MODEL-BASED DESIGN OF GARBLED CIRCUITS

By

Weijie Yu

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

August, 2015

Nashville, Tennessee

Approved:

Professor Bradley A. Malin

Professor Christopher J. White

ACKNOWLEDGMENTS

Firstly, I would like to express my sincere gratitude to my adviser, Professor Malin, for his continuous support of my thesis study. This thesis would not have been possible without his patience, motivation, and knowledge. His guidance helped me in all the time of writing of this thesis. I was touched by every feedback he sent to me, which was always full of comments and corrections, not missing even a punctuation problem. I could not have imagined having a better adviser for my M.S. study.

Besides my adviser, I would like to thank the second reader of my thesis, Professor White, for his insightful reviews in improving the thesis, but also for the hard question which widened my thesis from various perspectives.

My sincere thanks also goes to Wei Xie, who provided me this great idea and opportunity to start the thesis and enlightened me the first glance of research.

Last but not the least, I would like to thank my parents for their moral support and warm encouragements throughout writing this thesis. I also want to thank people in Nashville Chinese Baptism Church for their prayers. This two-year international study experience makes me know more about the significance of love.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS i

LIST OF TABLES iv

LIST OF FIGURES v

Chapter

I INTRODUCTION 1

I.1 Motivation 1
I.2 Goal 3

II PRELIMINARIES 5

II.1 Garbled Circuits 5
II.2 Garbled Circuit Implementations 6
II.3 Domain Specific Modeling Language 7
II.3.1 Advantages of DSML On GC Specification 8
II.4 Summary 9

III DSML DESIGN APPROACHES 10

III.1 Design Goal 10
III.2 Design Assumptions 10
III.3 Design Pattern 11
III.4 Metamodel Vocabulary 12
III.5 Connections in GC 13
III.6 Transformations from Models to Codes 15
III.6.1 Declaration of GC 15
III.6.2 Initialization of Sub-circuit 15
III.6.3 Wires 16
III.6.4 Fixed Value For Wire 16
III.7 Summary 17

IV DSML IMPLEMENTATION ON GME PLATFORM 18

IV.1 GME: DSML Development Environment 18
IV.2 Advantages of GME 19
IV.3 GC Metamodel on GME 20
IV.4 Build Your Own GC Domain Models 20
IV.5 Circuit Library 23
IV.6 Summary 24

V FASTGC INTERPRETER 25
V.1 GC Implementation: FastGC 25
V.2 GC Code In FastGC 26
V.2.1 Constructor Function 26
V.2.2 Interfaces To Be Implemented 27
V.2.3 For Loop 30
V.3 Interpreter Interface 30
V.3.1 CodeWriter 31
V.3.2 CodeGenerate 31
V.3.3 Comments in Codes 35
V.3.4 circuitID Alias 35
V.4 Summary 36
VI DISCUSSION AND CONCLUSION 37
VI.1 Future Work 37
A Appendix Circuit Library 39
B Appendix Class InitialtorStatement Implementation 44
BIBLIOGRAPHY 46

LIST OF TABLES

Table Page

A.1 GC Library Level 1 40

A.2 GC Library Level 2 41

A.3 GC Library Level 3 42

A.4 GC Library Level 4 43

LIST OF FIGURES

Figure Page

I.1 An architectural diagram of the system. DSML is the first step to specify GC model design rules and configure the development environment. GC models can be designed in this environment and models can be put into an database as a library. GC models are transformed into working codes with given interpreters. 3

II.1 An illustration of the logic circuit for the hamming distance which is composed by a number of XOR gates and a Counter. XOR gates compare each bit pair from two integers which are at the same position. The Counter counts the number of positions where the corresponding bit from the two integers are different. 6

III.1 Scenario 1: The same connection pattern applied to multiple circuit pairs. Scenario 2: A splitting of large circuit to connect it with a group of small circuits. 14

IV.1 The process for building the development environment. 19

IV.2 A class diagram of a GC model that presents each module in the model. Circuit is the first class object and is the parent class of LogicalGate and GarbledCircuit. LogicalGate defines the three gates in a logic circuit, and has three subclasses AND, OR, and XOR. GarbledCircuit represents the GC model. It contains: 1) the sub-circuits used, which are represented by LogicalGate and GarbledCircuit, 2) CircuitRef and its subclass, which includes InitialtorRef, FixedWireRef and ConnectionRef, which perform the tasks of sub-circuit initialization, setting fixed values for wires and connections in GC model, respectively, and 3) Wire and its subclass, DefineInOutput, which defines the input and output, and InternalWire which specifies the connections between sub-circuits. . 21

IV.3 (a) GC model HAMMING_2L_K accepts arguments L and K and has 2L input ports and K output ports. In the model, there are L XOR gates and one counter. (b) initialization of L paralleling XOR gates which are aligned from index 0 to index L-1. (c)how input is defined by the connection Input module with *ReplicateRef* of the XOR gates. The last part of L ports on Input module is split into L pieces to connect with L XOR gates. (d) attribute set for connecting Input and XOR gates. 22

V.1 The constructor function of Hamming distance circuit. 27

V.2 The implementation of the build function. 27

V.3 The initialization of the subcircuits in the FastGC code. 28

V.4 The initialization of the And gate and the Or gate in the FastGC code. . . 28

V.5 The definition of the input in FastGC code. 29

V.6 The definition of the connections between the subcircuits in FastGC code. 29

V.7 The definition of the output in FastGC code. 30

V.8 The specification of fixed value in FastGC code. 30

V.9 The for loop code template. 31

V.10 CodeWriter: a namespace for classes of code templates. In this class diagram, the class Statement is the abstract class for all the code template classes that have an abstract function toString. The toString function generates the codes stored by each class object. Each code template has subclasses including i) InitialtorStatement, ii) ConnectionStatement, iii) FixedWireStatement, and iv) ForLoopStatement. The GateStatement is specifically designed for AND and OR gate initialization. The GarbledClassFunction stores the basic information for model and implements the constructor function. 32

V.11 CodeGenerator: an namespace for storing the data from GC models and transforming models into CodeWriter classes. In this class diagram, CodeGenerator is the abstract class which defines the interfaces and class member for subclasses. It has a Statement member which stores the transformation result. setStatement function executes the transformation an printStatement outputs the generated codes. CodeGenerator has subclasses including i) Initialtor, ii) FixedWire, iii) Connection, and iv) GarbledClass, each of which corresponds to a module in GC metamodel. 33

V.12 Component represents subcircuits and input/output that participate in a connection. It has two classes SingleComponent and MultipleComponent. In comparison to metamodel design, SingleComponent is designed for *SplitRef* in the meta-models that may split its ports into smaller pieces, while MultipleComponent is designed for *ReplicateRef* that they can replicate itself for a group. The Input and Output derives from SingleComponent because they can be split to connect with subcircuits. . . . 34

V.13 The circuitID alias for a single subcircuit. 35

V.14 The circuitID alias for multiple subcircuits. 36

CHAPTER I

INTRODUCTION

Secure multi-party computation (SMC) [1], which is also referred to as Secure Function Evaluation (SFE) [2], focuses on solving the problem of how two or multiple parties can jointly compute a function over their inputs without revealing their own private input. A famous instance of the secure computation is the millionaires problem [3]. In this situation, two millionaires want to know who is richer while keep the other from knowing their own worth. In this thesis, we introduce a strategy to simplify the implementation of secure computation protocols based on the increasingly popular garbled circuits technique. This chapter elaborates on the motivation and provides a high-level overview of the solution.

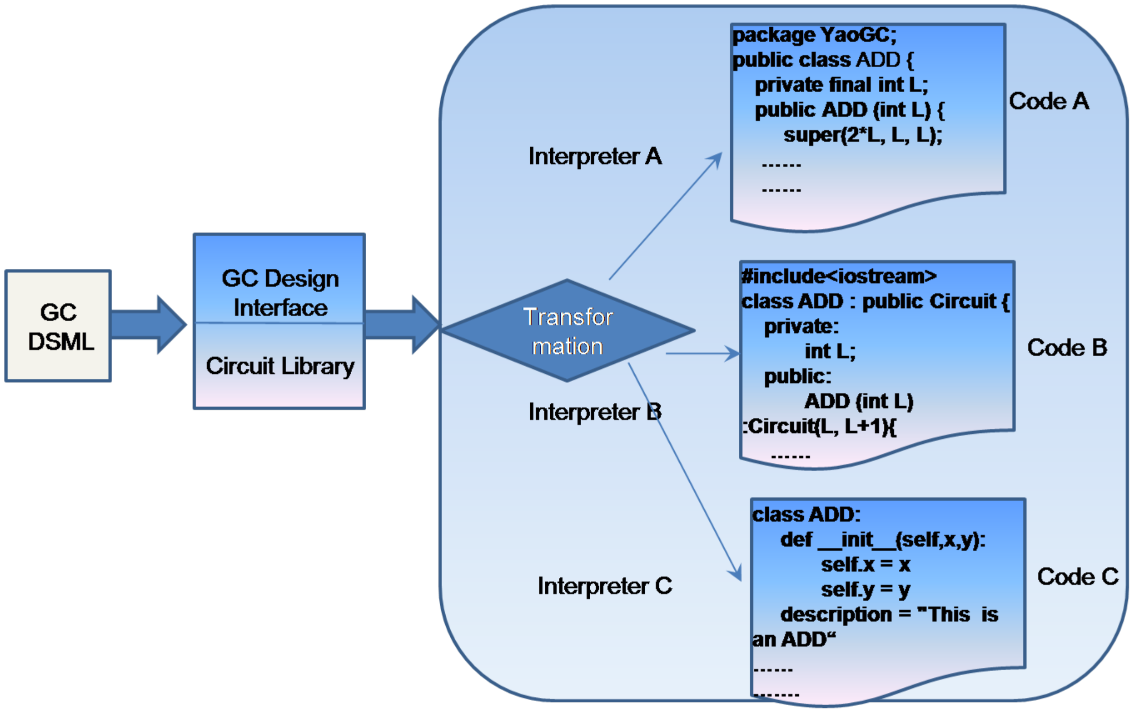
I.1 Motivation

Technically, secure multi-party computation problem focuses on ”how to compute any probabilistic function on any input, in a distributed network where each participant holds one of the inputs, ensuring independence of the inputs, correctness of the computation, and that no more information is revealed to a participant in the computation than can be inferred from that participants input and output” [4].

A garbled circuit [5] (GC) is one of main approaches for solving SMC problem. Informally, GCs transform arbitrary functions into secure functions by representing the function as a logic binary circuit which has the desired computation functionality. GCs have been operationalized through various numerous implementations, each of which are engineered to share the features that (1) build a computation framework, realized in an engine that executes secure protocols, and (2) enable developers to design custom protocols by transforming the functionality into a logic circuit representation. At the same time, each framework has its unique way of representing the logic circuit in operational code.

The logic circuit that represents certain computational functionality is determined by the functionality only, which is independent of the secure computation framework. However, when implementing the circuit functionality in code, the transformation from circuits to codes is specific to particular programming language and the construction of the framework. This becomes a problem when we want to execute the same GC across a variety of frameworks. More specifically, this problem arises because we need to write different codes for the same circuit to satisfy the implementation requirements of disparate frameworks. This problem is escalating in concern an increasing number of implementation are being introduced , such that simple existing GC code may not be sufficient for future compilation and (2) writing circuit codes is hard and debugging the code is complicated, considering that the developers may have to use a language they do not even know, or developers have to spend a lot of time in learning how the framework is built and execute the secure protocols.

This thesis is motivated by the need for a framework-independent approach to GC specification. Instead of writing codes for each framework, we propose designing and implementing a generic circuit representation in a manner that (1) is tailored to GCs and has a comprehensive vocabulary that it is capable of representing everything associated with a logic circuit, such as circuit components and wires, (2) is feasible for transformations from GCs into codes that work in frameworks to execute computation functionality GCs represent, and (3) has the flexibility to be extended and customized so that its generic representation meets the future requirements.



```
package YaoGC;
public class ADD {
   private final int L;
   public ADD (int L) {
      super(2*L, L, L);
   ......
   ......
```

```
#include<iostream>
class ADD : public Circuit {
   private:
         int L;
   public:
         ADD (int L)
:Circuit(L, L+1){
   ......
```

```
class ADD:
   def __init__(self,x,y):
         self.x = x
         self.y = y
   description = "This  is
an ADD“
......
......
```

Figure I.1: An architectural diagram of the system. DSML is the first step to specify GC model design rules and configure the development environment. GC models can be designed in this environment and models can be put into an database as a library. GC models are transformed into working codes with given interpreters.

I.2 Goal

In this thesis, we design a generic approach to represent GCs in a domain specific modeling language (DSML) [6] way. In doing so, the GC model will be transferable to automatically generated code that is amenable to the corresponding secure computation frameworks. The procedure of the system is demonstrated in Figure I.1. The generic specification of GCs, namely DSML, automatically generates a development environment, which could be graphical or textual. Such an environment can enable developers to design their own custom-defined GC models. Meanwhile, we can also provide a circuit library for reference and reuse during the development. During the transformation process, interpreters accept data from GC models and automatically generates corresponding working codes.

Given the feasibility and convenience of implementation, we use the Generic Modeling Environment (GME) [7] as the tool for circuit DSML-based circuit design. We build a circuit library which covers circuits with a wide range of computational functionality and show that: (1) our generic approach is capable of representing most commonly used circuits in secure computation and (2) users may take advantage of the library to design their own custom-defined circuit.

We also choose a GC implementation, FastGC, and implement its interpreter to demonstrate the capability of our circuit DSML in automatically generating codes. We test the generated codes by comparing and replacing with their target codes to verify the correctness of the interpreter.

CHAPTER II

PRELIMINARIES

This chapter specifically discusses significant concepts used in our approaches. We first present a quick overview of GCs, which is followed by a review of its recent implementations. Then we introduce the notion of a domain specific language (DSML), a special-purpose language which is applied to GCs specifications here.

II.1 Garbled Circuits

GCs were introduced as a solution to the two-party secure computation problem [5]. These were based on the assumption that function would be securely computed under a semi-honest model [1] [9]. Yao's construction presents a constant-round [2] protocol which is independent of the number of inputs or the size of the circuit for secure computation of any function in the semi-honest model [10]. This technique is based on modeling the functionality as a logic circuit composed of binary gates and encrypting the result tables.

Figure II.1 shows an example of a logic circuit that computes the hamming distance of two l-bit binary integers and outputs a result of *k*-bit binary integer. The hamming distance between two binary integers is the number of positions at which the corresponding digit from two integers are different. In this circuit design, there are *L* XOR gates in a parallel fashion. Each XOR gate receives a digit from both of the integers participating in computation at the same corresponding position. The XOR gate can compare whether the

[1] In a semi-honest model, each participant is assumed to execute the protocol properly with the exception that it keeps a record of all its intermediate computations and derives other parties private inputs from the record [8].

[2] A constant-round protocol means that the secure functions computation can be completed in a constant number of rounds.

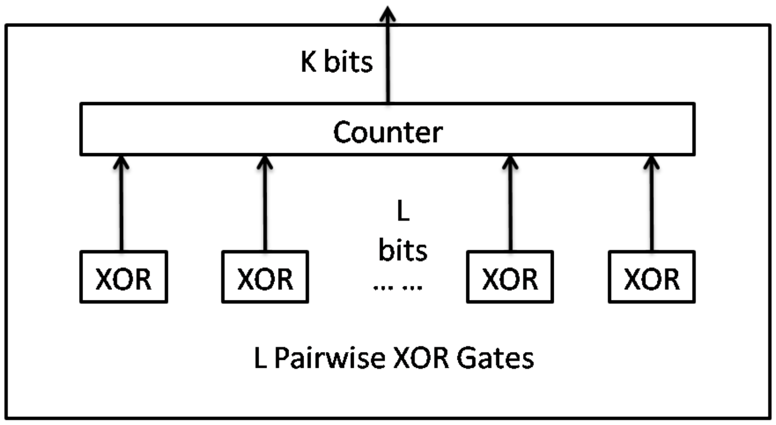


Figure II.1: An illustration of the logic circuit for the hamming distance which is composed by a number of XOR gates and a Counter. XOR gates compare each bit pair from two integers which are at the same position. The Counter counts the number of positions where the corresponding bit from the two integers are different.

two digits are the same and output 1 if they are different. The Counter counts the number of 1s' from the XOR output and return the final result.

In this technique, the computation function must be converted into a logic circuit. Our work in this thesis focuses on representing the logic circuit in a generic way, so that circuit codes working in secure frameworks can be automatically generated with the corresponding interpreters, which facilitates the conversion process.

II.2 Garbled Circuit Implementations

GCs show great practical significance with recent implementation of generic secure computation frameworks that build on this technique. Examples include (1) Fairplay [11] which is a secure two-party computation system that applies GCs for constructing secure computation protocols, (2) FastGC [12] which shows that generic protocols for secure computation can scale to handle large circuits or input sizes and be competitive with special-purpose protocols, and (3) ABY [13] which allows pre-computation of almost all cryptographic operations. ABY works like a virtual machine that abstracts the underlying secure computation protocols and provides efficient conversions between secure computation schemes.

Other than the above implementations aimed at implementing a platform for secure computation, there are some projects that work on the facilitation of functionality implementation so that GCs can be easily transformed into code that executes on these secure platforms. Rastogi and colleagues [14] introduced Wysteria, a high-level programming language for writing secure multiparty computations. ObliVM program [15] provides an user-friendly programming framework, a source language in which programs written will be compiled into concise circuits. In this thesis, we present our approaches in facilitating the implementation by domain model design and interpretation.

II.3 Domain Specific Modeling Language

Domain specific modeling languages (DSMLs) are a methodology for designing and developing systems in software engineering. A DSML refers to a modeling language that offers notations and concepts that can be directly manipulated by a domain expert to transform a solution into a model [6, 16]. The abstract syntax of a DSML is usually defined by a meta-model which describes the concepts of the language, the relationships between them, and the structuring rules that constraint the model elements and their combinations in order to respect the domain rules [17].

A DSML is a special-purpose language designed to solve a particular range of problems [18]. Widely-known examples include HTML which represents the layout of Web pages and SQL which queries and updates databases. It simplifies a large problem so that a programmer can identify a language that efficiently expresses that class of problems. In this way, we connect the vocabulary in the language with the final environment in which the problem is solved.

DSMLs make people focus just on the unique functionality, the differences between the various features, free from wasting time and effort in re-implementing similar functionality. Typically, a DSML has the following advantages [19]: (1) DSML is better suited for requirements engineering because users with the domain expertise can better understand the models, (2) They have restricted semantic scope, such that we only define what we need and thus control the complexity, and (3) They increase domain specific reuse of components, which improves the quality of systems.

II.3.1 Advantages of DSML On GC Specification

We prefer a DSML to an electronic design automation (EDA) [3] tools [20] which are usually used in integrated circuit design in industry because (1) these tools typically focus on industries generating integrated and printed circuit boards, not on logic circuit design, and (2) the number of components and connections in a GC is not fixed. For instance, in Figure II.1 as instance, there are a number of L XOR gates whereas L is a variable, which means that EDA tools can not specify a GC by listing each and every subcircuit and wire. We specify the circuit for automatically generating codes while EDA tools are used to design and analyze circuits.

Furthermore, we apply DSMLs for GC specification under the following considerations:

1. They provide the flexibility to specify the components and connections in the circuit even though their number is variable. They are capable and comprehensive for representing the GCs commonly invoked in secure computation.
2. They control the scalability of garble circuit specification. We can keep our language and tool compact, minimizing unnecessary modules irrelevant to GCs.

[3]Electronic design automation is a category of software tools for designing electronic systems.

3. They provide better extendability for further requirements in the future. Our generic circuit language is oriented to a multiple of secure computation frameworks, each of which may have own unique requirements.

II.4 Summary

This chapter provided a technical introduction to GCs, its current implementations, and applications. We illustrated the concept of a GC with an example and provided justification for designing domain specific modeling languages for their representation.

CHAPTER III

DSML DESIGN APPROACHES

In this chapter, we introduce our approaches for designing a GC modeling language. We begin with the design including the goals, assumptions and patterns. Next, we elaborate a GC vocabulary and how we specify the complicated connections in a GC. This chapter concludes with a discussion of transformations from GCs models to working code and how such code can be automatically generated.

III.1 Design Goal

Building on the motivations in Chapter II, the process of GC design can be simplified if we design a generic approach to represent the circuit for code generation. The design goals are listed as follows.

1. A generic way to represent the logic circuit including its attributes (e.g., circuit name, degrees of input/output, and number of components), sub-circuit, and the wires in the circuit.
2. An extensible and comprehensive vocabulary to cover all depicted aspects of a circuit.
3. A framework-dependent interpreter that is compatible with a GC meta-model.

III.2 Design Assumptions

Before designing a GC meta-model, we must introduce the basic definition of a GC, and its features and what a GC model is designed for.

First, a garbled circuit is a type of binary logic circuit that is composed by AND, OR, XOR gates and wires only. A GC model can have other GCs as sub-circuits, which makes the defined circuit reuseable and enables the composition of complex circuits.

Second, a GC language should allow for the specification of the initialization of components, wires connecting each components, and the definition of input and output.

Third, a GC is modeled and transformed to meet certain computation functionality. It is a special type of logic circuits in which connections between components can be established and expressed in certain rules in most cases. In other words, connecting each pair of ports one by one is not a requirement for GC representation. Most garbled circuit can be concise and organized if well-designed after determining what are the sub-circuits and wires.

Finally, a GC meta-model is designed for the automatic generation of codes in secure frameworks. Thus, we always keep in mind that the design of a meta-model should be strongly associated with code generation, so that transformation from models to code is readily implementable.

III.3 Design Pattern

In this thesis, we apply the software engineering principle of a Composite Pattern in the GC meta-model design. ”A pattern is said to be a composite pattern, if it can be best explained as the composition of further atomic or composite patterns” [21]. More specifically, the composite pattern is a partitioning design pattern which treats a group of objects in the same way as a single instance of an object.

The selection of Composite pattern is based on the aforementioned design assumptions. A GC model can be composed by other GCs, which is the essence of a Composite pattern.

Moreover, a Composite Pattern provides a clear view of the whole-part relationship [1] at the GC level.

III.4 Metamodel Vocabulary[2]

This section describes an elaboration of the vocabulary in the GC domain language. Each terminology in the vocabulary represents either an entity in GC or takes the task of code generations.

- **Logic Gates**: *AND*, *OR*, and *XOR* gates. These are all for logic and binary specification.
- **Input**: This defines the input ports of the custom-defined circuit.
- **Output**: This defines the output ports of the custom-defined circuit
- **GC**: This *CompositeCircuit* represents the custom-defined circuit under design.
- **Circuit Reference**: This *CircuitRef* is the reference of *CompositeCircuit*. A reference in meta-model design is similar to a pointer in a programming language. It is introduced under the considerations that: 1) with a reference, the GC model that serves as a subcircuit is responsible only for declaring that it is used in the whole circuit, independent of initialization, connection and specification of fixed value, and 2) it ensures the Custom-defined GC model has a low degree of coupling because each functionality is undertaken by a corresponding type of *CircuitRef*, which is also beneficial for automatic code generation. According to the functionality, *CircuitRef* has the following subclasses:

[1]A whole-part relationship represents the composition relationship that one object is composed by one or more components

[2]The module names in DSML are italicized.

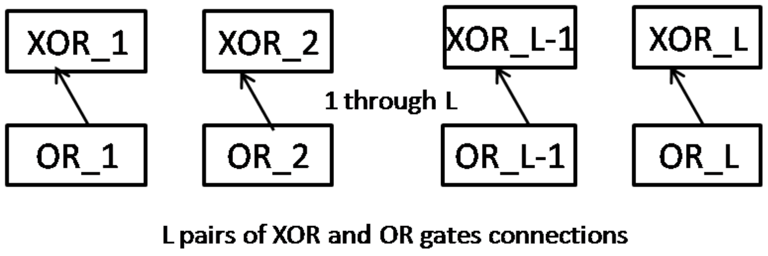
  - **Initialization**: The *InitialRef* module is designed for the initialization and building of a subcircuit.
  - **Wires with fixed value**: The *FixedWireRef* module is designed for specifying a fixed value for certain wires.
  - **Circuits in connection**: The *ConnectionRef* module is designed for representing the connections between subcircuits and defining the input and output.

- **Wire**: The *Wire* represents a connection in the circuit. It defines how two components connect with each other. It is an abstract class and has the *InternalWire* and the *DefineInOutPut*. *InternalWire* specifies the connections between subcircuits. *DefineInOutPut* is responsible for defining the input and output ports of the custom-defined circuit.

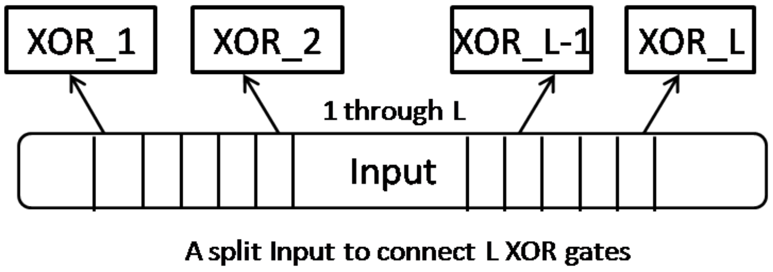
III.5 Connections in GC

The specification of connections in a GC is the key in defining a circuit language. In our circuit language, subcircuits that participate in the connections are represented by the *ConnectionRef*. We also have *Input* and *Output* for representing input and output ports. Building on the aforementioned design assumptions for a GC that certain rules may apply to make GC more concise and easy to express, we mainly focus on the following two connection scenarios:

- A connection pattern is applied in a group of circuit pairs. Technically, circuit pairs with the same connection pattern can be viewed as an aggregate. Figure III.1a shows the example of how L pair of the XOR and OR gates connect. Each output of OR gate connect to the first port of the corresponding XOR input.
- A circuit with a larger number of ports is connected to a group of circuits with a smaller number of ports. In this case, we need to split the larger circuit so that each



(a) Scenarios 1



(b) Scenarios 2

Figure III.1: Scenario 1: The same connection pattern applied to multiple circuit pairs. Scenario 2: A splitting of large circuit to connect it with a group of small circuits.

piece can connect to each smaller circuit. Figure III.1b shows an example of how we connect XOR gates with the counter in the Hamming distance circuit. The counter is split into L pieces so that each port connects with the output of XOR gate.

Thus, we have two sub-classes of *ConnectionRef* for each scenario.

- **ReplicateRef** is derived from *ConnectionRef* and it represents a group of subcircuits that participate in the same connection pattern.
- **SingleRef** is derived from *ConnectionRef* and it represents a single subcircuit. Unlike *ReplicateRef*, its ports can be split and the component can be separated into smaller pieces so that the *SingleRef* may connect to a group of circuits.

III.6 Transformations from Models to Codes

As stated earlier, the circuit language is designed for automatically generating codes. Our meta-model is strongly associated with code generations. Typically, circuit code has the following parts.

III.6.1 Declaration of GC

We use the term declaration to mean the basic information of a GC including the name of the circuit class, the arguments that may use, and the constructor function. The *CompositeCircuit* module itself, is responsible for this declaration part. The *CompositeCircuit* is composed of the following parts.

- **Variable**: The arguments that can be defined in the GC class.
- **InDegree**: The number of input ports in a GC.
- **OutDegree**: The number of output ports in a GC.
- **ComponentNumber**: The number of subcircuit or logic gates in a GC.
- **CircuitClassName**: The name of a GC.

III.6.2 Initialization of Sub-circuit

The initialization of a sub-circuit is achieved through the *InitialRef*, which is based on the following attributes.

- **CircuitIndex**: specifies the identification of a sub-circuit in a GC by which we can have access to it.

- **CircuitArgument**: the arguments that may be required in the initialization of sub-circuits.

III.6.3 Wires

The *Wire* in a GC meta-model represents the connections in a GC and is invoked to generate codes that define the input/output, as well as the connections between sub-circuits.

- **Define input/output** is represented by the *DefineInOutPut* which connect Input/Output with the *ConnectionRef*.
- **Wire connecting subcircuits** is represented by the *InternalWire* that connects the *ConnectionRef* with the *ConnectionRef*.

The *Wire* is specified by two attributes: 1) SourcePortIndex, the designation of ports on the source component, and 2) DestinationPortIndex, the designation of connected ports on the destination component. By associating ports from the source with the destination, we connect the circuit pair.

III.6.4 Fixed Value For Wire

FixedWireRef specifies fixed value for certain wires with the following attributes:

- **CircuitIndex** specifies the index of designated circuits.
- **PortIndex** specifies the index of wires on the circuit.
- **FixedValue**: The designated fixed value.

III.7 Summary

This chapter provided a GC specification approach under consideration for design goals and assumptions. The meta-model vocabulary covers a comprehensive set of concepts for GCs. Each scenario for GC connections are handled dealt with, which ensures a GC meta-model's capability in specifications. Each module in the meta-model has a clear path working code to model interpretations.

CHAPTER IV

DSML IMPLEMENTATION ON GME PLATFORM

In this chapter, we discuss the implementation of a DSML approach discussed in the Generic Modeling Environment (GME) platform. This chapter begins with an introduction of GME and its advantages. Next, implementation details of DSML are demonstrated along with a step-by-step guide to building a custom-define GC model. Finally, a library of GC models that covers an array of computational functionality is also provided to build new GC models.

IV.1 GME: DSML Development Environment

As mentioned earlier, GME serves as the development environment for the GC DSML. GME is a configurable tool-set that supports the easy creation of domain-specific modeling. The configuration is accomplished through meta-models, which specify the modeling paradigm (modeling language) of the application domain [7]. It is a tool-set that combines DSML specification, model design, and model interpretation. Specifically, the generic modeling environment itself is used to build the meta-models which specify the modeling language. The modeling language automatically generates the target domain-specific environment for building domain models, which are subsequently stored in a model database. The models work with an interpreter to automatically generate the applications and harmonize the input with different analysis tools. Figure IV.1 depicts how the development environment.

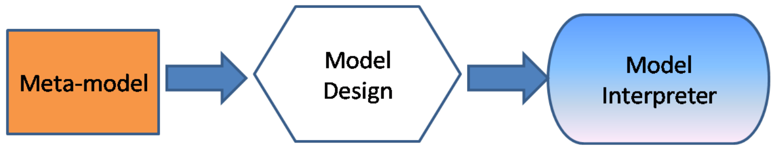


Figure IV.1: The process for building the development environment.

IV.2 Advantages of GME

We apply GME in this project as the DSML design tool for several reasons:

1. GME provides comprehensive support for various concepts for building large-scale and complex models. It enables the design of a hierarchy, multiple aspects, sets, references, and explicit constraints [7]. This ensures a GC can be represented in a DSML design.

2. GME is a graphical modeling environment which is relatively convenient for visual design, unlike other command-based interfaces[1]. Furthermore, it is an integrated environment which includes meta-model design, models building, and model interpretation. This means that work communicated in this thesis can be achieved through one integrated platform.

3. GME is extensible, such that writing an interpreter is relatively simple. This is because a more complex interface is layered on top of the COM interfaces, which ensures an easy-to-use extensible C++ API [22]. This API enables direct access to every module and attribute in the model.

4. GME has a built-in constraint manager which enforces all domain constraints during model building [7]. In this GC case, we can ensure the correctness in building the models and prevent improper wrong wires by setting constraints.

5. GME supports meta-model composition, which is a capability for reusing and com-

[1] A command-based interface is a user interface that directly interact with users by typing commands.

bining existing modeling languages and language concepts [23]. Models in the library thus can be reused in building mew models.

IV.3 GC Metamodel on GME

The implementation of GC metamodel on the GME platform is shown in Figure IV.2. This is an implementation of the circuit language discussed in Chapter II. From the design pattern view, the GarbledCircuit is composed by the LogicalGate and the GarbledCircuit itself, which means that a GC model is composed of logic gates and may have other GC models as sub-circuits.

IV.4 Build Your Own GC Domain Models[2]

The domain-specific environment automatically generated by a GC meta-model provides users with a comprehensive and flexible user interface to design a custom-define GC model with only five steps. We continue with the Hamming distance circuit whose corresponding code in FastGC is in class HAMMING_2L_K, as example. We demonstrate the steps to build a GC model to show the differences of implementing GCs as code between by building the model and by writing the code.

The circuit class HAMMING_2L_K computes the number of positions over digit from two *L*-bit integers are different and output a *K*-bit integer. It is composed by *L* number of XOR gates and a counter with *2L* input ports and *K* output ports.

First, we declare the basic information for HAMMING_2L_K as described above. We drag a *GarbledCircuit* module from the user interface and set its attributes. Figure IV.3a shows the attributes set for the GC model HAMMING_2L_K. It accepts a variable *L* which is the number of XOR gates and *K* which is the output length. As for degrees, it has *2L*

[2]In order to differentiate, GC model names will be small caps.

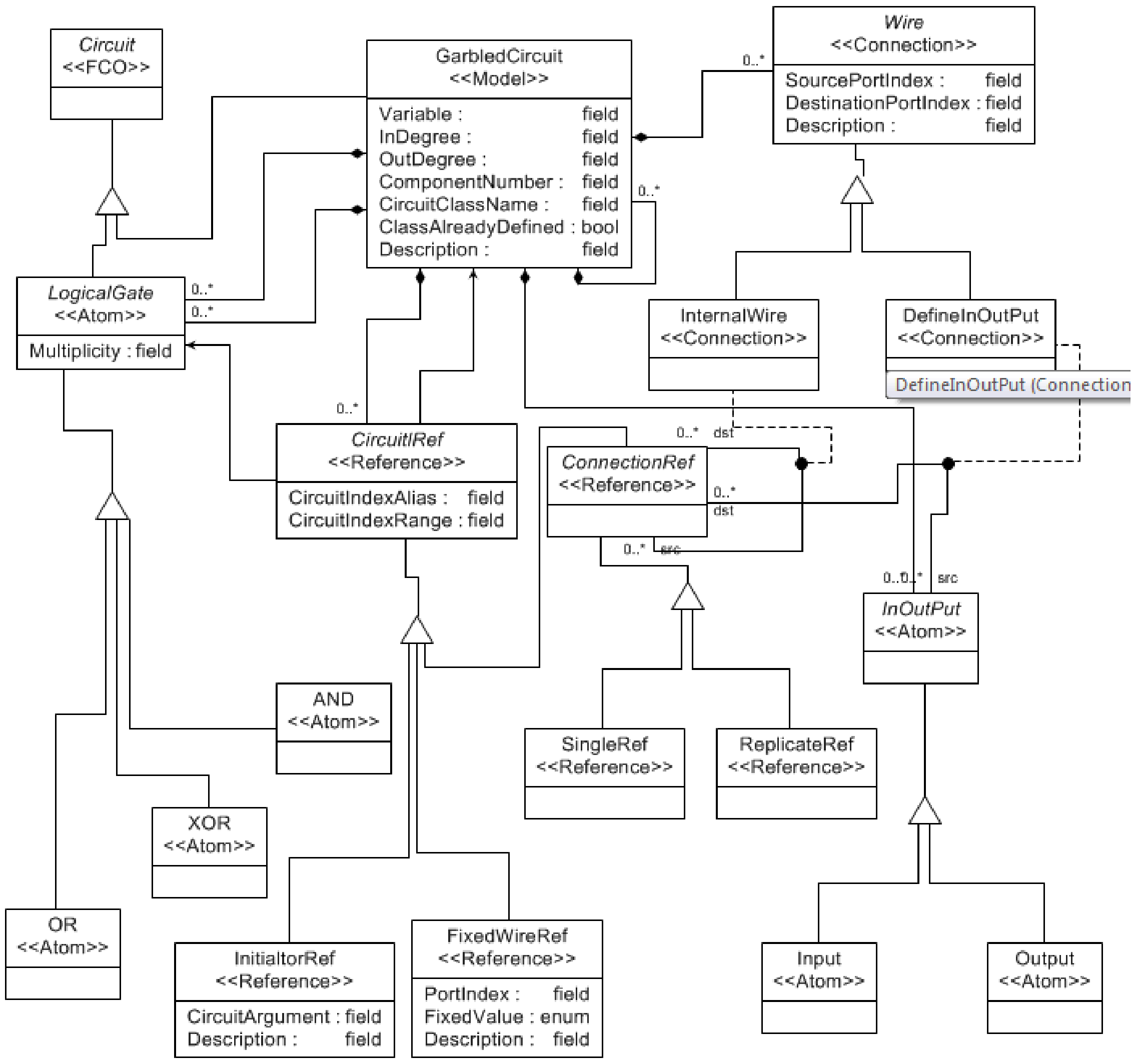
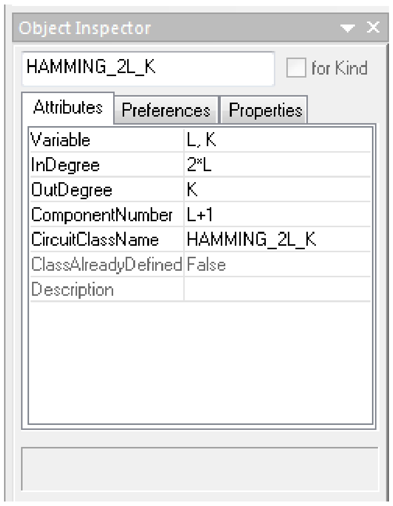


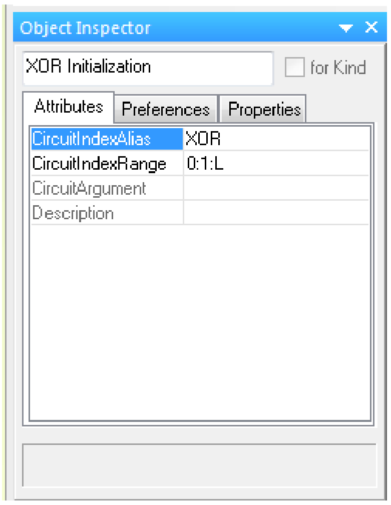
Figure IV.2: A class diagram of a GC model that presents each module in the model. Circuit is the first class object and is the parent class of LogicalGate and GarbledCircuit. LogicalGate defines the three gates in a logic circuit, and has three subclasses AND, OR, and XOR. GarbledCircuit represents the GC model. It contains: 1) the sub-circuits used, which are represented by LogicalGate and GarbledCircuit, 2) CircuitRef and its subclass, which includes InitialtorRef, FixedWireRef and ConnectionRef, which perform the tasks of sub-circuit initialization, setting fixed values for wires and connections in GC model, respectively, and 3) Wire and its subclass, DefineInOutput, which defines the input and output, and InternalWire which specifies the connections between sub-circuits.



HAMMING_2L_K

Variable	L, K
InDegree	2*L
OutDegree	K
ComponentNumber	L+1
CircuitClassName	HAMMING_2L_K
ClassAlreadyDefined	False
Description	

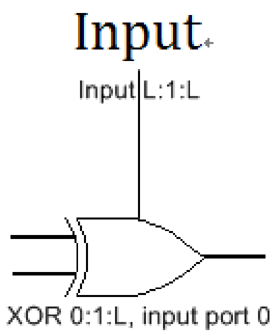
(a) GC Declaration



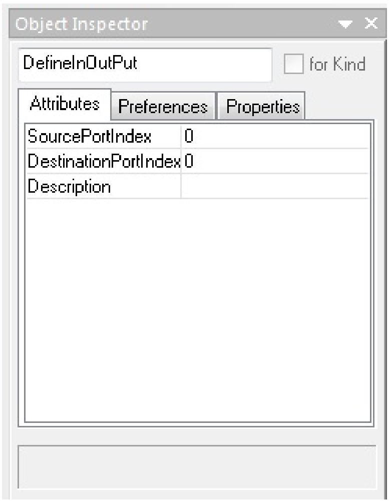
XOR Initialization

CircuitIndexAlias	XOR
CircuitIndexRange	0:1:L
CircuitArgument	
Description	

(b) XOR Initialization



(c) Define Input



DefineInOutPut

SourcePortIndex	0
DestinationPortIndex	0
Description	

(d) Specify Connection

Figure IV.3: (a) GC model HAMMING_2L_K accepts arguments L and K and has 2L input ports and K output ports. In the model, there are L XOR gates and one counter. (b) initialization of L paralleling XOR gates which are aligned from index 0 to index L-1. (c)how input is defined by the connection Input module with *ReplicateRef* of the XOR gates. The last part of L ports on Input module is split into L pieces to connect with L XOR gates. (d) attribute set for connecting Input and XOR gates.

input and K output ports, respectively.

Second, we declare and initialize the subcircuits used in a GC model by dragging its instance into a GC model. Initialization is accomplished by setting its *InitaltorRef*. For designation of multiple indexes, we use a vector format of start:step:number. For instance, 0:1:L means a HAMMING_2L_K of L indices starting at 0 and increasing by 1. Figure IV.3b shows how to set up L XOR gates which is aligned from index 0 to L-1.

Third, we make connections between the sub-circuits, input and output. Figure IV.3c shows the connection between the last L ports of Input with each first port of L concurrent XOR gates. Figure IV.3d shows the attributes set for *DefineInOutput* to specify such an connection.

Finally, we specify the fixed value for a specific port (wire). In this Hamming distance case, there are no wires with fixed value. However, the counter used in the Hamming distance computation is a good example of why a fixed value can be useful. Specifically, suppose we have L XOR gates, but the counter only accepts an N, where N is larger than L, bit input. In this case, we set value of the remaining (N-L) ports as 0.

IV.5 Circuit Library

We built a GC domain model database which contains GC models with a variety of computation functionality, such as addition, comparison. We built this database to test the capability of the GC meta-models in implementing the most commonly-used GCs. It also serves as a library which enables users to design their own custom-defined GCs directly based on models in the library. The circuit library is built in a bottom-up hierarchical structure. Technically, the GC is composed of three basic gates: AND, OR, and XOR. Complex circuits are usually composed by relatively simple circuits. In this bottom-up hierarchy, circuits in the higher level are composed of circuits in the lower levels. The details about the GC library can be found in Appendix A.

IV.6 Summary

In conclusion, GME offers a comprehensive ability to implement GC DSML. It also enables the integration of model design and interpretation, which supports our work from beginning to end. Our GC meta-model is capable of supporting common-used GC specification with flexibility and extensibility.

CHAPTER V

FASTGC INTERPRETER

This chapter presents our GC model interpreter implementation. We first introduce FastGC, a recent implementation of GC protocols. We provide justification for choosing this framework and what it is necessary to generate its working codes. In order to do this, we take a deeper look at the structures of FastGC codes and how each interface is implemented by showing code templates. Then we introduce the architectural design of our interpreter which can be classified into: 1) a GC model receiver which accepts data from models, 2) FastGC code generation which outputs code in the form of code templates and parameters, and 3) model interpretation which adapts model data for code generations.

V.1 GC Implementation: FastGC

FastGC [12] is short for Fast Secure Computation Using Garbled Circuits which is a secure two-party computation model that enables two parties to evaluate a function cooperatively without revealing to either party anything beyond the output. It has many feature that make this a popular tool for GC implementation. First, it is relatively new in comparison to other implementations. Second, it is efficient in speed and memory. Third, GC code in FastGC has a clear structure and interfaces to be implemented. Fourth, FastGC provides GC code and samples which cover a wide range of computational functionality that enables design and testing.

The GC code in FastGC is organized under the Composite design pattern, which is the same design pattern applied in our meta-models. The abstract class <u>Circuit</u>[1] serves as

[1]The class names in FastGC will be underlined.

the Component[2] in the pattern. It is the abstraction for all the GC classes which declare the wires and subcircuits and define the interfaces. Both wires and subcircuits in FastGC are stored in a single array, which means each subcircuit or wire is accessed by its index in the array, which works as the identification. The Circuit class has two subclasses, CompositeCircuit class and SimpleCircuit_2_1 class. SimpleCircuit_2_1 is the parent class that represents all types of basic gates, namely, AND, OR, and XOR gates. It plays the Leaf [3] role in the pattern. CompositeCircuit is the parent class for all the GCs that represent certain computation functionality. It is the Composite [4] part in the design pattern. It is the superclass from which the GC code is derived. CompositeCircuit defines the interfaces the GC code needs to implement.

V.2 GC Code In FastGC

The generation of FastGC code is achieved by the implementation of interfaces in class CompositeCircuit with the data from a GC model. FastGC code generation also involves a constructor function and a class layout. Each interface is implemented with a fixed format to mitigate the complexity of interpreter implementation.

V.2.1 Constructor Function

Constructor. The constructor function specifies the degree of input and output of the circuit. It also specifies the number of subcircuits within the circuit. This information is supplied by GC model attributes. The following code sample in Figure V.1 shows the constructor function of a Hamming distance circuit. The initialization of class member is by the super constructor.

[2]In a Composite pattern, a Component corresponds to the abstraction for all components, including composite objects.

[3]In a Composite pattern, a Leaf represents leaf objects in the composition.

[4]In a Composite pattern, a Composite represents a composite Component. It can have subclasses which have different implementation of interfaces from the parent class.

```
//HAMMING_2L_K Constructor
public HAMMING_2L_K_Weijie(int L, int K) {
   //indegree, outdegree, component number, circuit name
  super(2*L, K, L+1, "HAMMING_2L_K_2*K");
  //class variable list initialization
  this.L = L;
  this. K = K;
}
```

Figure V.1: The constructor function of Hamming distance circuit.

```
public void build() throws Exception {
   createInputWires();
   createSubCircuits();
   connectWires();
   defineOutputWires();
   fixInternalWires();
}
```

Figure V.2: The implementation of the build function.

V.2.2 Interfaces To Be Implemented

Build is the function that defines almost everything needed for a custom-defined circuit, including its subcircuits, its connections inside, how input and output are specified and if there are wires with a fixed value. The Build functions is implemented as in Figure V.2.

The first function, **createInputWires**, declares an array of wires which has a length of the circuit in-degree. It does not initialize the input ports. It is implemented in the parent class, namely Circuit. However, other functions in build() should be implemented in GC code:

- **createSubCircuits**. This function specifies the initialization of subcircuits in GC, including the circuit name and its parameters and how they are aligned in the vector, namely, their indexes. The codes in Figure V.3 corresponds to the template of sub-circuit initialization. Notice that it only needs parameters of circuitId, circuitName, and argument list to generate an initialization code [5].

```
//example: put a counter at location L
subCircuits[L] = new COUNTER_L_K(L, K);
//code template
subCircuits[circuitId] = new circuitName(arg 1,..., arg n);
```

Figure V.3: The initialization of the subcircuits in the FastGC code.

This code template covers the initialization cases for any kinds of sub-circuits except AND gate and OR gate. The initialization of these two is a little bit different from the template showed above, as shown in the following code in Figure V.4.

```
//initialization of AND gate
subCircuits[circuitId] = AND_2_1.newInstance();
//initialization of OR gate
subCircuits[circuitId] = OR_2_1.newInstance();
```

Figure V.4: The initialization of the And gate and the Or gate in the FastGC code.

- **connectWires**. This function defines the connections in the custom-defined circuit which can be categorized into two types: 1)how input ports are defined, and 2) how sub-circuits are connected with each other. The following code depicts the parameters necessary to generate such code in Figure V.5 and in Figure V.6.

[5] Parameters in code templates are italicized with red font.

- **defineOutputWires**. This functions defines the output ports. It selects output ports from certain sub-circuits as the output ports for the whole custom-defined circuit. As shown below in Figure V.7, its code template is similar to the one how we define input.

```
//example: define the input
for( int i=0; i<L; i=i+1)
  inputWires[i].connectTo(subCircuits[i].inputWires, 1);
//code template
inputWires[inputId].connectTo(
   subCircuits[circuitId].inputWires, portId);
```

Figure V.5: The definition of the input in FastGC code.

```
//example: connect the sub-circuits
for( int i=0; i<L; i=i+1)
  subCircuits[i].outputWires[0].
     connectTo(subCircuits[L].inputWires, i);
//code template
subCircuits[srcCircuitId].outputWires[srcPortId].
   connectTo(subCircuits[dstCircuitId].inputWires,
      dstPortId);
```

Figure V.6: The definition of the connections between the subcircuits in FastGC code.

- **fixInternalWires**. In a custom-defined circuit, to achieve a computation goal, we may simply want some ports or wires not to be involved in connections. The values of such wires are thus not influenced by connections and could have a fixed value. The code template is shown in Figure V.8.

```
//example: define output
for( int i=0; i<K; i=i+1)
   outputWires[i] = subCircuits[L].outputWires[i];
//code template
outputWires[outputId] =
    subCircuits[circuitId].outputWires[portId];
```

Figure V.7: The definition of the output in FastGC code.

```
//set the first XOR first port as 0
subCircuits[0].inputWires[0].fixWire(0);
//code template
subCircuits[circuitId].inputWires[portId].
    fixWire(fixedValue);
```

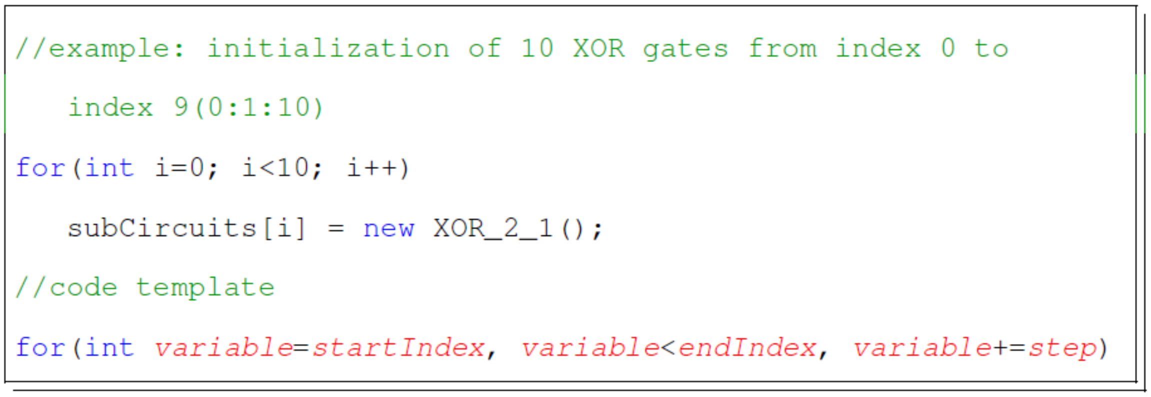
Figure V.8: The specification of fixed value in FastGC code.

V.2.3 For Loop

As discussed in the meta-model design earlier, a range of circuits may have the same circuit type and arguments in the initialization part, or may apply the same connection rule. In FastGC code, we use a for loop to specify such information for the GC. The following code in Figure V.9 illustrates how the loop is specified.

V.3 Interpreter Interface

The interpreter is composed of two parts. First, the CodeWriter corresponds to the classes of code templates discussed. Second, the CodeGenerate accepts data from a GC model and transforms the models into CodeWriter objects.



```
//example: initialization of 10 XOR gates from index 0 to
   index 9(0:1:10)
for(int i=0; i<10; i++)
   subCircuits[i] = new XOR_2_1();
//code template
for(int variable=startIndex, variable<endIndex, variable+=step)
```

Figure V.9: The for loop code template.

V.3.1 CodeWriter

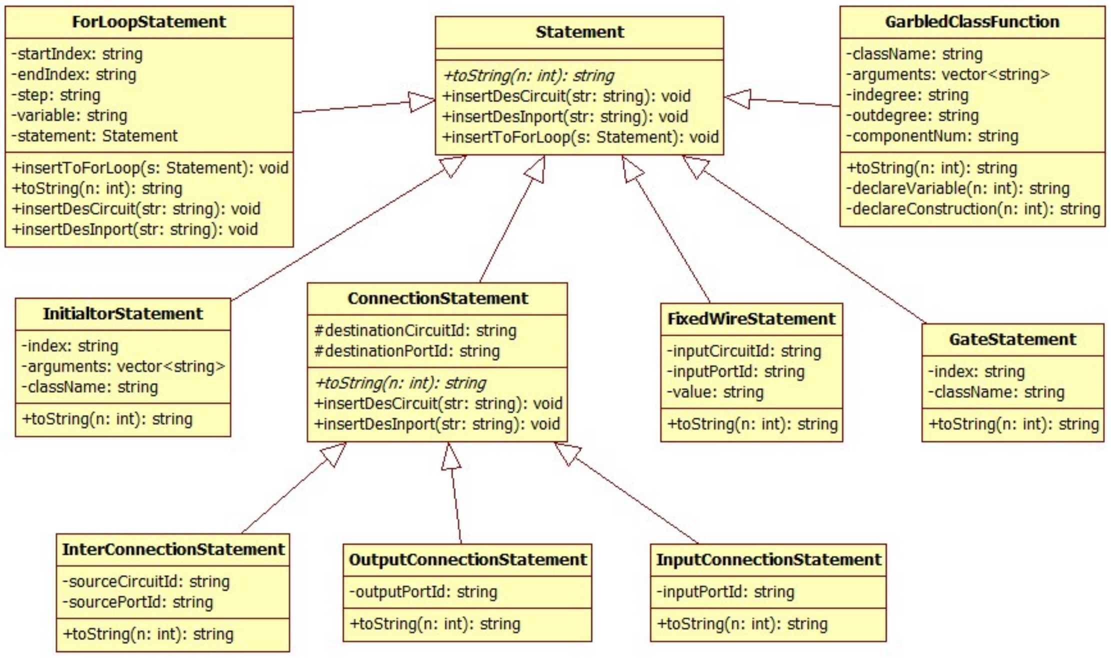
CodeWriter is an namespace in which each type of class generates its corresponding Java code sentences based on parameters and code templates. We design classes for each and every template that discussed above. Figure V.10 shows the UML of these classes and their hierarchy. Each type of Statement class is derived from the pure class Statement which has a abstract function toString which outputs the code that the class object represents.

For each subclass derived from class Statement, it can be implemented by: (1) a parameter list which is necessary to output the code, (2) a constructor function which accepts parameters during the model transformation, and (3) an implementation of the function toString, which is based on its respective code template.

Appendix B shows a concrete example of how we declare the class InitialtorStatement and how we implement the function toString. Other Statement subclasses can be implemented in the same way.

V.3.2 CodeGenerate

In Figure V.11, this is CodeGenerator namespace designed according to the meta-model so that it can accept the data from a GC model directly. Figure V.11 shows the UML graph



ForLoopStatement
-startIndex: string
-endIndex: string
-step: string
-variable: string
-statement: Statement
+insertToForLoop(s: Statement): void
+toString(n: int): string
+insertDesCircuit(str: string): void
+insertDesInport(str: string): void
Statement
+toString(n: int): string
+insertDesCircuit(str: string): void
+insertDesInport(str: string): void
+insertToForLoop(s: Statement): void
GarbledClassFunction
-className: string
-arguments: vector<string>
-indegree: string
-outdegree: string
-componentNum: string
+toString(n: int): string
-declareVariable(n: int): string
-declareConstruction(n: int): string
InitialtorStatement
-index: string
-arguments: vector<string>
-className: string
+toString(n: int): string
ConnectionStatement
#destinationCircuitId: string
#destinationPortId: string
+toString(n: int): string
+insertDesCircuit(str: string): void
+insertDesInport(str: string): void
FixedWireStatement
-inputCircuitId: string
-inputPortId: string
-value: string
+toString(n: int): string
GateStatement
-index: string
-className: string
+toString(n: int): string
InterConnectionStatement
-sourceCircuitId: string
-sourcePortId: string
+toString(n: int): string
OutputConnectionStatement
-outputPortId: string
+toString(n: int): string
InputConnectionStatement
-inputPortId: string
+toString(n: int): string

Figure V.10: CodeWriter: a namespace for classes of code templates. In this class diagram, the class Statement is the abstract class for all the code template classes that have an abstract function toString. The toString function generates the codes stored by each class object. Each code template has subclasses including i) InitialtorStatement, ii) ConnectionStatement, iii) FixedWireStatement, and iv) ForLoopStatement. The GateStatement is specifically designed for AND and OR gate initialization. The GarbledClassFunction stores the basic information for model and implements the constructor function.

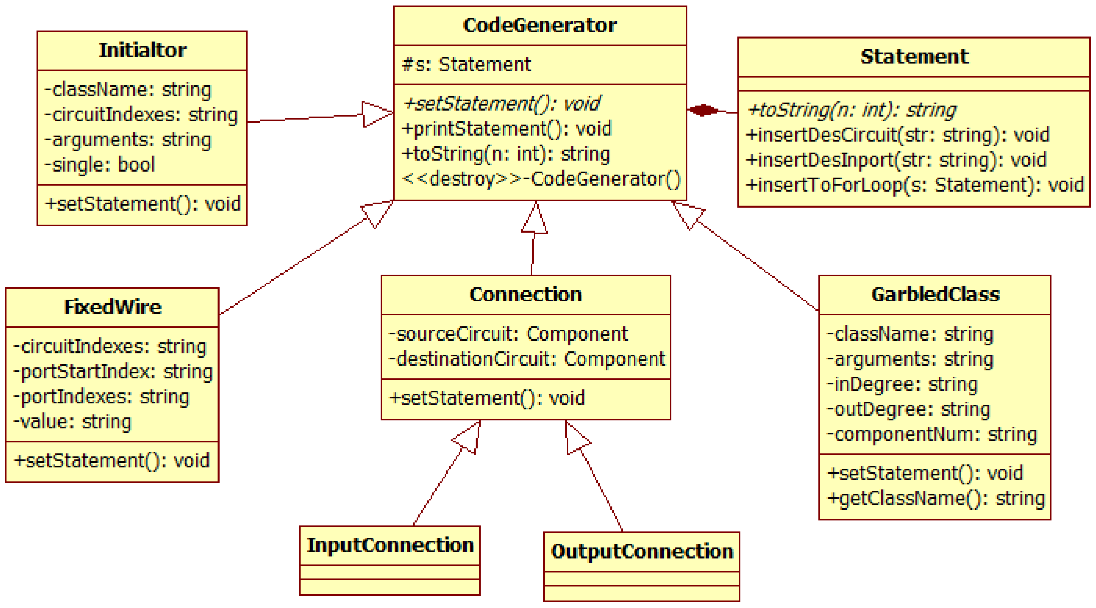


Figure V.11: CodeGenerator: an namespace for storing the data from GC models and transforming models into CodeWriter classes. In this class diagram, CodeGenerator is the abstract class which defines the interfaces and class member for subclasses. It has a Statement member which stores the transformation result. setStatement function executes the transformation an printStatement outputs the generated codes. CodeGenerator has subclasses including i) Initialtor, ii) FixedWire, iii) Connection, and iv) GarbledClass, each of which corresponds to a module in GC meta-model.

of CodeGenerator classes. The CodeGenerator has attributes, structure and pattern that are similar to a GC meta-model. Each CodeGenrate sub-class is associated with a module in the meta-model which directly generate codes.

The CodeGenerator class has a Statement object as its member which stores the GC codes its module represents. The function setStatement supports the transformation from a model to a statement.

In order to generate the whole file of a GC code, we engineered the class GCJava to contain members of all possible CodeGenerator variations. Each CodeGenerator subclass member is generated by reading data from a corresponding module. These members generates codes to implement corresponding interfaces. The GC code file is then generated by sorting these functions and codes in the conventional order in FastGC.

Besides these two namespaces, we also have two subsidiary classes, Component and Range. The Component class represents the component connected by wires. The Single-Component is the component that can be split into smaller pieces, corresponding to *SingleRef*, The Input and Output in the meta-model. The MultipleComponent can designate a range of components with the same connection pattern, corresponding to *ReplicateRef*.

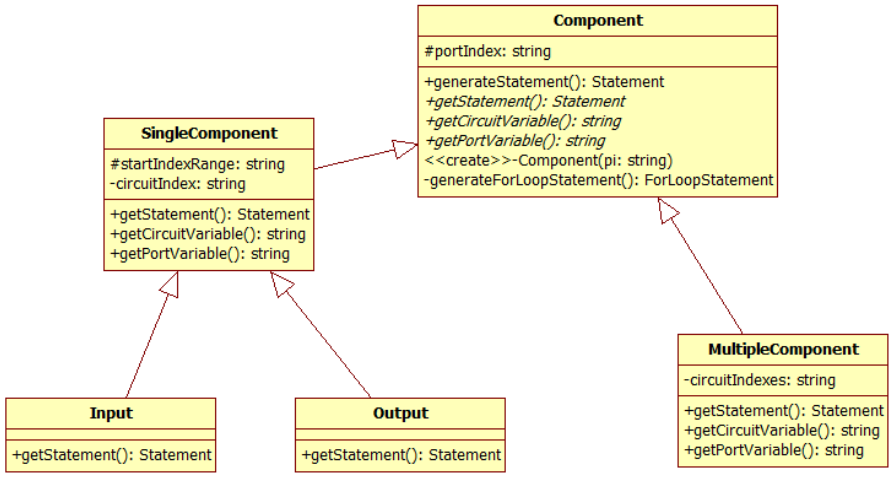

Component
#portIndex: string
+generateStatement(): Statement
+getStatement(): Statement
+getCircuitVariable(): string
+getPortVariable(): string
<<create>>-Component(pi: string)
-generateForLoopStatement(): ForLoopStatement
SingleComponent
#startIndexRange: string
-circuitIndex: string
+getStatement(): Statement
+getCircuitVariable(): string
+getPortVariable(): string
MultipleComponent
-circuitIndexes: string
+getStatement(): Statement
+getCircuitVariable(): string
+getPortVariable(): string
Input
+getStatement(): Statement
Output
+getStatement(): Statement

Figure V.12: Component represents subcircuits and input/output that participate in a connection. It has two classes SingleComponent and MultipleComponent. In comparison to metamodel design, SingleComponent is designed for *SplitRef* in the meta-models that may split its ports into smaller pieces, while MultipleComponent is designed for *ReplicateRef* that they can replicate itself for a group. The Input and Output derives from SingleComponent because they can be split to connect with subcircuits.

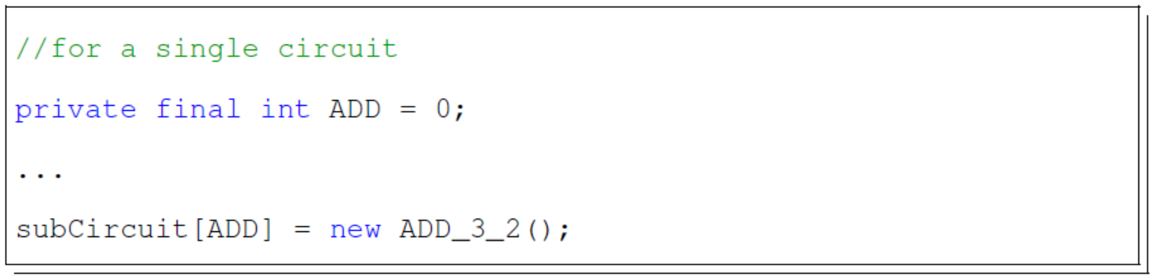
Range class represents the index of circuits and ports. SingleRange denotes a single index only, while MultipleRange can represent a range of indexes. Range is an accessory class for getting the index for both circuit and port. It also participates in generating the Forloop statement.

V.3.3 Comments in Codes

For modules which directly participate in code generations, (e.g., *InitialRef*, *Wire*, *FixedWireRef*, *CompositeCircuit*), we incorporate an description attribute that allow users to add descriptions during the development of GC models. The intent of the description content is to remind the model designer of the functionality of each module. Moreover, the content for the description will be automatically transformed into comments in the codes generated, which should improve the readability of automatically generated code.

V.3.4 circuitID Alias

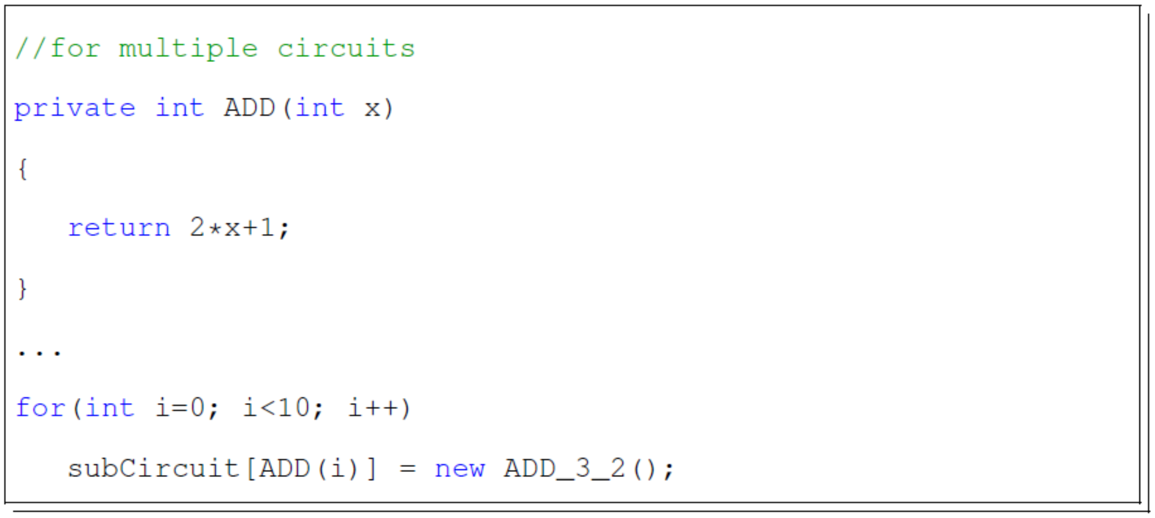
We also improve the readability of codes automatically generated by introducing an alias for subcircuit. Typically, in code for the framework, sub-circuits are aligned in vector-style container that is accessed by its index. This may make it confusing for engineers because they may have difficulty in associating an index with the corresponding a sub-circuit index points to. This problem may occur when the engineer wants to access certain type of sub-circuit, but may not be aware of its index in the vector, or when an engineer sees that the sub-circuit is accessed by its index, but does not know what this subcircuit correspond to.



```
//for a single circuit
private final int ADD = 0;
...
subCircuit[ADD] = new ADD_3_2();
```

Figure V.13: The circuitID alias for a single subcircuit.

The concept of an alias should mitigate this problem by giving a common name that is easy for recognition. With a circuitID alias, we replace the index of the sub-circuit with a



```
//for multiple circuits
private int ADD(int x)
{
   return 2*x+1;
}
...
for(int i=0; i<10; i++)
   subCircuit[ADD(i)] = new ADD_3_2();
```

Figure V.14: The circuitID alias for multiple subcircuits.

special name designated by the engineer. The code template in Figure V.13 and in Figure V.14 provides examples of this concept.

V.4 Summary

In conclusion, we reviewed how code in FastGC is organized and composed. Focused on interfaces the required for implementation and the corresponding code, we demonstrate how each part of the FastGC interpreter design takes part into code generations.

CHAPTER VI

DISCUSSION AND CONCLUSION

In this thesis, we (1) showed that a generic representation of garbled circuit by DSML is feasible, (2) implemented a garbled circuit meta-model on GME platform, and (3) implemented a FastGC interpreter to work with GC models to automatically generate working codes in FastGC framework.

VI.1 Future Work

While this thesis demonstrates that meta-modeling can be applied in GC model development and transformation, there are several limitations that we wish to highlight as opportunities for future research.

First, DSML is a generic approach which efficiently describes the unique functionality of GCs and of specification and transformation to working code. While we incorporated it into the GME modeling platform, this is only one possible environment. A domain-specific modeling language can be a visual diagramming language created by the Generic Eclipse Modeling System [24], a programmatic abstraction created by Eclipse Modeling Framework [25], or even a textual language. As such, the GC meta-model can be implemented on other platforms, particularly considering OS-independent tools given that GME works only on Windows.

We only implemented the FastGC framework to demonstrate the potential for integration of models and interpreters in the automatic GC code generation.. More interpreters can be implemented to make the GC meta-model proposed more pragmatic. On the other hand, we did not consider that the proposed GC language would be capable of representing all types of GCs for all of frameworks. However, we always emphasize the significance of extendability during the whole procedure of development. Potential specification requirement from new GC implementations will also expand the vocabulary of circuit language, and enhance the comprehensiveness in circuit specification.

Appendix A

Circuit Library

The three logic gates are stored in Level 1 which is the lowest level and are showed in Table A.1. Table A.2 shows the GCs in the second level, which is built on three logical gates only. Table A.3 shows the GCs in the third level. Table A.4 shows the GCs in the forth level.

Circuit	Functionality & Description
XOR_2_1	**Functionality**: XOR gate: compare two bit, outputs 1 when inputs are different, otherwise output 0.
	Description: This is the basic gate in circuit with 2 ports as input and 1 port as output.
AND_2_1	**Functionality**: AND gate: outputs 1 when two input ports are both 1; otherwise outputs 0.
	Description: This is the basic AND gate of logical circuit with 2 ports as input and 1 port as output.
OR_2_1	**Functionality**: outputs 0 when two input ports are both 0; otherwise it outputs 1.
	Description: This is the basic OR gate of logical circuit with 2 ports as input and 1 port as output.

Table A.1: GC Library Level 1

Circuit	Functionality & Description
EDT_4_1	**Functionality**: This computes the editing distance of two integers both with two bits.
	Description: This is composed by two XOR_2_1 and one OR_2_1, with four ports as its input and one port as its output.
GT_3_1	**Functionality**: This compares the value of first two bit and use its last input as indicator for greater than or less than.
	Description: This is composed by three XOR_2_1 and one AND_2_1, with three ports as its input and one port as its output.
MUX_3_1	**Functionality**: This is a multiplexer which selects one of its first two digital input signals and forwards the selected input as output into a single line. The thirds input port serves as indicator for the selection.
	Description: This is composed by two XOR_2_1 and one AND_2_1, with three ports as its input and one port as its output.
XOR_2L_L	**Functionality**: This computes the exclusive disjunction of two integers both with L bits.
	Description: This is composed by a number of L paralleling XOR_2_1 with 2L ports as its input and L port as its output.
OR_L_1	**Functionality**: This computes the logical disjunction of L bits.
	Description: This is composed by a number of L-1 series OR_2_1 with L ports as its input and 1 port as its output.
XOR_L_1	**Functionality**: This computes the logical exclusive disjunction of L bits.
	Description: This is composed by a number of L-1 series XOR_2_1 with L ports as its input and 1 port as its output.
ADD_3_2	**Functionality**: This computes the add or subtract of the first two bits with the third input port as indicator.
	Description: This is composed by four XOR_2_1 and one AND_2_1 with 3 ports as its input and 2 port as its output.

Table A.2: GC Library Level 2

Circuit	Functionality & Description
EDT_2L_1	**Functionality**: This compares two integer with both L bit, output 0 when equal; otherwise outputs 1.
	Description: This is composed by a number of L paralleling XOR_2_1 and one OR_L_1(L) with 2L ports as its input and 1 port as its output.
ADD_2L_L	**Functionality**: This computes the addition of two integers.
	Description: This is composed by a number of L ADD_3_2, with 2L ports as its input and L port as its output.
SUB_2L_L	**Functionality**: This computes the subtraction of two integers via two-complement and addition.
	Description: This is composed by a number of L ADD_3_2 and one XOR_2L_L, with 2L ports as its input and L port as its output.
ADD_2L_LPLUS1	**Functionality**: This computes the addition of two integers with the carry save.
	Description: This is composed by a number of L ADD_3_2, with 2L ports as its input and L+1 port as its output.
GT_2L_1	**Functionality**: This compares the value of two integers both with L bits.
	Description: This is composed by a number of L GT_3_1, with 2L ports as its input and 1 port as its output.
MUX_2LPLUS1_L	**Functionality**: This is a multiplexer which selects one of its two input integers. The thirds input port serves as indicator for the selection.
	Description: This is composed by a number of L MUX_3_1, with 2L+1 ports as its input and L port as its output.

Table A.3: GC Library Level 3

Circuit	Functionality & Description
MAX_2L_L	**Functionality**: This returns max value of two input integers both with L bits.
	Description: This is composed by one GT_2L_1 and one MUX_2Lplus1_L, with 2L ports as its input and L port as its output.
MIN_2L_L	**Functionality**: This returns min value of two input integers both with L bits.
	Description: This is composed by one GT_2L_1 and one MUX_2Lplus1_L, with 2L ports as its input and L port as its output.
ADD1_LPLUS1_L	**Functionality**: This adds the input integer with L bit by 1.
	Description: This is composed by one ADD_2L_Lplus1, with L+1 ports as its input and L port as its output.
ADD1_LPLUS1_LPLUS1	**Functionality**: This adds the input integer with L bit by 1 with a carry saver.
	Description: This is composed by one ADD_2L_Lplus1, with L+1 ports as its input and L+1 port as its output.

Table A.4: GC Library Level 4

Appendix B

Class InitialtorStatement Implementation

Here, we use the class InitialtorStatement as an example, to illustrate the process of initialization. We know that an initialization sentence of codes needs parameters which includes circuitId, className, and arguments for sub-circuits. Thus the InitialtorStatement can be declared as follows:

```
class InitialtorStatement: public Statement {
public:
    //Constructor Function
   InitialtorStatement(string index, vector<string> arg, string
       cn);
   //Output the code, n is the indent
   virtual string toString(int n = 0);
private:
    //indexId of the subcircuit
   string index;
   //arguments subcircuit required to initialize
   vector<string> arguments;
   //the type of subcircuit
   string className;
};
```

And according to the code template of initialization, the toString function can be implemented as follows:

```
string InitialtorStatement:: toString(int n) {
  string indent(n, '\t');
  //code template: "subCircuits[circuitId]"
  string output = indent+"subCircuits["+index;
  //code template: "= new circuitName("
  output += "] = new "+className+"(";
  //the arguemnts list
  int size = arguments.size();
  for(int i=0; i<size; i++)
  {
    output += arguments[i];
    if(size-1 != i)
      output += ", ";
  }
  output += ");\n";
  return output;
}
```

BIBLIOGRAPHY

[1] Silvio Micali and Phillip Rogaway. Secure computation. In *Advances in Cryptology-CRYPTO91*, pages 392–404. Springer, 1992.

[2] Markus Jakobsson and Ari Juels. Mix and match: Secure function evaluation via ciphertexts. In *Advances in CryptologyASIACRYPT 2000*, pages 162–177. Springer, 2000.

[3] Ioannis Ioannidis and Ananth Grama. An efficient protocol for Yao's millionaires' problem. In *Proceedings of the 36th Annual Hawaii International Conference on, System Sciences.*, pages 6–pp, 2003.

[4] Shafi Goldwasser. Multi party computations: past and present. In *Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, pages 1–6. ACM, 1997.

[5] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *27th Annual Symposium on, Foundations of Computer Science.*, pages 162–167, 1986.

[6] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *Sigplan Notices*, 35(6):26–36, 2000.

[7] Akos Ledeczi, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle, and Peter Volgyesi. The generic modeling environment. In *Workshop on Intelligent Signal Processing*, volume 17, page 1, 2001.

[8] Mikhail J Atallah and Wenliang Du. Secure multi-party computational geometry. In *Algorithms and Data Structures*, pages 165–179. Springer, 2001.

[9] Wei Jiang, Chris Clifton, and Murat Kantarcıoğlu. Transforming semi-honest protocols to ensure accountability. *Data & Knowledge Engineering*, 65(1):57–74, 2008.

[10] Yehuda Lindell and Benny Pinkas. A proof of security of Yao's protocol for two-party computation. *Journal of Cryptology*, 22(2):161–188, 2009.

[11] Dahlia Malkhi, Noam Nisan, Benny Pinkas, Yaron Sella, et al. Fairplay-secure two-party computation system. In *USENIX Security Symposium*, 2004.

[12] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security Symposium*, volume 201, 2011.

[13] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY–a framework for efficient mixed-protocol secure two-party computation. In *Network and Distributed System Security*, 2015.

[14] Ayush Rastogi, Matthew Hammer, Michael Hicks, et al. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *IEEE Symposium on*, pages 655–670. IEEE, 2014.

[15] Xiao Shaun Wang, Chang Liu, E Shi, Yan Huang, and K Nayak. Oblivm: A generic, customizable, and reusable secure computation architecture, 2014.

[16] Fabien Latry, Julien Mercadal, and Charles Consel. Processing domain-specific modeling languages: A case study in telephony services. In *Generative Programming and Component Engineering for QoS Provisioning in Distributed Systems*, 2006.

[17] José Raúl Romero, José Eduardo Rivera, Francisco Durán, and Antonio Vallecillo. Formal and tool support for model driven engineering with maude. *Journal of Object Technology*, 6(9):187–207, 2007.

[18] Vladimir Dimitrieski, Milan Čeliković, Vladimir Ivančević, and Ivan Luković. A comparison of ecore and gopprr through an information system meta modeling approach. 2012.

[19] Robert France and Bernhard Rumpe. Domain specific modeling. *Software and Systems Modeling*, 4(1):1–3, 2005.

[20] Laung-Terng Wang, Yao-Wen Chang, and Kwang-Ting Tim Cheng. *Electronic design automation: synthesis, verification, and test*. Morgan Kaufmann, 2009.

[21] Dirk Riehle. Composite design patterns. In *ACM SIGPLAN Notices*, volume 32, pages 218–228, 1997.

[22] Ákos Lédeczi, Arpad Bakay, Miklos Maroti, Péter Völgyesi, Greg Nordstrom, Jonathan Sprinkle, and Gábor Karsai. Composing domain-specific design environments. *Computer*, 34(11):44–51, 2001.

[23] Pierre F Tiako. *Designing Software-Intensive Systems: Methods and Principles: Methods and Principles*. IGI Global, 2008.

[24] Jules White, Douglas C Schmidt, and Sean Mulligan. The generic eclipse modeling system. In *Model-Driven Development Tool Implementers Forum, TOOLS*, volume 7, 2007.

[25] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.