Code Synthesis Based on Object-Oriented Design Models and Formal Specifications

Xiaoping Jia, Sotiris Skevoulis
Division of Software Engineering
School of Computer Science, Telecommunication, and Information Systems
DePaul University
Chicago, Illinois, U.S.A.
E-mail: `jia@cs.depaul.edu, sskevoul@cs.depaul.edu`

Abstract

This paper presents an approach to synthesizing functional and robust code from object-oriented design models and Z data and operation specifications. The approach used here is based on an integrated notation of the Unified Modeling Language and a slightly extended Z notation to include object-oriented concepts and structures. Our approach generates fully functional code which can be compiled and executed without modifications. The information from object-oriented analysis and design models along with the formal specifications are combined, analyzed, and translated into an intermediate representation from which code can be generated. A research prototype has been developed to demonstrate the feasibility and the effectiveness of our approach.

1. Introduction

Object-oriented approach has matured and has been widely used in the software development industry. Object-oriented languages like C++ [6, 13] and Java [1, 14] have matured. Object-oriented techniques and notations are becoming more and more precise in syntax and semantics. Notable examples include Fusion [4], Syntropy [5] and recently UML [3]. Most of the object-oriented methodologies focus on the analysis and design phases of software development. Recently, object-oriented CASE tools with code generation capabilities such as Rational Rose [2] have become available. However they generate only skeletal code and programmers have to fill in the functionality. On the other hand in the area of formal methods, there are only a few supporting tools that are capable of generating prototype quality code.

In this paper we present an approach that integrates object-oriented methodology and formal methods, to automate many of the tedious and error-prone tasks of the construction phase and generate robust and fully functional C++ code. Our approach uses an automatic light-weight theorem prover to assist in the optimization of the generated code. At the same time it supports animation based on the specifications. The key characteristics of our approach are the following:

Substantial automation Instead of generating *skeletal* or prototype quality code the synthesizer component is capable of generating complete and fully functional C++ code.

Design and implementation choices The code synthesizer allows designers to choose from an extensive collection of architecture, design, and implementation schemes that are proven sound and efficient, and the chosen schemes will be applied automatically.

Conformance and absence of errors The code synthesizer guarantees that the generated code conforms to object-oriented model and is absent of many common types of errors, including memory leaks, dereferencing invalid pointers, object slicing, *etc.*

Integration with manually coded components The generated code can be easily integrated with manually coded C++ components. Iterative development by preserving manually coded components through iterations is also supported.

Our approach is based on a tight integration of UML object-oriented modeling and design method and the Z notation. The code generation mechanism described in this paper is based in the synergetic integration of three key components: the *model analyzer* which analyzes the integrated input models, the *animator* which can animate system behaviors and the *code synthesizer*, which generates robust C++ code. In this paper we briefly discuss and summarize the analyzer and animator components and focus on the *code synthesizer*. The *analyzer* has been described in details in [10]. The *animator* is an extension of ZANS [9] that animates a large subset of Z specifications and generates prototypes from object-oriented models along with their specifications [8].

2. The Model Analyzer

The model analyzer performs extensive analysis on both object-oriented design models and formal specifications written in Z. The analyzer performs the following tasks:

1. Translating UML diagrams into Z segments.
2. Combining the translated Z segments with Z operation and data specifications.
3. Extensively analyzing combined and sorted specifications.
4. Performing consistency analysis on UML diagrams.

The goal of model analysis is not to guarantee model consistency and completeness, but to identify anomalies in early stages.

2.1. The Translation Process

In our approach we use as specification notation the Augmented Modeling Language [10] . It is an attempt to integrate two existing notations: Unified Modeling Language (UML) [3] and Z [12].

The integration of the two notations UML and Z is based on the complementary use of information extracted from each one of them and the retainment of as much as possible of the original notations, in order to avoid unnecessary complications and confusion. The integration is accomplished by mapping the information provided in UML to a slightly extended Z notation called *Zext* [10].

2.2. Analyzing the Modeling Elements

The schemas that resulted from the mapping process become the system's modeling elements. In such a model, there are certain schema dependencies that must hold in order for the model to maintain its consistent state. Such dependencies include relations among classes (associations, inheritance *etc.*) and relations among global names that need to follow the definition before use principle. The *dependency checking* is performed at this stage in order to detect various anomalies such as *cyclic schema dependency*.

2.3. Analyzing Operations Specifications

The operation schemas in Z provide details about the functionality of the system's operations. The key to our code generation approach is the classification of operation schemas into two categories: *explicit* and *non-explicit*. Code can be generated from explicit operation schemas. Informally, an operation schema is *explicit* if the values of all its output variables, including post state, can be determined by evaluating some of the expressions in the schema. Explicit operation schemas can be translated into an intermediate format (*Extended Guarded Command* - EGC) [8] before the code generation process starts. The expressions in EGC may involve mathematical objects that do not exist in most programming languages such as sets, relations, functions, bags *etc.* For that reason, a special C++ library (ZLIB) has been developed.

3. The ZLIB

Z offers a rich collection of mathematical objects that do not have counterparts in imperative languages like C++ , such as sets, relations, sequences, bags, *etc.* A specialized class library is needed in order to provide implementations for those mathematical objects in Z.

ZLIB supports *all* the mathematical objects and their operations defined in Z. Specifically, the data types in Z and their counterparts in ZLIB have as follows:

Z Data Types	ZLIB Abstract Classes
• Sets	• ZSet
• Numbers	• ZInt
• Relations	• ZRelation
• Functions	• ZFunction
• Sequences	• ZSequence
• Bags	• ZBag

The class hierarchy of ZLIB library conforms to the one defined in Z Mathematical Tool-kit [12]. The ZLIB consists of the following components:

- ZLIB classes:
  - **abstract classes**: These class directly correspond to the Z mathematical objects. They define complete interfaces for all the Z mathematical objects.
  - **concrete classes**: They are derived classes of the abstract classes. They provide various implementations of the Z mathematical objects.
  - **auxiliary classes**: They include iterator classes, *etc.*
- Implementation classes
  They include a variety of implementations for linked list, chained and open address hash table, bit string and bit matrix and AVL trees.

The hierarchy of the abstract and concrete classes is shown in the following class diagram. We use italic fonts to denote abstract classes and upright fonts for the concrete implementations.

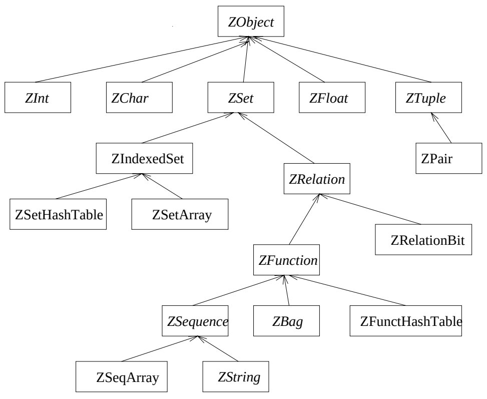

Figure 1. Partial Class Hierarchy of ZLIB

For example the *Set* data type in Z has as a counterpart in ZLIB, the `ZSet`. A Z statement involving the membership of a *set_member* in the union of two sets *SetA* and *SetB* could be the following:

$$set\_member? \notin SetA \cup SetB$$

This is translated to the following with the use of ZLIB:

```
notin(set_member, Union(SetA , SetB))
```

An important feature of *ZLIB* is that for each of the abstract data type, it provides a variety of implementations in the form of concrete subclasses. For abstract data type *ZSet* implementations such as *ZSetArray* and *ZSetHashTable* are provided. ZLIB has been designed and implemented using *Design Patterns* [7]. Object instantiation process is implemented using a combination of creational patterns like *Abstract Factory*, *Prototype* and *Singleton*. This scheme offers the following advantages:

- it reveals to the client only the interface of classes, hiding at the same time their implementation.
- flexibility in changing from one concrete implementation class to another at run time by registering prototypical instances of these classes with the client code.

4. The Code Synthesizer

The generated C++ source code contains the following:

- declarations and implementations of all the classes in the object-oriented analysis and design models, including the implementations of all member functions and associated state diagrams;
- an `Application` class that initializes, runs, and cleans up the entire system;
- a `main` function that creates a single instance of the `Application` class and runs the application.

Global Initializations. The following rules apply to the initialization of global variables:

- The *axiomatic definitions* in Z specification are transformed into enumeration types in C++ ; The constraint part is not used during the mapping but it is considered later as one of the invariant properties of the system.

$SomeDecl : SomeType$ `extern`*SomeType SomeDecl*;

- *free types* are transformed to global variables. If the basic type is T and some new variables of this type are $c_1, c_2, \ldots c_n$ then the transformation is described by:

$T ::= c_1 \mid c_2 \mid ... \mid c_n$ `enum` T `{` c_1, c_2, ... c_n, `};`

- Input and output overloaded operators are generated for each class and type.

```
ostream& operator<<(ostream&, const T);
istream& operator>>(istream&, T& );
```

- The main routine is being initialized and a pointer type of the application is declared.

```
int main(int argc, char* argv[]) {
  theApp = new ApplicationName();
  theApp->AppRun(argc, argv);
  delete theApp;
}
```

Synthesis of Class Interface. Class interface is synthesized according to the following:

- Synthesize code for class methods that store and retrieve class attributes like, for example, *put_attr* and *get_attr*;
- Generate class constructor(s) using the initialization schema provided by the specifications;
- Generate method declarations based on the specifications

For sets and functions we have specific template classes that provide the particular implementation. The translation from the Z types to the C++ data types is based on the following rules:

- Numbers (N and Z) are translated into `ZInt`
- $\mathbb{P}\,S$ where S is of some type, is translated into a template class `set<S>`
- $A \pfun B$, where A and B are of some type T is translated into a template class `function<A,B>`

The following generic example shows the transformation process from the Z schemas to the C++ declarations. *ClassA* is a class state schema and $Op1$ is an operation declared in the class:

$$\begin{array}{|l}\hline \textit{ClassA} \\ \textit{attr}_1 : \textit{ZType}_1 \\ \ldots \\ \textit{attr}_n : \textit{ZType}_n \\ \hline \textit{pred}_1 \\ \ldots \\ \textit{pred}_n \\ \hline \end{array} \qquad \begin{array}{|l}\hline \textit{Op}1 \\ \Delta\,\textit{ClassA} \\ \textit{input}? : \textit{ZType}_2 \\ \textit{response}! : \textit{Response} \\ \hline \textit{oppred}_1 \\ \ldots \\ \textit{oppred}_j \\ \hline \end{array}$$

The information extracted from the above specification is used to synthesize the interface for this particular class `ClassA`. Each one of the $\textit{ZType}_n$ in the above schemas is translated to the corresponding data type *CPPType* and is included in the class interface. *CPPType* can be either primitive (integer, characters, *etc.*) or composite (set, sequence, bag *etc*). In *italics* we present the changeable parts of the template form:

```
class ClassA {
public:
      ClassA ();
      ClassA(const ClassA&);
      virtual  ClassA();
      const CPPType < Type1 >&
          get_attr_1(void) const
              { return attr_1;  }
      void put_attr_1 (CPPType < Type1 >& attr_1 )
          { ClassA::attr_1 = attr_1 ; }
      Response Op1(const& Type1 input );
      ...
protected:
      CPPType < Type1 > attr_1;
      ...
      CPPType < Typen > attr_n;
}
```

Virtual destructor, *copy constructor*, and the *equality* and *assingment operators* are also generated.

Synthesis of Class Operations Implementation. Each class operation in Z specification is described as a conjunction of several schemas, each one describing a possible situation. The operation's implementation is generated from the extended guarded command. The code generation involves translation of Z entities into their counterparts in C++ using the ZLIB.

Synthesis of State Transition Diagrams. An important feature of our approach is the synthesis of code from state transition diagrams (STD). The code synthesis for state transition diagrams is supported by the following class library:

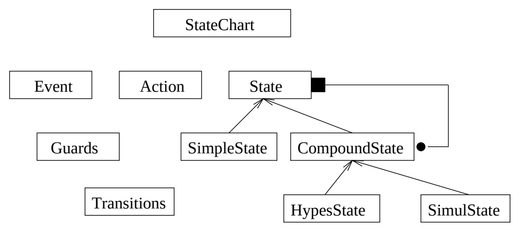

Figure 2. STD Class Library

The classes in this library provide the necessary implementations for all the elements of a STD. For some state *Somestate*, of a STD which includes any number of states and is part of an application named *MyApp*, the code is generated automatically as follows: First an instance of the `Statechart` class is included in the generated code for the class *MyApp*. `Statechart` is a class that represents the state transition diagram and contains instances of states, a queue of events and a list of states. *MyApp's* interface includes an `EnqueueSTDEvent(Event*)` method that enqueues the events as they happen and an `InitializeSTD()`

method which initializes the STD. Part of the interface for the class *MyApp* that is synthesized, is shown here:

```
class MyApp {
public:
  AppRun();
  void EnqueueSTDEvent(Event* e);
  ...
protected:
  Statechart* theSTD
  void _initializeSTD();
  ...
};
```

The parsing of a STD, generates the following code. We present the initialization of the STD into separate code segments that correspond to each one of the STD elements:

States: Each one of the states described in the diagram is generated and inserted into the applications's class state.

```
void MyApp::initializeSTD(void) {
  SimpleState* MyAppSomeState =
        new SimpleState("MyAppSomeState");
  MyApp_top->AddState( MyAppSomeState );
  ...
```

Transitions: Each transition from the starting to the ending state, is created and inserted to the starting state.

```
  Transition* t;
  t = new Transition(MyAppSomeState,MyAppSomeState,
        "MyApp_SomeState_TO_MyApp_SomeState");
  MyAppSomeState->AddTransition(t);
```

Actions: The actions in the STD are associated with the approapriate events.

```
  Action* a;
  a = new ClassAction<MyApp>
          (this, & MyApp::SomeAction);
  t->SetAction(a); }
```

Events: All the events and their associated actions are monitored, grouped and registered with the appropriate state.

```
  Event* e;
  e = new EventGroupOrg(SomeEvent, "SomeAction");
  t->SetEvent(e);
}
```

Each event is put in a queue that is contained in the `MyApp` class's state with the following segment of code:

```
void MyApp::EnqueueSTDEvent(Event* e) {
  theSTD.EnqueueEvent(e);
  theSTD.Run();
}
```

Based on this mapping and the supporting STD library, the code that represents the transitions between the various system states is executed. The code monitors all triggering events, enqueues the events as they happen, associates them with the appropriate actions and displays the entire listing to the user. Such a mechanism, increases significantly the understanding of system's behavior from the user's point of view.

Generating User Interface. Our approach includes a mechanism to generate a variety of user interfaces. At this point we support options of either text based (possibly with menus) or graphical user interface. We currently support the automatic code generation for a variety of motif widgets like pop-up windows, labels, pull-down menus, buttons *etc.* Each GUI widget has been specified using *Zext* notation.

The user only needs to provide a few specific information about the characteristics of the GUI, such as the size of the windows, the name and dimensions of the buttons *etc.* For example, in order to generate window and button widgets, the user has to enter their location and dimensions as follows:

```
Window top [width=num1, height=num2,
        title = \str{MyWindow}]
Button btnSomeAction [x,num2,y,num3,
        width,num4,string \str{SomeAction}]
```

where num_i are numbers that specify the size and location of the widgets and *string* represents the names attached to the widgets . Based on this information, the following code that builds the widgets and sets up the interface for the data exchange, is generated *automatically.* The task of building the various widgets starts with the following segment of the generated code which is part of the method `AppRun` in *MyApp* class:

```
void MyApp::AppRun(int argc, char* argv[])
{
  BuildWidgets(argc, argv);
  setupInterfaceWidgets();
  WindowMainLoop();
}
```

The process of constructing the widgets is conceptually the same for all the different kinds of them. The user provides the dimensions and the position for ie. a button with the numbers num_2 , num_3, num_4 then within the `BuildWidgets` method we have:

```
btnSomeAction =
  XtVaCreateManagedWidget("btnSomeAction",
  xmPushButtonWidgetClass, workingarea,
  XmNx, num2  , XmNy, num3 , XmNwidth, num4 ,
  XmNlabelString, xmstr0, NULL);
```

Our approach integrates the code that is generated for the STD with the GUI code. This integration is achieved by accosiating the widget with the appropriate event and registering the event in the queue of events for the corresponding transition. The following segment performs this association:

```
void OnButton_btnSomeAction(Widget widget,
     XtPointer client_data, XtPointer call_data) {
  Event* e;
  e = new Event(ButtonClick);
  e->originator = "btnSomeAction";
  theApp->_EnqueueSTDEvent(e);
}
```

The user is given the opportunity to visually inspect the transitions made by the code that is currently executed. This feature assists in rapid prototyping.

5. Conclusions

In this paper, we presented a code generation approach which is based on the analysis of object-oriented design models and formal specifications. Our approach offers a substantial automation to the construction phase of software development, by synthesizing fully functional and robust code. It provides a number of choices for different design and implementation schemes that have been proven sound and efficient and the software developer can choose the best ones for the problem at hand. The chosen schemes are applied automatically. The code that is synthesized is guaranteed to be free of many difficult to trace bugs such as dereferencing an invalid pointer, memory leaks, object slicing, *etc.* A prototype which fully demonstrates the capabilities and feasibility of our approach has been developed [11].

6. Future Work

We continue to work in the automatic code generation area towards a variety of directions. We are working on enhancements that:

- facilitate the customization of design refinement in data and algorithms.
- integrate the tool with other libraries available like the *Standard Template Library.*
- optimize the generated code.

Our work continues in the development of a more complete, capable and usable code generation approach.

7. Acknowledgments

This work has been partly sponsored by the National Science Foundation under contract CCR-9410236.

References

[1] K. Arnold and J. Gosling. *The Java™ Programming Language.* Addison-Wesley, 1996.

[2] G. Booch and J. Rumbaugh. *Rational Rose*, 1997. Available from Rational, Inc.

[3] G. Booch and J. Rumbaugh. *Unified Methods for Object-Oriented Development, Documentation Set, Version 1.1*, 1997. Available from Rational, Inc.

[4] D. Coleman et al. *Object-Oriented Development — The Fusion Method.* Prentice Hall, 1994.

[5] S. Cook and J. Daniels. *Designing Object-Oriented Systems: Object-Oriented Modelling with Syntropy.* Prentice-Hall, 1994.

[6] M. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[7] E. Gamma et al. *Design Patterns — Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[8] X. Jia. An approach to animating Z specifications. In *Proc. 19th Annual Int'l Computer Software and Applications Conf.*, Dallas, Texas, USA, Aug. 1995.

[9] X. Jia. *A Tutorial of ZANS — A Z Animation System*, June 1995. Available via anonymous ftp at `ise.cs.depaul.edu`.

[10] X. Jia. A pragmatic approach to formalizing object-oriented modeling and development. In *Proc. 21st Annual Int'l Computer Software and Applications Conf.*, pages 240–245, Washington, D.C., USA, Aug. 1997.

[11] X. Jia and S. Skevoulis. Venus: An object-oriented development tool. Technical report, School of Computer Science, Telecommunication and Information Systems, DePaul University, 1997. Available via anonymous ftp at `ise.cs.depaul.edu`.

[12] J. Spivey. *The Z Notation, A Reference Manual.* Prentice Hall International, second edition, 1992.

[13] B. Stroustrup. *The Design and Evolution of C++.* Addison-Wesley, 1994.

[14] T. J. Team. *The Java™ Application Programming Interface, Volumn II, Window Toolkit and Applets.* Addison-Wesley, 1996.