, when they serve as the left variables, they will be processed to generate from itself and its source connected node. On the contrary, when they serve as the right variables, they won't be used until their sink connected nodes are processed. However, for some special nodes in this classification, they have their own processing method due to their properties and functions.

For the IDataAccessor named n, if it appears as the left variable, we need program the sentence `"printf(\"" + Specifier(n) + "\\n\"," + n.GetName() + ")"` as outputting this variable in C code. Because it's necessary to assign the datatype of variable when inputting or outputting, the Specifier() can obtain the datatype of this variable. Likely, when it appears as the right variable, we can program the sentence `"scanf_s(\""+Specifier(n)+"\", &" + n.GetName() + ")"` as inputting this variable in C code. Notably, because C cannot input or output Boolean value, I use the value 0, 1(int) to replace the false, true(bool).

Furthermore, as for the ILeftShiftRegister, because of multiple input terminals, it can pass different value in the diagram. I create a list of bool Occupied field for every input terminal. Hence, when we process the ILeftShiftRegister as the left variable, we need traverse all of its input terminals to generate expression of every input terminal if the Occupied is false.

In addition, if there is the output ITunnel for ICaseStructure, we need process it according to its source connected node under the case IDiagram. Moreover, if it's the indexing ITunnel, we should declare different datatypes of its inner terminal and outer terminal because of the difference between the dimension of their datatypes.

- **Operator**

The operator can be the ICompoundArithmeticNode (mode= Add, Multiply, Or, And, Xor) and IPrimitive (mode= ExSubtractPrimitive, ExDividePrimitive). I declare the IDictionary of ICompoundArithmeticNodeModeToString and IPrimitiveModeToString fields separately to convert their mode into operator in C, which are +, *, ||, &&, ^, -, / correspondingly.

Although only these two modes in IPrimitive have corresponding operator in C, they're not assigned into ICompoundArithmeticNode because they have only two input terminals with different properties. And the nodes' input terminals in ICompoundArithmeticNode, namely the order of the operands in C, have the same identities. Besides, the terminals of ICompoundArithmeticNode can be inverted, which means the value passed this terminal will be negative.

In order to obtain an executable and integrated expression including operator in C, we need use "=" to connect both left identifier and right expression, which combines multiple operands and the operator.

On one side, for the left identifier, we can use the name of its sink nodes as the left variables, such as output IDataAccessor. Furthermore, if all of its sink node cannot be the left variables alone, we can create the temporary variable for it. On the other side, for the right expression, the operator's source connected nodes by their name will be declared if they're used as the operands. And if the source connected node is IConstant, its value will be used as the operands directly.

- **For Loop**

Due to the nested IDiagram inside the IForLoop, it's necessary for us to invoke the processing IDiagram function recursively.

At first, we should consider about the declaration of the border nodes in IForLoop. Undeniably we ought to declare the input ITunnels and ILeftShiftRegister before entering the IForLoop. Besides, if the output ITunnels and IRightShiftRegister are used outside before entering into this IForLoop, we also should declare them firstly.

In addition, after entering into the IForLoop, we can generate the for head expression as the "for(int ILoopIndex=0; ILoopIndex<ILoopMax; ILoopIndex++)". Moreover, the inside IDiagram will be processed recursively to print out the expression in the for expression. And if the ILeftShiftRegister has multiple terminals, we should process it to output the expression, which can pass down the value among the terminals of ILeftShiftRegister one by one and then receive the value from the IRightShiftRegister to the first index terminal of ILeftShiftRegister.

- **Case Structure**

The ICaseStructure can also contain nested IDiagrams, which means that we can call the processing IDiagram function recursively as well. And I choose the if-else statement in C to execute the ICaseStructure rather than switch statement. Because the judgement in switch statement of C must be a constant of a literal, we can set either single value or the range to navigate the specific case in LabVIEW Communications. The range is the Boolean expression, which means we need utilize if-else statement instead of switch statement.

Obviously, we also need declare the border nodes, which is the output ITunnel. And it's important for us to count the number of cases, that's the number of IDiagrams ICaseStructure has. If there is only one case, it must be the default case. And it can be processed directly when the targeted value passes through the ICaseSelector. But if there is more than one case, I choose to generate all the non-default cases' code by the method NonDefaultCaseCode() firstly and then print out their inside IDiagram. And the default case is the last one in the else statement.

- **Packaged Function**

For all of the IPrimitives (except mode= ExSubtractPrimitive and ExDividePrimitive) and the array operation nodes, I package them into separate functions. They will be generated once the diagram in LabVIEW Communications uses the corresponding nodes.

And main function in the C code also should generate the calling expression if it need invoke the function. Therefore, we need obtain the list of the parameters, which can be the data in both input terminals and output terminals, to pass into the function. For the output terminals, we need declare them before calling the targeted function because they didn't appear before. And as for the input terminals, we don't have to declare them because they must have been declared or used before.

4.4 Compiler Result

After I fulfilled my compiler, I create a series of tests for the main nodes to compare the running results between the diagram in the LabVIEW Communications and the generated C code.

4.4.1 ICompoundArithmeticNode

- **Mode=Add, Multiply**

Based on the ICompoundArithmeticNode processing method above, I create the sample Add_Multiply.gvi in LabVIEW Communication as the Figure 4-2:

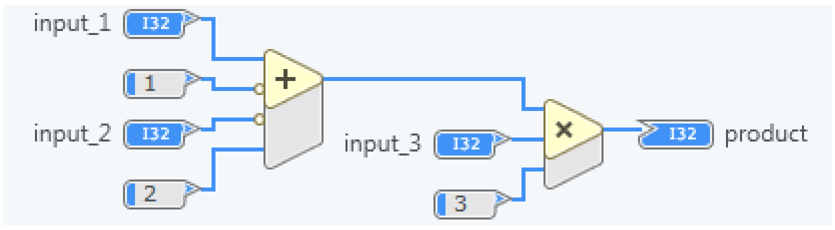


Figure 4-2 The Add_Multiply.gvi in LabVIEW Communications

And the converted C code Add_Multiply.c file sample about operator ”+” and operator ”*” from Add_Multiply.gvi is shown in the Figure 4-3:

```
#include<stdio.h>
#include<stdlib.h>

int main()
{
    int input_1;
    scanf_s("%d", &input_1);
    int input_2;
    scanf_s("%d", &input_2);
    int input_3;
    scanf_s("%d", &input_3);
    int ICompoundArithmeticNode2 = input_1 + (-1) + (-input_2) + 2;
    int product = ICompoundArithmeticNode2*input_3 * 3;
    printf("%d\n", product);

    system("pause");
    return 0;
}
```

Figure 4-3 The Converted C Code Add_Multiply.c

Accordingly, I test plenty of data and list 5 groups of data randomly as the test sample to compare their running result as Table 4-8:

Table 4-8 The Result Comparison Between Add_Multiply.gvi and Add_Multiply.c

Add_Multiply.gvi				Add_Multiply.c			
input_1	input_2	input_3	product	input_1	input_2	input_3	product
4	3	5	30	4	3	5	30
1	2	3	0	1	2	3	0
-2	10	-4	132	-2	10	-4	132
23	0	6	432	23	0	6	432
154	66	-27	-7209	154	66	-27	-7209

From the exact same sample result in the Add_Multiply.gvi and Add_Multiply.c, we can conclude that the operator “+” and operator ”*” work perfectly and we convert the diagram in LabVIEW Communications into C code successfully.

- **Mode= And, Or, Xor**

For the ICompoundArithmeticNode (mode=And, Or, Xor), I combine them together to design the And_Or_Xor.gvi in LabVIEW Communications as Figure 4-4:

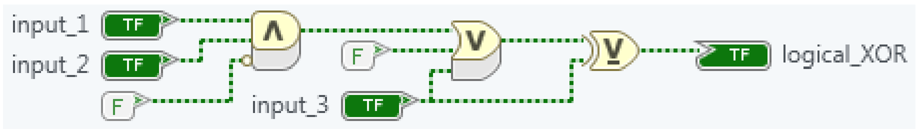


Figure 4-4 The And_Or_Xor.gvi in LabVIEW Communications

Next, I run my compiler to generate corresponding And_Or_Xor.c file about operator “&”, “|”, “^” as Figure 4-5:

```
#include<stdio.h>
#include<stdlib.h>

int main()
{
    int input_1;
    scanf_s("%d", &input_1);
    int input_2;
    scanf_s("%d", &input_2);
    int input_3;
    scanf_s("%d", &input_3);
    int ICompoundArithmeticNode2 = input_1&input_2 & true;
    int ICompoundArithmeticNode5 = ICompoundArithmeticNode2 | false | input_3;
    int logical_XOR = ICompoundArithmeticNode5^input_3;
    printf("%d\n", logical_XOR);

    system("pause");
    return 0;
}
```

Figure 4-5 The Converted C Code And_Or_Xor.c

Accordingly, I test plenty of data and list 5 groups of data randomly as the test sample to compare their running result as Table 4-9:

Table 4-9 The Result Comparison Between And_Or_Xor.gvi and And_Or_Xor.c

And_Or_Xor.gvi				And_Or_Xor.c			
input_1	input_2	input_3	product	input_1	input_2	input_3	product
0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0
1	1	0	1	1	1	0	1
0	0	1	0	0	0	1	0
1	1	1	0	1	1	1	0

Above all, according to the comparison, we convert the diagram in LabVIEW Communications into C code successfully.

4.4.2 IPrimitive

- **mode= ExSubtractPrimitive, ExDividePrimitive**

In LabVIEW Communications, I design the “Subtract_Divide.gvi” as Figure 4-6:

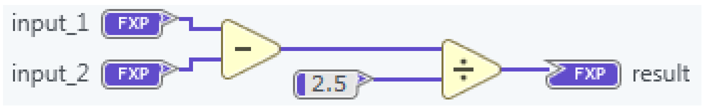


Figure 4-6 The Subtract_Divide.gvi in LabVIEW Communications

According to the similar IPrimitive (mode= ExSubtractPrimitive, ExDividePrimitive) processing method, I generate the converted C code sample Subtract_Divide.c file about operator “-“, “/” from the “Subtract_Divide.gvi” as Figure 4-7:

```
#include<stdio.h>
#include<stdlib.h>

int main()
{
    double input_1;
    scanf_s("%lf", &input_1);
    double input_2;
    scanf_s("%lf", &input_2);
    double IPrimitive5 = input_1 - input_2;
    double result = IPrimitive5 / 2.5;
    printf("%lf\n", result);

    system("pause");
    return 0;
}
```

Figure 4-7 The Converted C Code Subtract_Divide.c

Consequently, I test plenty of data and list 5 groups of data randomly as the test sample to compare their running result as Table 4-10:

Table 4-10 The Comparison Between Subtract_Divide.gvi and Subtract_Divide.c

Subtract_Divide.gvi			Subtract_Divide.c		
input_1	input_2	product	input_1	input_2	product
3.379882	1.452148	0.771118	3.379882	1.452148	0.771094
4.705078	11.64453	-2.77575	4.705078	11.64453	-2.775781
46.13476	35.24365	4.356445	46.13476	35.24365	4.356444
-8.46191	462.4428	-188.361	-8.46191	462.4428	-188.361884
323.4643	55.53320	107.1724	323.4643	55.53320	107.172440

On the basis of the comparison of these two results, we can conclude that we convert IPrimitive (mode== ExSubtractPrimitive, ExDividePrimitive) into the operator “-“, “/” as well as the ISignedFixedPoint datatype triumphantly.

- **Other IPrimitives**

For the other IPrimitives in LabVIEW Communications, I choose ExSelectPrimitive, ExDecrementPrimitive, ExIncrementPrimitive, ExAbsoluteValuePrimitive, ExSquarePrimitive, ExNegatePrimitive, ExMaxAndMinPrimitive to design the IPrimitive.gvi as Figure 4-8:

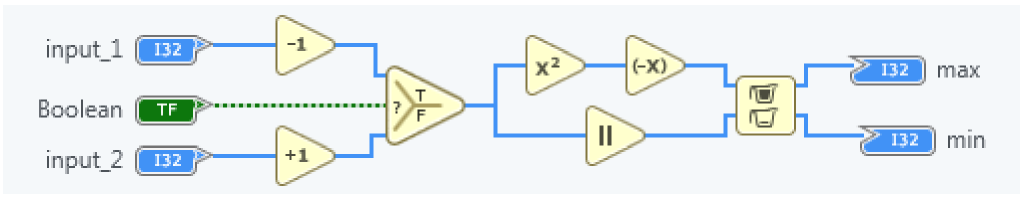
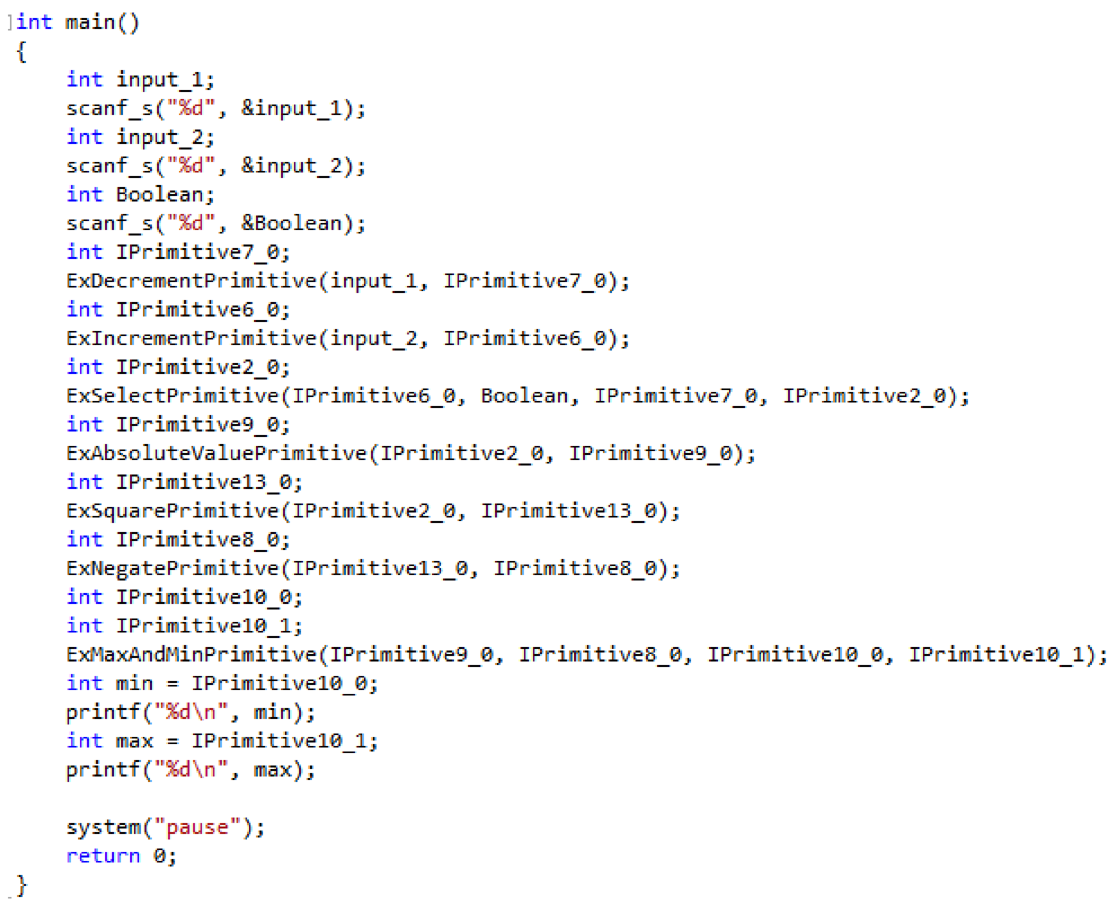


Figure 4-8 The IPrimitive.gvi in LabVIEW Communications

In the light of package function processing method, we can obtain the converted C code sample IPrimitive.c file as Figure 4-9:

```
#include<stdio.h>
#include<stdlib.h>
#include<cmath>
void ExDecrementPrimitive(int& x0, int& y0)
{
    y0 = x0 - 1;
}
void ExIncrementPrimitive(int& x0, int& y0)
{
    y0 = x0 + 1;
}
void ExSelectPrimitive(int& x0, int& x1, int& x2, int& y0)
{
    if (x1 == true)
        y0 = x2;
    else
        y0 = x0;
}
void ExAbsoluteValuePrimitive(int& x0, int& y0)
{
    y0 = abs(x0);
}
void ExSquarePrimitive(int& x0, int& y0)
{
    y0 = x0*x0;
}
void ExNegatePrimitive(int& x0, int& y0)
{
    y0 = -x0;
}
void ExMaxAndMinPrimitive(int& x0, int& x1, int& y0, int& y1)
{
    if (x0 >= x1) {
        y0 = x1;
        y1 = x0;
    }
    else {
        y0 = x0;
        y1 = x1;
    }
}
```



```
int main()
{
    int input_1;
    scanf_s("%d", &input_1);
    int input_2;
    scanf_s("%d", &input_2);
    int Boolean;
    scanf_s("%d", &Boolean);
    int IPrimitive7_0;
    ExDecrementPrimitive(input_1, IPrimitive7_0);
    int IPrimitive6_0;
    ExIncrementPrimitive(input_2, IPrimitive6_0);
    int IPrimitive2_0;
    ExSelectPrimitive(IPrimitive6_0, Boolean, IPrimitive7_0, IPrimitive2_0);
    int IPrimitive9_0;
    ExAbsoluteValuePrimitive(IPrimitive2_0, IPrimitive9_0);
    int IPrimitive13_0;
    ExSquarePrimitive(IPrimitive2_0, IPrimitive13_0);
    int IPrimitive8_0;
    ExNegatePrimitive(IPrimitive13_0, IPrimitive8_0);
    int IPrimitive10_0;
    int IPrimitive10_1;
    ExMaxAndMinPrimitive(IPrimitive9_0, IPrimitive8_0, IPrimitive10_0, IPrimitive10_1);
    int min = IPrimitive10_0;
    printf("%d\n", min);
    int max = IPrimitive10_1;
    printf("%d\n", max);

    system("pause");
    return 0;
}
```

Figure 4-9 The Converted C Code IPrimitive.c

As a result, I test plenty of data and list 5 groups of data randomly as the test sample to compare their running result as Table 4-11:

Table 4-11 The Result Comparison Between IPrimitive.gvi and IPrimitive.c

IPrimitive.gvi					IPrimitive.c				
input_1	input_2	Boolean	min	max	input_1	input_2	Boolean	min	max
3	-7	1	-4	2	3	-7	1	-4	2
3	-7	0	-36	6	3	-7	0	-36	6
66	2	1	-9	3	66	2	1	-9	3
-10	8	0	-81	9	-10	8	0	-81	9
5	-342	1	-16	4	5	-342	1	-16	4

The result comparison above validates the correctness of the packaged function in my compiler. And we can conclude that we can create the packaged function in C for every IPrimitive nodes as long as we these processing method.

4.4.3 IForLoop

To test the IForLoop processing method, I design the forloop.gvi in LabVIEW Communications as the Figure 4-10:

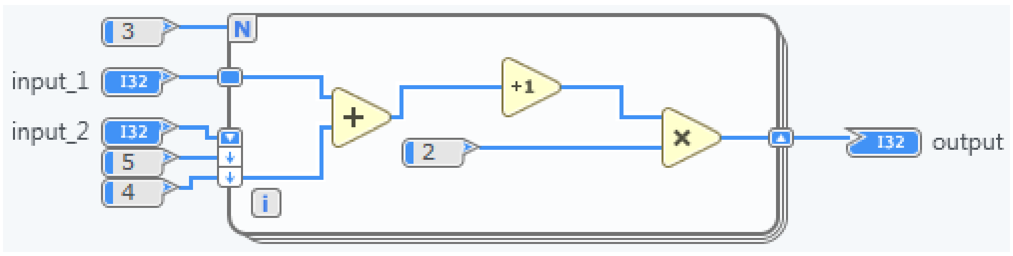


Figure 4-10 The IForLoop.gvi in LabVIEW Communications

In terms of the IForLoop processing method we talk about, we can obtain the converted C code sample IForLoop.c file as Figure 4-11:

```
#include<stdio.h>
#include<stdlib.h>
void ExIncrementPrimitive(int& x0, int& y0)
{
    y0 = x0 + 1;
}

int main()
{
    int input_2;
    scanf_s("%d", &input_2);
    int input_1;
    scanf_s("%d", &input_1);
    int ILoopMax5 = 3;
    int ILeftShiftRegister7_0 = input_2;
    int ILeftShiftRegister7_1 = 5;
    int ILeftShiftRegister7_2 = 4;
    int ITunnel6 = input_1;
    int IRightShiftRegister8;
    for (int ILoopIndex4 = 0; ILoopIndex4 < ILoopMax5; ILoopIndex4++) {
        int ICompoundArithmeticNode9 = ITunnel6 + ILeftShiftRegister7_2;
        int IPrimitive12_0;
        ExIncrementPrimitive(ICompoundArithmeticNode9, IPrimitive12_0);
        IRightShiftRegister8 = IPrimitive12_0 * 2;
        ILeftShiftRegister7_2 = ILeftShiftRegister7_1;
        ILeftShiftRegister7_1 = ILeftShiftRegister7_0;
        ILeftShiftRegister7_0 = IRightShiftRegister8;
    }
    int output = IRightShiftRegister8;
    printf("%d\n", output);

    system("pause");
    return 0;
}
```

Figure 4-11 The Converted Code IForLoop.c

As a result, I test plenty of data and list 5 groups of data randomly as the test sample to compare their running result as Table 4-12:

Table 4-12 The Result Comparison Between IForLoop.gvi and IForLoop.c

IForLoop.gvi			IForLoop.c		
input_1	input_2	output	input_1	input_2	output
4	5	20	4	5	20
-8	2	10	-8	2	10
36	90	254	36	90	254
-32	68	74	-32	68	74
1	0	4	1	0	4

According to the result comparison above, it can demonstrate the validity of the IForLoop processing method in my compiler, which means my compiler can also handle the border nodes, ITunnel, ILeftShiftRegister, IRightShiftRegister, ILoopMax, ILoopIndex triumphantly.

4.4.4 ICaseStructure

In LabVIEW Communications, I design four cases in the ICaseStructure.gvi as the Figure 4-12:

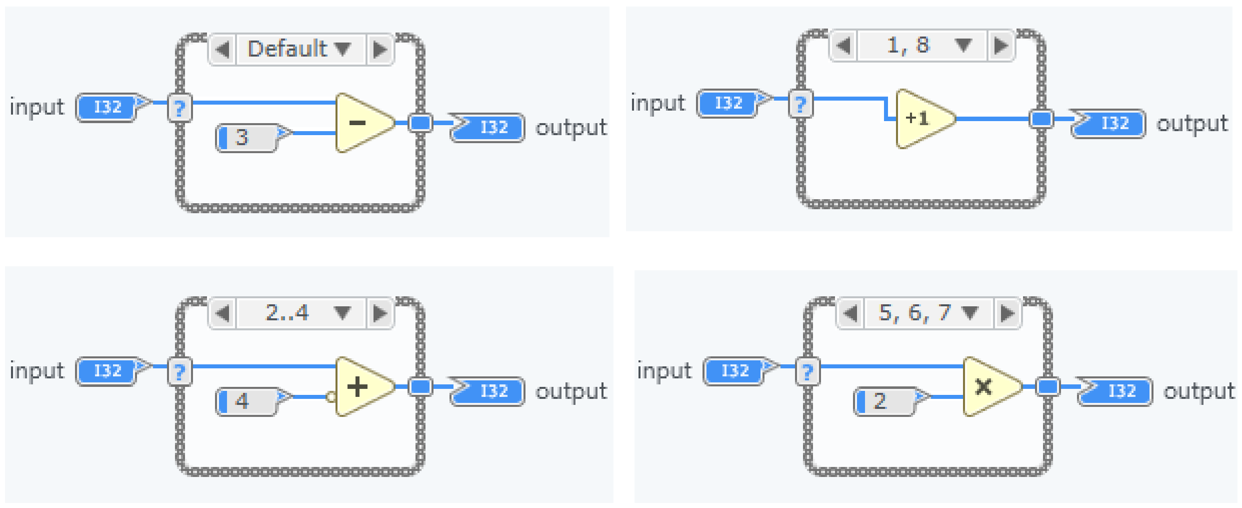


Figure 4-12 The ICaseStructure.gvi in LabVIEW Communications

On the principle of the ICaseStructure processing method we discuss above, we can obtain the converted C code sample ICaseStructure.c as Figure 4-13:

```
#include<stdio.h>
#include<stdlib.h>
void ExIncrementPrimitive(int& x0, int& y0)
{
    y0 = x0 + 1;
}

int main()
{
    int input;
    scanf_s("%d", &input);
    int ICaseSelector4 = input;
    int ITunnel8;
    if (ICaseSelector4 == 5 || ICaseSelector4 == 6 || ICaseSelector4 == 7)
    {
        ITunnel8 = ICaseSelector4 * 2;
    }
    else if (2 <= ICaseSelector4 <= 4)
    {
        ITunnel8 = ICaseSelector4 + (-4);
    }
    else if (ICaseSelector4 == 1 || ICaseSelector4 == 8)
    {
        int IPrimitive15_0;
        ExIncrementPrimitive(ICaseSelector4, IPrimitive15_0);
        ITunnel8 = IPrimitive15_0;
    }
    else
    {
        ITunnel8 = ICaseSelector4 - 3;
    }
    int output = ITunnel8;
    printf("%d\n", output);

    system("pause");
    return 0;
}
```

Figure 4-13 The Converted C Code ICaseStructure.c

As a result, I test plenty of data and list 5 groups of data randomly as the test sample to compare their running result as Table 4-13:

Table 4-13 The Result Comparison Between ICaseStructure.gvi and ICaseStructure.c

ICaseStructure.gvi		ICaseStructure.c	
input	output	input	output
0	-3	0	-3
1	2	1	2
3	-1	3	-1
5	10	5	10
7	14	7	14

Accordingly, the result comparison indicates the vadility of ICaseStructure processing method. And my compile can also handle the border node ICaseSelector.

To sum up, on the principle of the converted C code and running result comparison, we can conclude that my compiler can handle the ICompoundArithmeticNode, IPrimitive, IForLoop and ICaseStructure successfully.

4.5 The Summary of Chapter 4

To sum up, in Chapter 4, I mainly talked about the organization of my compiler and how it works. The whole procedure of my compiler can be summarized as the flow chart in the Figure 4-14:

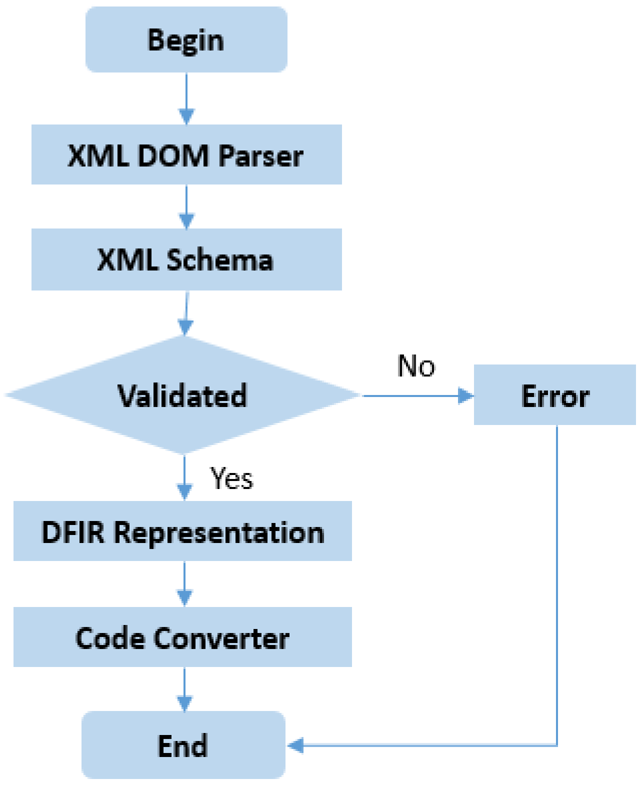


Figure 4-14 The Procedure of My Compiler

I use the built-in XML DOM parser in Microsoft’s .NET C# Framework to parse my XML file and store the information into a series of corresponding nodes classes in different .cs files. Furthermore, in terms of the topological sorting, I can obtain the sorted sequence of nodes in the diagram and then process them into generating C code in my compiler.

Chapter 5 Summary and Perspective

5.1 The Summary of My Research

In my research, I utilize the API provided by the National Instruments to access DFIR in LabVIEW Communications. DFIR can establish the connection between the diagram in LabVIEW Communications and other programming language. Accordingly, I learned how to design the FPGA diagram in LabVIEW Communications. At the same time, I grabbed DFIR’s organization and namespaces, which is necessary to develop my C# project to obtain the representation.

Correspondingly, I design the intermediate representation(IR) as XML file in my C# project to print out the information of all nodes by invoking the nodes’ properties and methods by DFIR. Meanwhile, I also design the corresponding XML schema, which can verify the validation of the XML file. It’s hard to overstate the significance of XML file and XML schema for developers, because they can utilize them to convert the diagram in LabVIEW into other programming language.

More importantly, with the help of the built-in XML DOM Parser in C# of Visual Studio, I can parse my XML file and obtain the representation information as a list of nodes’ classes. And according to the topological sorting ideas, I establish the code converter to sort, process the nodes’ classes and generate C code.

5.2 The Perspective of My Research

Undoubtedly, my compiler can still encounter some problems, hence it need some standard optimizations, such as dead/unreachable code elimination, in-line procedure calls, CSE elimination, strength reduction, constant propagation and so on. If I can figure out all of these optimization, my compiler will become much more effective and efficient and handle more situations.

Last but not least, my research provides the ideas about how to convert the graphical programming language into text-based programming language. It’s very efficient and convenient for those developers, which equip them the ability to take control of the diagram without learning this programing language. Thence, under the development of this thought, I predict that the converter tools between two different programming languages will be more and more professional and proficient in the future!

References

[1] Lee, S.H.; Barnwell, T.P., TOPOLOGICAL SORTING AND LOOP CLEANSING ALGORITHM FOR A CONSTRAINED MIMD COMPILER OF SHIFT-INVARIANT FigureS, 1986

[2] Toda, Seinosuke, On the complexity of topological sorting, Aug 28 1990

[3] Saha, A.R.; Mazumder, B.C. ,SUCCESSIVE APPROXIMATION FREQUENCY TO CODE CONVERTER WITH A MINIMUM SYSTEM MICROPROCESSOR, 1987

[4] Chang, C.-N.; Teng, H.-K.; Chen, J.-Y.; Chiu, H.-J., Computerized conducted EMI filter design system using LabVIEW and its application, National Science Council, 2001

[5] Tucker Kevin; Solomon Eli; Littlejohn Kenneth, Compiler optimization and its impact on development of real-time systems, Proceedings of the 1998 17th AIAA/IEEE/SAE Digital Avionics Systems Conference, DASC. Part 1 (of 2), October 31, 1998 - November 7, 1998

[6] Hatcher, P.J., Equational specification of efficient compiler code generation, International Conference on Computer Languages - ICCL '88 Part 1 (of 2), October 1, 1988 - October 1, 1988

[7] Jantz, Michael R., Kulkarni, Prasad A., Analyzing and addressing false interactions during compiler optimization phase ordering, 2014

[8] Wu, Ai Hua, Paquet, Joey, The translator generation in the general intensional programming compiler, CSCWD 2004 - 8th International Conference on Computer Supported Cooperative Work in Design - Proceedings, May 26, 2004 - May 28, 2004

[9] Iimuro, Satoshi, Sugino, Nobuhiko, Nishihara, Akinori, Fujii, Nobuo, Code optimization method utilizing memory addressing operation and its application to DSP compiler, Proceedings of the 1994 IEEE Asia-Pacific Conference on Circuits and Systems, December 5, 1994 - December 8, 1994

[10] Lee, Seong-Won, Moon, Soo-Mook, Jung, Won-Ki, Jin, Code size and performance optimization for mobile JavaScript just-in-time compiler, 2010 Workshop on Interaction between Compilers and Computer Architecture, INTERACT-14, March 13, 2010 - March 13, 2010

Gratitude

I’m so grateful to Professor Brent Nelson and Professor Dong Zhang. Thanks to their meticulous direction, I can complete my thesis successfully. I appreciate their earnest attitude of tutoring and scientific ways of working, which have a significant influence on me. And they concerned about both my study and life drastically. Besides, I’m so thankful that Professor Nelson can provide many precious suggestions for both my thesis and project, which enabled me to fulfill them smoothly. And I’m grateful for Professor Zhang who gave us such a fantastic chance to study abroad and experience different life.

Furthermore, when I worked in the ECE Lab of Brigham Young University, I want to express my gratitude to my research assistant, Thomas Townsend, and my teammate, Wayne Wang. They offered me so much enthusiastic flavor when I was stuck by my code.

In addition, I’m so thankful for the understanding and support of my family. Thanks to their agreement and subsidization for me, I can experience the American culture and finish my graduate project in BYU abroad.

Jiaxin Chen

10th, April, 2016

Appendix

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:complexType name="ElementType">
    <xs:sequence>
        <xs:element name="IDataType" type="IDataTypeType"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="ISignedIntType">
    <xs:sequence>
        <xs:element type="xs:byte" name="WordLength"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="IUnsignedIntType">
    <xs:sequence>
        <xs:element type="xs:byte" name="WordLength"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="ISignedFixedPointType">
    <xs:sequence>
        <xs:element type="xs:byte" name="LeftLength"/>
        <xs:element type="xs:byte" name="RightLength"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="IUnsignedFixedPointType">
    <xs:sequence>
        <xs:element type="xs:byte" name="LeftLength"/>
        <xs:element type="xs:byte" name="RightLength"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="IArrayType">
    <xs:sequence>
        <xs:element type="xs:byte" name="Dimensions"/>
        <xs:element name="Element" type="ElementType"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="IVariableSizedArrayType">
    <xs:sequence>
        <xs:element type="xs:byte" name="Dimensions"/>
        <xs:element name="Element" type="ElementType"/>
        </xs:sequence>
</xs:complexType>
<xs:complexType name="IComplexType">
        <xs:sequence>
                <xs:element name="Element" type="ElementType"/>
        </xs:sequence>
</xs:complexType>
<xs:complexType name="IFixedSizeArrayType">
        <xs:sequence>
                <xs:element type="xs:byte" name="Dimensions"/>
                <xs:element type="xs:byte" name="ArraySize"/>
                <xs:element name="Element" type="ElementType"/>
        </xs:sequence>
</xs:complexType>
<xs:complexType name="IDataTypeType">
        <xs:choice>
                <xs:element name="ISignedInt" type="ISignedIntType"/>
                <xs:element name="IUnsignedInt" type="IUnsignedIntType"/>
                <xs:element name="ISignedFixedPoint" type="ISignedFixedPointType"/>
                <xs:element name="IUnsignedFixedPoint" type="IUnsignedFixedPointType"/>
                <xs:element name="IArray" type="IArrayType"/>
                <xs:element name="IVariableSizedArray" type="IVariableSizedArrayType"/>
                <xs:element name="IComplex" type="IComplexType"/>
                <xs:element name="IFixedSizeArray" type="IFixedSizeArrayType"/>
                <xs:element name="IBit"/>
                <xs:element name="IBoolean"/>
                <xs:element name="IDouble"/>
                <xs:element name="IIncorrect" />
                <xs:element name="ISingle" />
                <xs:element name="IString" />
                <xs:element name="IUnknown" />
                <xs:element name="IUnsupported" />
                <xs:element name="IVoid"/>
        </xs:choice>
</xs:complexType>
<xs:complexType name="ConnectionType">
        <xs:simpleContent>
                <xs:extension base="xs:string">
                        <xs:attribute type="xs:byte" name="TerminalId" use="required"/>
                        <xs:attribute type="xs:byte" name="NodeId" use="required"/>
                </xs:extension>
        </xs:simpleContent>
</xs:complexType>
<xs:complexType name="ConnectionsType">
```

```
        <xs:sequence>
                <xs:element name="Connection" type="ConnectionType" minOccurs="0"
maxOccurs="unbounded"/>
        </xs:sequence>
</xs:complexType>
<xs:complexType name="ITerminalType">
        <xs:sequence>
                <xs:element name="IDataType" type="IDataTypeType"/>
                <xs:element name="Connections"    type="ConnectionsType"/>
        </xs:sequence>
        <xs:attribute type="xs:byte" name="TerminalId" use="required"/>
        <xs:attribute type="xs:byte" name="TerminalIndex" use="required"/>
</xs:complexType>
<xs:complexType name="InputTerminalsType">
        <xs:sequence>
                <xs:element name="ITerminal" type="ITerminalType" maxOccurs="unbounded"
minOccurs="1"/>
        </xs:sequence>
</xs:complexType>
<xs:complexType name="OutputTerminalsType">
        <xs:sequence>
                <xs:element name="ITerminal" type="ITerminalType" maxOccurs="unbounded"
minOccurs="1"/>
        </xs:sequence>
</xs:complexType>
<xs:complexType name="IDataAccessorType">
        <xs:sequence>
                <xs:element type="xs:string" name="Name"/>
                <xs:element type="direction" name="Direction"/>
                <xs:choice minOccurs="0">
                        <xs:element name="InputTerminals" type="InputTerminalsType"/>
                        <xs:element name="OutputTerminals" type="OutputTerminalsType"/>
                </xs:choice>
        </xs:sequence>
        <xs:attribute type="xs:byte" name="NodeId" use="required"/>
        <xs:attribute type="xs:byte" name="ParentId" use="required"/>
</xs:complexType>
<xs:simpleType name="direction">
        <xs:restriction base="xs:string">
                <xs:pattern value="INPUT|OUTPUT"/>
        </xs:restriction>
</xs:simpleType>
<xs:complexType name="RangeType" >
        <xs:choice>
```

```
            <xs:sequence>
                <xs:element type="xs:byte" name="LowValue" />
                <xs:element type="xs:byte" name="HighValue" />
            </xs:sequence>
            <xs:element type="xs:byte" name="SingleValue"/>
        </xs:choice>
        <xs:attribute type="xs:byte" name="DiagramIndex" use="required"/>
</xs:complexType>
<xs:complexType name="RangesType">
        <xs:sequence>
            <xs:element name="Range" type="RangeType" maxOccurs="unbounded"
minOccurs="0"/>
        </xs:sequence>
</xs:complexType>
<xs:complexType name="ICaseSelectorType">
        <xs:sequence>
            <xs:element name="DefaultDiagramIndex">
            <xs:complexType>
                <xs:simpleContent>
                    <xs:extension base="xs:string">
                        <xs:attribute type="xs:byte" name="DiagramIndex"/>
                    </xs:extension>
                </xs:simpleContent>
            </xs:complexType>
            </xs:element>
            <xs:element name="Ranges"   type="RangesType"/>
            <xs:element name="InputTerminals" type="InputTerminalsType" minOccurs="0"/>
            <xs:element name="OutputTerminals" type="OutputTerminalsType"
minOccurs="0"/>
        </xs:sequence>
        <xs:attribute type="xs:byte" name="NodeId" use="required"/>
        <xs:attribute type="xs:byte" name="ParentId" use="required"/>
</xs:complexType>
<xs:complexType name="ITunnelType">
        <xs:sequence>
            <xs:element type="xs:string" name="IsInput"/>
            <xs:element name="GetInnerTerminal">
                <xs:complexType>
                    <xs:attribute type="xs:byte" name="TerminalId"/>
                </xs:complexType>
            </xs:element>
            <xs:element name="GetOuterTerminal">
                <xs:complexType>
                    <xs:attribute type="xs:byte" name="TerminalId"/>
                </xs:complexType>
            </xs:element>
            <xs:element type="IndexingModeType" name="IndexingMode"/>
            <xs:element name="InputTerminals" type="InputTerminalsType" minOccurs="0"/>
            <xs:element name="OutputTerminals" type="OutputTerminalsType"
minOccurs="0"/>
        </xs:sequence>
        <xs:attribute type="xs:byte" name="NodeId" use="required"/>
        <xs:attribute type="xs:byte" name="ParentId" use="required"/>
</xs:complexType>
<xs:simpleType name="IndexingModeType">
        <xs:restriction base="xs:string">
            <xs:pattern value="NonIndexing|Indexing"/>
        </xs:restriction>
</xs:simpleType>
<xs:complexType name="ICaseStructureType">
        <xs:sequence>
            <xs:element name="ICaseSelector" type="ICaseSelectorType"/>
            <xs:element name="ITunnel" type="ITunnelType" maxOccurs="unbounded"
minOccurs="0"/>
            <xs:element ref="IDiagram" maxOccurs="unbounded" minOccurs="0"/>
        </xs:sequence>
        <xs:attribute type="xs:byte" name="NodeId" use="required"/>
        <xs:attribute type="xs:byte" name="ParentId" use="required"/>
</xs:complexType>
<xs:complexType name="IConstantType">
        <xs:sequence>
            <xs:element type="xs:string" name="Value"/>
            <xs:element name="IDataType" type="IDataTypeType"/>
            <xs:element name="OutputTerminals" type="OutputTerminalsType"
minOccurs="0"/>
        </xs:sequence>
        <xs:attribute type="xs:byte" name="NodeId" use="required"/>
        <xs:attribute type="xs:byte" name="ParentId" use="required"/>
</xs:complexType>
<xs:complexType name="IBlackBoxNodeType">
         <xs:sequence>
            <xs:element name="InputTerminals" type="InputTerminalsType" minOccurs="0"/>
            <xs:element name="OutputTerminals" type="OutputTerminalsType"
minOccurs="0"/>
        </xs:sequence>
        <xs:attribute type="xs:byte" name="NodeId" use="required"/>
        <xs:attribute type="xs:byte" name="ParentId" use="required"/>
</xs:complexType>
```

```
<xs:complexType name="PairType">
     <xs:simpleContent>
        <xs:extension base="xs:string">
            <xs:attribute type="xs:byte" name="TerminalId" use="required"/>
            <xs:attribute type="xs:byte" name="SubVIDataAccessorId" use="required"/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>
<xs:complexType name="SubVIType">
    <xs:sequence>
        <xs:element name="Pair" type="PairType" maxOccurs="unbounded"
minOccurs="0"/>
    </xs:sequence>
    <xs:attribute type="xs:byte" name="NodeId" use="required"/>
    <xs:attribute type="xs:byte" name="ParentId" use="required"/>
</xs:complexType>
<xs:complexType name="IMethodCallType">
    <xs:sequence>
        <xs:element type="xs:string" name="TargetName"/>
        <xs:element name="SubVI" type="SubVIType"/>
        <xs:element name="InputTerminals" type="InputTerminalsType" minOccurs="0"/>
        <xs:element name="OutputTerminals" type="OutputTerminalsType"
minOccurs="0"/>
    </xs:sequence>
    <xs:attribute type="xs:byte" name="NodeId" use="required"/>
    <xs:attribute type="xs:byte" name="ParentId" use="required"/>
</xs:complexType>
<xs:complexType name="InitTerminalType">
    <xs:simpleContent>
        <xs:extension base="xs:string">
            <xs:attribute type="xs:byte" name="TerminalId" use="required"/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>
<xs:complexType name="InputNodeType">
    <xs:simpleContent>
        <xs:extension base="xs:string">
            <xs:attribute type="xs:byte" name="NodeId" use="required"/>
            <xs:attribute type="xs:byte" name="ParentId" use="required"/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>
<xs:complexType name="OutputNodeType">
    <xs:simpleContent>
```

```
        <xs:extension base="xs:string">
            <xs:attribute type="xs:byte" name="NodeId" use="required"/>
            <xs:attribute type="xs:byte" name="ParentId" use="required"/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>
<xs:complexType name="IFeedbackOutputNodeType">
    <xs:sequence>
        <xs:element name="InitTerminal" type="InitTerminalType"/>
        <xs:element name="InputNode" type="InputNodeType"/>
        <xs:element name="InputTerminals" type="InputTerminalsType" minOccurs="0"/>
        <xs:element name="OutputTerminals" type="OutputTerminalsType"
minOccurs="0"/>
    </xs:sequence>
    <xs:attribute type="xs:byte" name="NodeId" use="required"/>
    <xs:attribute type="xs:byte" name="ParentId" use="required"/>
</xs:complexType>
<xs:complexType name="FeedbackTerminalType">
    <xs:simpleContent>
        <xs:extension base="xs:string">
            <xs:attribute type="xs:byte" name="TerminalId"/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>
<xs:complexType name="IFeedbackInputNodeType">
    <xs:sequence>
        <xs:element type="xs:byte" name="Delay"/>
        <xs:element name="FeedbackTerminal" type="FeedbackTerminalType"
minOccurs="0"/>
        <xs:element name="OutputNode" type="OutputNodeType"/>
        <xs:element name="InputTerminals" type="InputTerminalsType" minOccurs="0"/>
        <xs:element name="OutputTerminals" type="OutputTerminalsType"
minOccurs="0"/>
    </xs:sequence>
    <xs:attribute type="xs:byte" name="NodeId" use="required"/>
    <xs:attribute type="xs:byte" name="ParentId" use="required"/>
</xs:complexType>
<xs:complexType name="ICompoundArithmeticNodeType">
    <xs:sequence>
        <xs:element name="InvertedInputs" type="InvertedInputsType"/>
        <xs:element name="InvertedOutput" type="xs:string"/>
        <xs:element name="InputTerminals" type="InputTerminalsType" minOccurs="0"/>
        <xs:element name="OutputTerminals" type="OutputTerminalsType"
minOccurs="0"/>
```

```
	</xs:sequence>
	<xs:attribute type="xs:byte" name="NodeId" use="required"/>
	<xs:attribute type="xs:byte" name="ParentId" use="required"/>
	<xs:attribute type="compoundMode" name="Mode" use="required"/>
</xs:complexType>
<xs:complexType name="InvertedInputsType">
	<xs:sequence>
		<xs:element name="InvertedInput" type="xs:string" maxOccurs="unbounded"/>
	</xs:sequence>
</xs:complexType>
<xs:simpleType name="compoundMode">
	<xs:restriction base="xs:string">
		<xs:pattern value="Add|Multiply|Or|And|Xor"/>
	</xs:restriction>
</xs:simpleType>
<xs:complexType name="IRightShiftRegisterType">
	<xs:sequence>
		<xs:element name="AssociatedLeftShiftRegister">
			<xs:complexType>
				<xs:simpleContent>
					<xs:extension base="xs:string">
						<xs:attribute type="xs:byte" name="NodeId"/>
						<xs:attribute type="xs:byte" name="ParentId"/>
					</xs:extension>
					</xs:simpleContent>
				</xs:complexType>
		</xs:element>
		<xs:element name="InputTerminals" type="InputTerminalsType" minOccurs="0"/>
	<xs:element name="OutputTerminals" type="OutputTerminalsType" minOccurs="0"/>
	</xs:sequence>
	<xs:attribute type="xs:byte" name="NodeId" use="required"/>
	<xs:attribute type="xs:byte" name="ParentId" use="required"/>
</xs:complexType>
<xs:complexType name="ILeftShiftRegisterType">
	<xs:sequence>
		<xs:element name="AssociatedRightShiftRegister">
			<xs:complexType>
				<xs:simpleContent>
					<xs:extension base="xs:string">
						<xs:attribute type="xs:byte" name="NodeId"/>
						<xs:attribute type="xs:byte" name="ParentId"/>
					</xs:extension>
				</xs:simpleContent>
			</xs:complexType>
```

```
            </xs:element>
            <xs:element name="InputTerminals" type="InputTerminalsType" minOccurs="0"/>
            <xs:element name="OutputTerminals" type="OutputTerminalsType"
minOccurs="0"/>
        </xs:sequence>
        <xs:attribute type="xs:byte" name="NodeId" use="required"/>
        <xs:attribute type="xs:byte" name="ParentId" use="required"/>
</xs:complexType>
<xs:group name="shiftRegister">
        <xs:sequence>
            <xs:element name="ILeftShiftRegister" type="ILeftShiftRegisterType"/>
            <xs:element name="IRightShiftRegister" type="IRightShiftRegisterType"/>
        </xs:sequence>
</xs:group>
<xs:complexType name="IForLoopType">
        <xs:sequence>
            <xs:element name="ILoopIndex" type="ILoopIndexType" maxOccurs="unbounded"
minOccurs="0"/>
            <xs:element name="ILoopMax" type="ILoopMaxType" maxOccurs="unbounded"
minOccurs="0"/>
            <xs:element name="ITunnel" type="ITunnelType" maxOccurs="unbounded"
minOccurs="0"/>
            <xs:group ref="shiftRegister" maxOccurs="unbounded" minOccurs="0"/>
        <xs:element ref="IDiagram"/>
        </xs:sequence>
        <xs:attribute type="xs:byte" name="NodeId" use="required"/>
        <xs:attribute type="xs:byte" name="ParentId" use="required"/>
</xs:complexType>
<xs:complexType name="IWhileLoopType">
        <xs:sequence>
            <xs:element name="ILoopCondition" type="ILoopConditionType"
maxOccurs="unbounded" minOccurs="0"/>
            <xs:element name="ILoopIndex" type="ILoopIndexType" maxOccurs="unbounded"
minOccurs="0"/>
            <xs:element name="ITunnel" type="ITunnelType" maxOccurs="unbounded"
minOccurs="0"/>
            <xs:group ref="shiftRegister" maxOccurs="unbounded" minOccurs="0"/>
            <xs:element ref="IDiagram"/>
        </xs:sequence>
            <xs:attribute type="xs:byte" name="NodeId" use="required"/>
            <xs:attribute type="xs:byte" name="ParentId" use="required"/>
</xs:complexType>
<xs:complexType name="ILoopIndexType">
        <xs:sequence>
```

```
            <xs:element name="InputTerminals" type="InputTerminalsType" minOccurs="0"/>
            <xs:element name="OutputTerminals" type="OutputTerminalsType"
minOccurs="0"/>
      </xs:sequence>
      <xs:attribute type="xs:byte" name="NodeId" use="required"/>
      <xs:attribute type="xs:byte" name="ParentId" use="required"/>
</xs:complexType>
<xs:complexType name="ILoopMaxType">
      <xs:sequence>
            <xs:element name="InputTerminals" type="InputTerminalsType" minOccurs="0"/>
            <xs:element name="OutputTerminals" type="OutputTerminalsType"
minOccurs="0"/>
      </xs:sequence>
      <xs:attribute type="xs:byte" name="NodeId" use="required"/>
      <xs:attribute type="xs:byte" name="ParentId" use="required"/>
</xs:complexType>
<xs:complexType name="ILoopConditionType">
      <xs:sequence>
            <xs:element name="InputTerminals" type="InputTerminalsType" minOccurs="0"/>
            <xs:element name="OutputTerminals" type="OutputTerminalsType"
minOccurs="0"/>
      </xs:sequence>
      <xs:attribute type="xs:byte" name="NodeId" use="required"/>
      <xs:attribute type="xs:byte" name="ParentId" use="required"/>
</xs:complexType>
<xs:complexType name="IArrayIndexNodeType">
      <xs:sequence>
            <xs:element name="InputTerminals" type="InputTerminalsType" minOccurs="0"/>
            <xs:element name="OutputTerminals" type="OutputTerminalsType"
minOccurs="0"/>
      </xs:sequence>
      <xs:attribute type="xs:byte" name="NodeId" use="required"/>
      <xs:attribute type="xs:byte" name="ParentId" use="required"/>
</xs:complexType>
<xs:complexType name="IReplaceArraySubsetNodeType">
      <xs:sequence>
            <xs:element name="InputTerminals" type="InputTerminalsType" minOccurs="0"/>
            <xs:element name="OutputTerminals" type="OutputTerminalsType"
minOccurs="0"/>
      </xs:sequence>
      <xs:attribute type="xs:byte" name="NodeId" use="required"/>
      <xs:attribute type="xs:byte" name="ParentId" use="required"/>
</xs:complexType>
<xs:complexType name="IInsertIntoArrayNodeType">
```

```
        <xs:sequence>
                <xs:element name="InputTerminals" type="InputTerminalsType" minOccurs="0"/>
                <xs:element name="OutputTerminals" type="OutputTerminalsType"
minOccurs="0"/>
        </xs:sequence>
        <xs:attribute type="xs:byte" name="NodeId" use="required"/>
        <xs:attribute type="xs:byte" name="ParentId" use="required"/>
</xs:complexType>
<xs:complexType name="IDeleteFromArrayNodeType">
        <xs:sequence>
                <xs:element name="InputTerminals" type="InputTerminalsType" minOccurs="0"/>
                <xs:element name="OutputTerminals" type="OutputTerminalsType"
minOccurs="0"/>
        </xs:sequence>
        <xs:attribute type="xs:byte" name="NodeId" use="required"/>
        <xs:attribute type="xs:byte" name="ParentId" use="required"/>
</xs:complexType>
<xs:complexType name="IInitializeArrayNodeType">
        <xs:sequence>
                <xs:element name="InputTerminals" type="InputTerminalsType" minOccurs="0"/>
                <xs:element name="OutputTerminals" type="OutputTerminalsType"
minOccurs="0"/>
        </xs:sequence>
        <xs:attribute type="xs:byte" name="NodeId" use="required"/>
        <xs:attribute type="xs:byte" name="ParentId" use="required"/>
</xs:complexType>
<xs:complexType name="IBuildArrayNodeType">
        <xs:sequence>
                <xs:element name="InputTerminals" type="InputTerminalsType" minOccurs="0"/>
                <xs:element name="OutputTerminals" type="OutputTerminalsType"
minOccurs="0"/>
        </xs:sequence>
        <xs:attribute type="xs:byte" name="NodeId" use="required"/>
        <xs:attribute type="xs:byte" name="ParentId" use="required"/>
</xs:complexType>
<xs:complexType name="IArraySubsetNodeType">
        <xs:sequence>
                <xs:element name="InputTerminals" type="InputTerminalsType" minOccurs="0"/>
                <xs:element name="OutputTerminals" type="OutputTerminalsType"
minOccurs="0"/>
        </xs:sequence>
        <xs:attribute type="xs:byte" name="NodeId" use="required"/>
        <xs:attribute type="xs:byte" name="ParentId" use="required"/>
</xs:complexType>
```

```
<xs:complexType name="IPrimitiveType">
        <xs:sequence>
                <xs:element name="InputTerminals" type="InputTerminalsType" minOccurs="0"/>
                <xs:element name="OutputTerminals" type="OutputTerminalsType"
minOccurs="0"/>
        </xs:sequence>
        <xs:attribute type="xs:byte" name="NodeId" use="required"/>
        <xs:attribute type="xs:byte" name="ParentId" use="required"/>
        <xs:attribute type="primitiveMode" name="Mode" use="required"/>
</xs:complexType>
<xs:simpleType name="primitiveMode">
        <xs:restriction base="xs:string">
                <xs:pattern value="Ex[a-zA-Z0-9]+Primitive"/>
        </xs:restriction>
</xs:simpleType>
<xs:complexType name="INodeType">
        <xs:sequence>
                <xs:element name="InputTerminals" type="InputTerminalsType" minOccurs="0"/>
                <xs:element name="OutputTerminals" type="OutputTerminalsType"
minOccurs="0"/>
        </xs:sequence>
        <xs:attribute type="xs:byte" name="NodeId" use="required"/>
        <xs:attribute type="xs:byte" name="ParentId" use="required"/>
        <xs:attribute type="xs:string" name="Mode" use="required"/>
</xs:complexType>
<xs:element name="IDiagram" type="IDiagramType"/>
  <xs:complexType name="IDiagramType">
    <xs:choice maxOccurs="unbounded" minOccurs="0">
                        <xs:element name="ICompoundArithmeticNode"
type="ICompoundArithmeticNodeType" maxOccurs="unbounded" minOccurs="0"/>
                        <xs:element name="IDataAccessor" type="IDataAccessorType"
maxOccurs="unbounded" minOccurs="0"/>
                        <xs:element name="IConstant" type="IConstantType"
maxOccurs="unbounded" minOccurs="0"/>
                        <xs:element name="IForLoop" type="IForLoopType" maxOccurs="unbounded"
minOccurs="0"/>
                        <xs:element name="IWhileLoop" type="IWhileLoopType"
maxOccurs="unbounded" minOccurs="0"/>
                        <xs:element name="ICaseStructure" type="ICaseStructureType"
maxOccurs="unbounded" minOccurs="0"/>
                        <xs:element name="IMethodCall" type="IMethodCallType"
maxOccurs="unbounded" minOccurs="0"/>
                        <xs:element name="IPrimitive" type="IPrimitiveType"
maxOccurs="unbounded" minOccurs="0"/>
```

```
                    <xs:element name="IFeedbackInputNode" type="IFeedbackInputNodeType"
maxOccurs="unbounded" minOccurs="0"/>
                    <xs:element name="IFeedbackOutputNode"
type="IFeedbackOutputNodeType" maxOccurs="unbounded" minOccurs="0"/>
                    <xs:element name="IBlackBoxNode" type="IBlackBoxNodeType"
maxOccurs="unbounded" minOccurs="0"/>
                    <xs:element name="IArrayIndexNode" type="IArrayIndexNodeType"
maxOccurs="unbounded" minOccurs="0"/>
                    <xs:element name="IReplaceArraySubsetNode"
type="IReplaceArraySubsetNodeType" maxOccurs="unbounded" minOccurs="0"/>
                    <xs:element name="IInsertIntoArrayNode" type="IInsertIntoArrayNodeType"
maxOccurs="unbounded" minOccurs="0"/>
                    <xs:element name="IDeleteFromArrayNode"
type="IDeleteFromArrayNodeType" maxOccurs="unbounded" minOccurs="0"/>
                    <xs:element name="IInitializeArrayNode" type="IInitializeArrayNodeType"
maxOccurs="unbounded" minOccurs="0"/>
                    <xs:element name="IBuildArrayNode" type="IBuildArrayNodeType"
maxOccurs="unbounded" minOccurs="0"/>
                    <xs:element name="IArraySubsetNode" type="IArraySubsetNodeType"
maxOccurs="unbounded" minOccurs="0"/>
   </xs:choice>
             <xs:attribute type="xs:byte" name="NodeId"/>
            <xs:attribute type="xs:byte" name="ParentId"/>
            <xs:attribute type="xs:byte" name="DiagramIndex"/>
  </xs:complexType>
</xs:schema>
```