Round-trip Software Engineering with CodeDesigner RAD

MICHAL BLIŽŇÁK, TOMÁŠ DULÍK, ROMAN JAŠEK
Tomas Bata University in Zlin, Faculty of Applied Informatics
Department of Informatics and Artificial Intelligence
nám. T.G.Masaryka 5555, 760 01 Zlin
Czech Republic
1bliznak@fai.utb.cz, 2dulik@fai.utb.cz, 3jasek@fai.utb.cz

Abstract: Round-trip source code engineering is often an integral part of modern CASE tools. It allows programmers to design their applications in high-level, intuitive and self-documenting way. This paper shows principles and algorithms used in new open-source cross-platform CASE tool called CodeDesigner RAD developed at Tomas Bata University suitable for production-ready source code generation and reverse engineering which allows users to generate complete C/C++ applications from a formal visual description or for creation of UML diagrams from an existing source code.

Key–Words: RAD, CASE, UML, code generation, C/C++, production-ready, cross-platform, CodeDesigner, reverse, engineering

1 Introduction

Nowadays, there exist many software development tools able to generate source code of basic application skeleton from its formal description. Unfortunately, in many cases these tools lack an ability to generate complete, production-ready source code or to import an existing source code as its formal visual description. This paper shows principles and algorithms used in cross-platform development tool called CodeDesigner RAD [3] aimed for production-ready source code generation and reverse engineering which allows users to generate complete C/C++ applications from a formal description based on UML diagrams or to make complete round-trip software engineering.

2 Source Code Generation in Detail

The source code generation process consists of four steps. First of all a source diagram can be preprocessed so its topology will change. This task should produce a new diagram more suitable for further processing by the next code generator components. Preprocessed diagram must be verified to find possible inconsistencies. If the verification fails the code generation process is aborted. After that, a set of optimizations leading to various simplifications can be performed on the verified diagram. The last step represents a final generation of a source code from verified and optimized diagram. This task is performed by a functional object called generator.

The generator reads the structure of modified diagram and writes source code fragments accordingly to the used code generation algorithm to output source file(s). Code generation algorithms can be filtered by output programming language since some language don’t have to support all command statements produced by the algorithm (e.g. switch command statement is supported in C/C++ but not in Python).

Code generation algorithms use so called element processors which provide symbolic code tokens for processed diagram elements. These symbolic tokens are converted into textual code fragments by language processors with syntax in accordance to the used output programming language specification. The complete structure of source code generator implemented in CodeDesigner RAD is shown in Figure 1.

Figure 1 Structure of code generator
2.1 Class Diagram Code Generator

The class diagram is the main building block of object-oriented modeling. It is used both for general conceptual modeling of the systematics of the application, and for detailed modeling translating the models into programming code. The classes in a class diagram represent both the main objects and interactions in the application and the objects to be programmed.

In the design of a system, a number of classes are identified and grouped together in a class diagram which helps to determine the static relations between those objects. With detailed modeling, the classes of the conceptual design are often split into a number of subclasses. In order to further describe the behavior of systems, these class diagrams can be complemented by state charts as shown in chapter 2.2.

In addition to the classic classes the class diagram can contain also elements representing class templates and enumerations.

The relations between class objects in the class diagram called associations and aggregations are defined in [1].

For instance, Figure 2 shows an interface (abstract class), a base class implementing the interface and a class inheriting the base class. Also virtual functions and destructor are illustrated there.

![Class Diagram](image)

**Figure 2 Class inheritance**

The following listing shows mapping of the class diagram in Figure 2 to C++ source code.

```cpp
class Interface {
public:
    virtual void method() = 0;
};

class BaseClass : public Interface {
public:
    virtual void method();
};

class InheritedClass : public BaseClass {
public:
    virtual ~InheritedClass();
    void BaseClass::method() {
    }
    void InheritedClass::method() {
    }
    InheritedClass::~InheritedClass() {
    }
};
```

2.2 State Chart Code Generator

UML state chart is a significantly enhanced realization of the mathematical concept of a finite automaton [9] in Computer Science applications as expressed in the Unified Modeling Language notation [1].

The concepts behind this are about organizing the way a device, computer program, or other (often technical) process works such that an entity or each of its sub-entities are always in exactly one of a number of possible states and where there are well-defined conditional transitions between these states. UML state machine, known also as UML state chart, is an object-based variant of Harel state chart [8] adapted and extended by UML. UML state machines overcome the main limitations of traditional finite-state machines while retaining their main benefits. UML state charts introduce the new concepts of hierarchically nested states and orthogonal regions, while extending the notion of actions. UML state machines have the characteristics of both Mealy machines and Moore machines [9] defined as (1). They support actions that depend on both the state of the system and the triggering event, as in Mealy machines (2), as well as entry and exit actions, which are associated with states rather than transitions, as in Moore machines (3).

Both Mealy and Moore machines are a 6-tuple,

\[(S, s_0, \Sigma, \Lambda, T, G)\]  

, consisting of the following:

- a finite set of states \((S)\)
- a start state (also called initial state) \(s_0\) where \(s_0 \in S\)
- a finite set called the input alphabet \((\Sigma)\)
- a finite set called the output alphabet \((\Lambda)\)
- a transition function \((T : S \times \Sigma \rightarrow S)\) mapping pairs of a state and an input symbol to the corresponding next state.
An output function of Mealy machine is defined as follows
\[ G : S \times \Sigma \rightarrow \Lambda \] (2)
It maps pairs of a state and an input symbol to the corresponding output symbol. In contrast to the Mealy machine a Moore machine’s output function
\[ G : S \rightarrow \Lambda \] (3)
maps each state to the output alphabet.

From the code generation point of view the symbols of both the input alphabet \( \Sigma \) and the output alphabet \( \Lambda \) can be mapped to user-defined code snippets covering the platform-dependent or implementation-specific functionality. User-defined conditional statements or functions returning boolean/numerical values can be regarded as symbols of \( \Sigma \) while the user-defined actions (i.e. source code fragments or invokable methods) can be regarded as symbols of \( \Lambda \).

A projection of state chart describing an application logic into a source code is involved by used code generation algorithm. There exist number of the algorithms which differ in used program flow controlling command statements, coding style and extent of produced source code. Some of them produce state tables hard to read by humans but saving the disk space while the other ones write sequence of conditional statements and composed commands which take much more spaces but can be easily read or modified by the programmer.

CodeDesigner RAD supports three code generation algorithms provided by state chart code generator: **Loop-case** algorithm, **Else-If** algorithm and **Go-To** algorithm. All of them are optimized for production of easily readable and modifiable source code. For illustration see an output of Loop-case algorithm processing state chart shown in Figure 3.

**Loop-case** algorithm produces highly structured source code easily readable and maintainable by humans. Another advantage is that **switch-case** command sequence allows to optimize number of iterations of the main application loop implementing the state chart behavior. It is possible by omitting of **break** command statements used for separation of the switch cases like shown in Listing 2 where the **break** command is missing between states **ID_INITIAL**, **ID_HELLO** and **ID_WAIT_FOR_ENTER_KEY**.

Listing 2 Loop-case algorithm output

```c
STATE_T Hello_World( )
{
    STATE_T state = ID_INITIAL;
    for( ; ; ) {
        switch( state ) {
            case ID_INITIAL : {
                sayHello();
                state = ID_HELLO;
            }
            case ID_HELLO : {
                askForENTER();
                state = ID_WAIT_FOR_ENTER_KEY;
            }
            case ID_WAIT_FOR_ENTER_KEY : {
                if ( ! ( isEnter() ) ) {
                    readKey();
                    state = ID_FINAL;
                }
            }
            else {
                state = ID_FINAL;
            }
            break;
        }
    }
    return ID_FINAL;
}
```

CodeDesigner RAD allows each diagram to be processed by a different code generation algorithm so it is completely up to the user which one he will use for a specific diagram.

### 3 Reverse Source Code Engineering

In addition to discussed source code generation capabilities modern CASE tools like Visual Paradigm [6], Enterprise Architect [2] or CodeDesigner RAD offers also source code reverse engineering functionality. As stated in [4] the "Reverse engineering is the process of analyzing a subject system to create representations of the system at a higher level of abstraction" or it can
also be seen as "going backwards through the development cycle" [7].

The substance of the reverse engineering process in CASE tools is an analysis of existing source code and visualization of its static structure and implemented application logic. This chapter will focus mainly to reverse engineering capabilities provided by CodeDesigner RAD tool.

### 3.1 Source Code Analysis

Static source code analysis can cover both source code structure and application logic.

There exist plenty of software tools able to perform those analysis. For instance, Exuberant CTAGS [5] command line tool can be used for generation an index (or tag) file of language objects found in source files. Software tools able to analyze behavioral aspects of examined source code are for instance Ablegold Computer’s Easystucture tool [11] and C Algorithm Viewer tool [10]. Both of them can then visualize the program flow as a flowchart.

In the current version, the CodeDesigner RAD v1.5.4 supports static source code structure analysis provided by externally called CTAGS utility. In addition to CTAGS functionality the CodeDesigner RAD enhances the code objects’ import so also source code implementing the functions bodies can be imported into to the CASE tool which is crucial for complete round-trip code engineering implementation as discussed in Chapter 3.2.

Now lets see an example of analyzed C++ source code (Listing 3) and class diagram created from them by CodeDesigner RAD (Figure 4). The following sample code shows an ability to import classes where multiple inheritance is used. It also illustrates how referenced classes are associated in the created class diagram.

![Figure 4 Class sample 1](image)

As can be seen in Figure 4 the class Butterfly inherits two base classes Animal and Being. The Butterfly class also contains a pointer to Data class as its member so the uni-directional association leading referenced Data class is created in the diagram.

### 3.2 Round-trip Code Engineering

Round-trip engineering refers to the ability of a CASE tool to perform code generation from models, and model generation from code (a.k.a., reverse engineering), while keeping both the model and the code semantically consistent with each other. Is is a functionality of software development tools that synchronizes two or more related software artifacts such as source code, models, configuration files, and other documents.

Round-trip engineering process consists of several repetitive tasks as illustrated in Figure 5.

```cpp
class Data {
protected:
    int i;
    const char *str;
};

class Being {
};

class Animal {
public:
    Animal() { ; }
    Animal(const char* specie) { ; }
    ~Animal() { ; }
    const char* GetSpecie() const { ; }
};

class Butterfly : public Animal, public Being {
public:
    Butterfly() { ; }
protected:
    Data *m_pData;
};
```
Figure 5 Round-trip engineering

Now let’s look how the round-trip source code engineering can be done by using CodeDesigner RAD.

1. Suppose existing simple "Hello World" application written in C++ programming language saved in "main.cpp" source file as follows:

Listing 4 Original source code

```cpp
#include <stdio.h>
int main(int argc, char **argv) {
   return 0;
}
```

2. Create empty CodeDesigner RAD project and save it in the same location like the C++ source code above. In the project settings, set Output directory to ".", Base file name to "main" and Code items file name to "functions".

3. Create a package with one class diagram called Presenter classes. Create Presenter class with void sayHello() method in the diagram. At this point do not define the sayHello()’s function body from within the CodeDesigner RAD.

4. Set C++ language for the code generator and run the code generation process. As a result 3 new source files main.h, functions.h and functions.cpp are created and existing main.cpp file is modified. The main.h/cpp source files contain Presenter class declaration/definition and functions.h/cpp would contain other generic functions and variables potentially defined in the project. The content of the main source files is shown below.

Listing 5 main.h

```cpp
/* [ 'Common headers' begin (DON’T REMOVE THIS LINE!) ] */
#include "functions.h"
/* [ 'Common headers' end (DON’T REMOVE THIS LINE!) ] */
/* [ 'Presenter classes' begin (DON’T REMOVE THIS LINE!) ] */
class Presenter {
   public:
      void sayHello( );
   protected:
      private:
   };
/* [ 'Presenter classes' end (DON’T REMOVE THIS LINE!) ] */
```

Listing 6 main.cpp

```cpp
/* [ 'Common headers' begin (DON’T REMOVE THIS LINE!) ] */
#include "main.h"
/* [ 'Common headers' end (DON’T REMOVE THIS LINE!) ] */
#include <stdio.h>
int main(int argc, char **argv) {
   return 0;
}
/* [ 'Presenter classes' begin (DON’T REMOVE THIS LINE!) ] */
void Presenter::sayHello( ) {
   /* [ 'Presenter::sayHello' begin ] */
   /* [ 'Presenter::sayHello' end ] */
} /* [ 'Presenter classes' end (DON’T REMOVE THIS LINE!) ] */
```

5. Modify generated main.cpp file to add some required functionality. Print "Hello World!" message from sayHello() function which will be in-
voked from the Presenter class instance created in the main() function as shown in Listing 7:

```
Listing 7 main.cpp

/* [‘Common headers’ begin (DON’T REMOVE THIS LINE!)] */
#include "main.h"
/* [‘Common headers’ end (DON’T REMOVE THIS LINE!)] */
#include <stdio.h>

int main(int argc, char **argv)
{
    Presenter p;
    p.sayHello();
    return 0;
}
/* [‘Presenter classes’ begin (DON’T REMOVE THIS LINE!)] */
/* public function members of ’Presenter’ class */
void Presenter::sayHello()
{
    /* [‘Presenter::sayHello’ begin] */
    printf("Hello World!\n");
    /* [‘Presenter::sayHello’ end] */
}
/* [‘Presenter classes’ end (DON’T REMOVE THIS LINE!)] */
```

6. Build, run and test the application.

7. Synchronize source files with CodeDesigner RAD project. The content of modified functions managed by CodeDesigner RAD can be synchronized via menu item Code generation->Synchronize code or automatically just before the next code generation step. Note that if new classes were added to the source code then reverse engineering CodeDesigner RAD’s functionality must be used for import of those classes into the CodeDesigner RAD project.

4 Conclusion

As shown in the paper, fully functional, production-ready source code can be generated and complete source code round-trip engineering can be done by using nowadays modern CASE tools like CodeDesigner RAD. The illustrated application development approach has major advantages such it is self-documenting, the application models can be re-used as generic design patterns and at least the application skeleton can be generated by using different programming languages.

Acknowledgements: The research was supported by the European Regional Development Fund under the project CEBIA-Tech No. CZ.1.05/2.1.00/03.0089.acm

References:


