Production-Ready Source Code Round-trip Engineering

Michal Bližňák, Tomáš Dulík, Roman Jašek

Abstracts—Automated source code generation is often present in modern CASE and IDE tools. Unfortunately, the generated code often covers a basic application functionality/ structure only. This paper shows principles and algorithms used in 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.

Keywords—RAD, CASE, UML, code generation, C/C++, production-ready, cross-platform, CodeDesigner, reverse, engineering

I. 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 CASE tool called CodeDesigner RAD [3] aimed for production-ready source code generation and reverse engineering which allows users to generate complete applications from their formal description based on UML diagrams or to make complete round-trip software engineering.

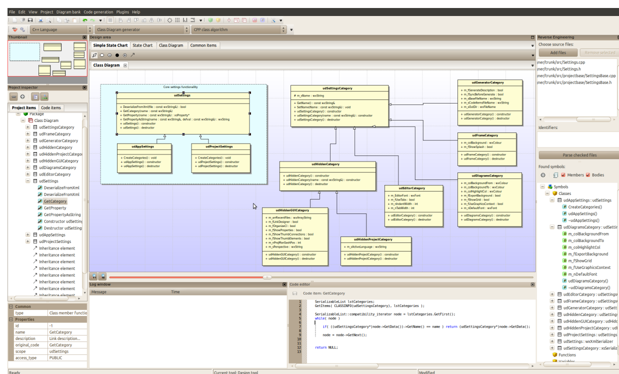

Fig. 1 CodeDesigner RAD

CodeDesigner RAD application shown in Figure 1 has been developed by using well known cross-platform programming toolkit called wxWidgets [14] together with its add-ons wxXmlSerializer [4] and wxShapeFramework [5] so it provides very simple and intuitive way how to graphically describe an application structure and logic on MS Windows and Linux platforms. Moreover, it offers also reverse source code engineering capabilities so user can simply import existing C/C++ or Python source code into the tool.

II. Formal Application Description

For our needs the UML diagrams will be used for describing both application structure and behavior in this article. The reasons are that the UML is well know and widely documented standardized formalism used in many existing applications and CASE tools suitable for both managers and developers.

In general, there are three main types of UML diagrams [1]:

1. Structure diagrams
2. Behavior diagrams
3. Interaction diagrams

Structure diagrams represent the structure, so they are used extensively in documenting the software architecture of software systems.

Behavior diagrams emphasize what must happen in the system being modeled. Since behavior diagrams illustrate the behavior of a system, they are used extensively to describe the functionality of software systems.

The last one subset, interaction diagrams, are a subset of behavior diagrams, emphasize the flow of control and data among the things in the system being modeled.

A. *Static application structure*

One of the possible insights into an application internals is so called *static application structure* view covered by UML structure diagrams. In general, the these diagrams show how the application is divided into logical components or groupings. From the code generation point of view the most important view is **class diagram**. Several class diagrams can be grouped by **package/-component diagrams**.

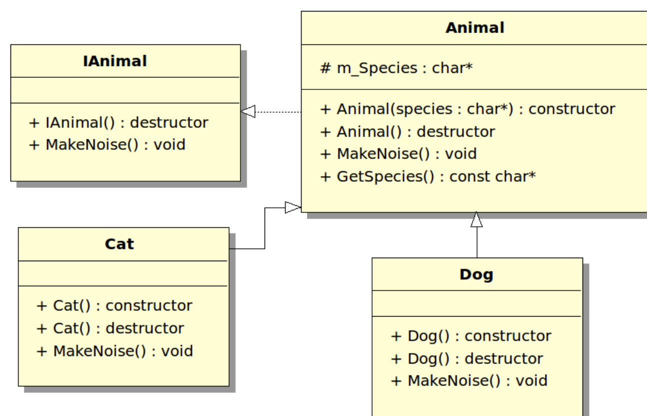

Fig. 2 Class diagram

Figure 2 shows class diagram illustrating properties and relations between four classes: a base class *Animal* which implements *IAnimal* abstract class and two derived classes *Dog* and

Cat inheriting the base *Animal* class. As can be seen also class members are visible in the figure; member variables are shown in the middle class element field and member functions are shown in the bottom field. Also the members' accessibility is indicated by the symbols '+', '-', '#' placed before the member's name where '+' means *public*, '-' means *private* and '#' means *protected*. All the information is crucial for code generation purpose.

B. Dynamic application logic

In contrast to the static application structure view the dynamic application logic view covered by behavior UML diagrams focuses onto description of an application logic. Typically, state charts are used for that purpose. There are several types of state charts like a classic form of state machines described in [11], UML state charts [1] or Harel state charts [10]. All of them are suitable for the code generation purposes but we will focus to state charts used in CodeDesigner RAD tool which are based on Harel state charts with UML-based notation extension which means that entry/exit actions from UML specification as well as transition conditions, events and actions from Harel state charts are supported.

Thanks to that approach an application logic can be described in very intuitive way and CodeDesigner RAD tool can generate optimal production-ready source code covering full application logic and strucure; not only the application skeleton.[1]

Following Figure 3 shows how a simple *"Hello World"* code can be described by UML state chart.

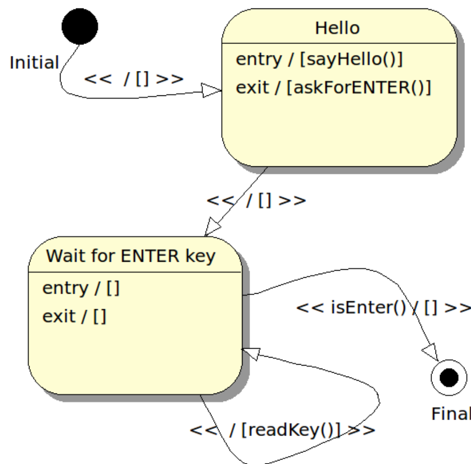

Fig. 3 State Chart

III. 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 4.

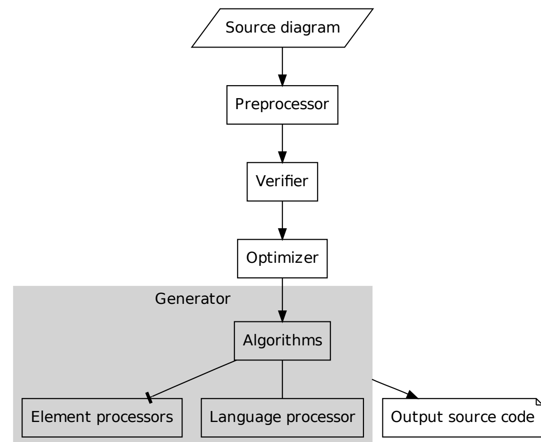

Fig. 4 Structure of code generator

A. 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 class diagram these classes are represented with boxes which contain three parts [1]:

- the upper part of holds the name of the class
- the middle part contains the attributes of the class
- the bottom part gives the methods or operations the class can take or undertake

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

[1]Note that state charts can "live" with class diagrams in very pleasant conjunction; state charts can be used for implementation of class member functions as well as the classes described by class diagrams can be instantiated in a source code generated from state charts. The both approaches are supported in CodeDesigner RAD tool.

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 B..

In addition to the classic classes the class diagram can contain also elements representing *class templates* and *enumerations* as shown in Figure 5.

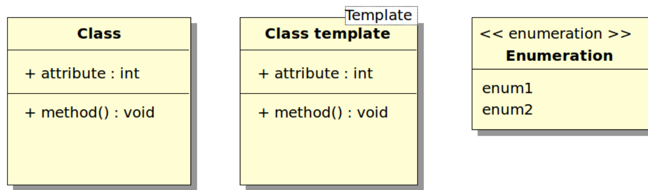

Fig. 5 Basic class diagram elements

The relations between class objects in the class diagram called *associations* and *aggregations* are defined in [1]. The most common are inheritance/interface association, uni-directional/bi-directional association, aggregation/composition aggregation, template binding and include association.

The following paragraphs show how class diagram elements are processed by the class diagram generator implemented in CodeDesigner RAD.

Class element Figure 6 shows basic class element with following members defined: **public** *attribute1*, **protected** *attribute2*, public *method1*, **private** *method2*, public *constructor* and public *destructor*.

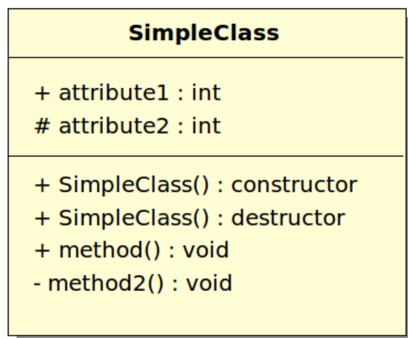

Fig. 6 Basic class element

Class inheritance Figure 7 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.

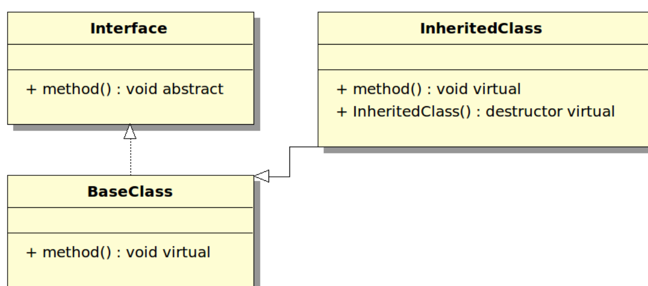

Fig. 7 Class inheritance

The following listing shows mapping of the class diagram in Figure 7 to C++ source code.

Listing 1 C++ class inheritance

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

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

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

void BaseClass::method() {
}

void InheritedClass::method() {
}

InheritedClass::~InheritedClass() {
}
```

Class inclusion Figure 8 shows inclusion of classes defined via *include association*. Note that *enumeration* elements can be included into parent classes in the same way. This operation ensures that included elements will be accessible under *namespace* defined by its parent element's name only.

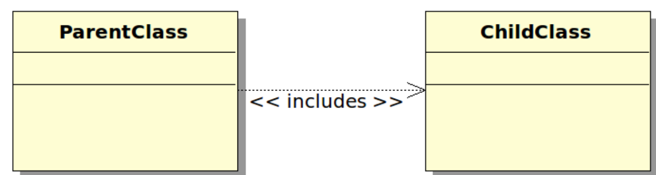

Fig. 8 Class inclusion

Listing 2 C++ class inclusion

```
class ParentClass {
  public:
    class ChildClass {
    };
};
```

Class template binding Figure 9 shows class *template* binded to *specialized* class. CodeDesigner RAD support class templates code generation for C++ language only because Python language doesn't use class templates.

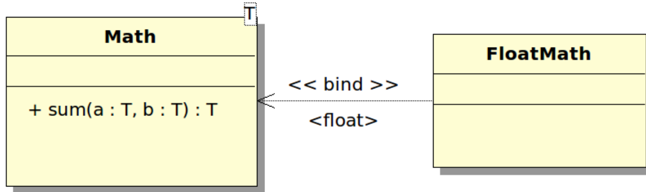

Fig. 9 Class template

Listing 3 C++ class template

```
template <typename T>
class Math {
  public:
    T sum( T a, T b );
};

class FloatMath : public Math<float> {
};

template <typename T>
T Math<T>::sum( T a, T b ) {
  return a + b;
}

template class Math<float>;
```

Enumeration element Enumeration element shown in Figure 10 is generated in different ways for C/C++ and Python languages. C/C++ language processor uses special *enum* keyword for the code generation but standard class is used instead for Python source code generation since the Python language doesn't provide reserved keyword for the enumerations.

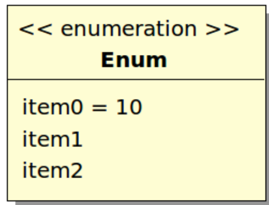

Fig. 10 Enumeration

Listing 4 C/C++ enumeration

```
enum Enum {
  item0 = 10,
  item1,
  item2
};
```

B. State Chart Code Generator

UML state chart is a significantly enhanced realization of the mathematical concept of a finite automaton [11] 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 [10] 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** [11] 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,

$$(S, s_0, \Sigma, \Lambda, T, G) \tag{1}$$

, 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 (Σ)
- a finite set called the output alphabet (Λ)
- 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

$$(G : S \times \Sigma \rightarrow \Lambda) \tag{2}$$

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

$$(G : S \rightarrow \Lambda) \tag{3}$$

maps each state to the output alphabet.

Graphical representation of Mealy and Moore state charts visualized by using UML state chart notation illustrates Figure 11. Note that Mealy state chart requires fewer states than the Moore state chart because all triggered actions are assigned directly to the transitions while the Moore state chart needs two extra states for writing the output by using their entry actions.

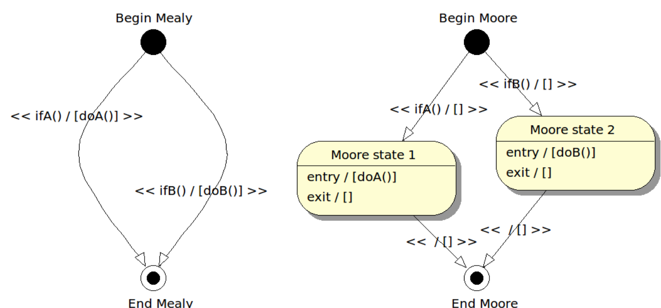

Fig. 11 Mealy vs Moore state charts

From the code generation point of view the symbols of both the input alphabet Σ and the output alphabet Λ 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 Σ while the user-defined actions (i.e. source code fragments or other methods) can be regarded as symbols of Λ.

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 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
- Go-To algorithm

All of them are optimized for production of easily readable and modifiable source code. For better understanding lets compare an output of the algorithms processing state chart shown in Figure 3.

The Listing 5 reveals an output produced by *Loop-case* algorithm. This algorithm is suitable for programming languages supporting *switch* command statement like C/C++ and JAVA. Unfortunately, it cannot be used in conjunction with Python language due to lack of *switch* command statement.

Listing 5 Loop-case algorithm output

```
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();
        }
        else {
          state = ID_FINAL;
        }
        break;
      }
      case ID_FINAL: {
        return ID_FINAL;
      }
    }
  }
}
```

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 5 where the *break* command is missing between states *ID_INITIAL*, *ID_HELLO* and *ID_WAIT_FOR_ENTER_KEY*.

Else-If algorithm's output based on the same input diagram is shown in Listing 6. The main difference is that is uses simple *if-else if-else* command sequence instead of *switch-case*. This approach solves the problem of missing *switch* command in some programming languages but precludes better program flow optimization like done in Loop-case algorithm because just one state can be entered per one main loop iteration.

Listing 6 Else-If algorithm output

```
STATE_T Hello_World(  ) {
  STATE_T state = ID_INITIAL;

  for( ;; ) {
    if( state==ID_INITIAL ) {
      sayHello();
      state = ID_HELLO;
    }
    else if( state==ID_HELLO ) {
      askForENTER();
      state = ID_WAIT_FOR_ENTER_KEY;
    }
    else if( state==ID_WAIT_FOR_ENTER_KEY ) {
      if( ! ( isEnter() ) ) {
        readKey();
      }
      else {
        state = ID_FINAL;
      }
    }
    else if( state==ID_FINAL ) {
      return ID_FINAL;
    }
  }
}
```

Both the Loop-case and Else-If algorithms produce quite large source code. This drawback solves the last mentioned algorithm called **Go-To** whose output is shown in Listing 7. As can be seen the Go-To algorithm produces the smallest extent of source code and the program flow is the most natural. In some cases it is able to generate source code similar to that one written manually by a human programmer. Unfortunately, it uses "evil" *goto* command statement which causes the source code to be un-structured little bit.

Listing 7 Go-To algorithm output

```
STATE_T Hello_World( )
{
  sayHello();
  askForENTER();

  ID_WAIT_FOR_ENTER_KEY_L:
  if( !( isEnter() ) )
  {
    readKey();
    goto ID_WAIT_FOR_ENTER_KEY_L;
  }

  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.

Simple State Charts Lets observe how simple state chart elements can be mapped to C/C++ source code by using different code generation algorithms.

Consider one initial pseudo state as a source of two transitions leading to two different final pseudo states as shown in Figure 12. One of the transitions is guarded by a conditional statement encapsulated inside a function returning boolean value in accordance to the evaluated logical expression. Both of the transitions have action code assigned.

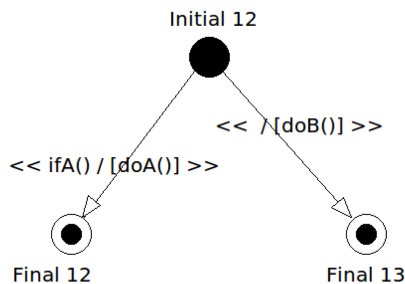

Fig. 12 Simple state chart

In this case the input alphabet as defined in (1) contains two symbols $\Sigma = \{ifA, \epsilon\}$ where ϵ is empty string and the output alphabet contains symbols $\Lambda = \{doA, doB\}$. If the state machine reads the symbol $\{ifA\}$ then it writes the output symbol $\{doA\}$ and it transits to the final state "Final 12". If the state machine reads $\{\epsilon\}$, i.e. there is no conditional statement guarding the transition then the state machine writes symbol $\{doB\}$ and transits to the final state "Final 13". Output of the Loop-case algorithm implementing the state chart show in Figure 12 if as follows:

Listing 8 Loop-case implementation of Figure 12

```
STATE_T Simple_State_Chart( )
{
  STATE_T state = ID_INITIAL_12;

  for( ;; ) {
    switch( state ) {
      case ID_INITIAL_12: {
        if( ifA() ) {
          doA();
          state = ID_FINAL_12;
        }
        else {
          doB();
          state = ID_FINAL_13;
        }
        break;
      }
      case ID_FINAL_12: {
        return ID_FINAL_12;
      }
      case ID_FINAL_13: {
        return ID_FINAL_13;
      }
    }
  }
}
```

Hierarchical State Charts Hierarchical state charts [10] allow definition of complex application behavior in a simple way with reduced number of used states than a standard state charts require. As mentioned above the UML state charts are based on the hierarchical state charts so the notation is nearly the same.

There are two main additions to the standard state charts defined in hierarchical ones and supported by CodeDesigner RAD:

- nested/composition states
- history pseudo states

At the first lets examine how **nested states** map into generated source code. Consider hierarchical state chart as shown in Figure 13.

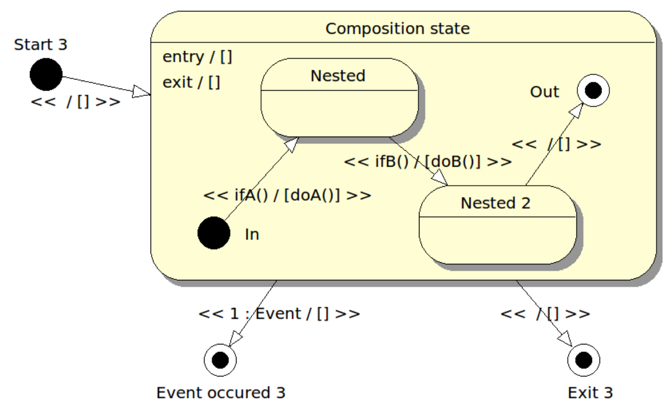

Fig. 13 Hierarchical state chart

Note the number *1* placed in front of the *Event* flag guarding a transition leading to "Event occurred 3" final state. This notation means that the transition trigger has the highest priority (a range <1, 255> can be used where 1 is the highest priority and 255 is the lowest - default - priority) so the conditional statement will be tested preferentially as can be seen in Listing 9.

During the state chart preprocessing the initial state "In" is merged with the parent state "Composition state" and two outcomming transitions pointing to final states "Event occurred 3" and "Exit 3" are re-connected to all remaining nested states so the transition guarded by *Event* flag is re-connected to both "Nested" and "Nested 2" states and the condition-less transition leading

to "Exit 3" final state is re-connected to nested final state called "Out".

The modifications performed on the parent "Composition state" ensures that *Event* flag is tested in all standard nested states. C/C++ source code generated from the diagram covering the discussed functionality is as follows:

Listing 9 Loop-case implementation of Figure 13

```
STATE_T State_Chart_2( )
{
  STATE_T state = ID_START_3;

  for( ;; ) {
    switch( state ) {
      case ID_START_3: {
        state = ID_COMPOSITION_STATE;
      }
      case ID_COMPOSITION_STATE: {
        if( ifA() ) {
          doA();
          state = ID_NESTED;
        }
        break;
      }
      case ID_NESTED: {
        if( Event ) {
          state = ID_EVENT_OCCURED_3;
        }
        else if( ifB() ) {
          doB();
          state = ID_NESTED_2;
        }
        break;
      }
      case ID_NESTED_2: {
        if( Event ) {
          state = ID_EVENT_OCCURED_3;
        }
        else {
          state = ID_OUT;
        }
        break;
      }
      case ID_OUT: {
        state = ID_EXIT_3;
      }
      case ID_EXIT_3: {
        return ID_EXIT_3;
      }
      case ID_EVENT_OCCURED_3: {
        return ID_EVENT_OCCURED_3;
      }
    }
  }
}
```

The **history pseudo state** can be used in conjunction with nested states as illustrated in Figure 14.

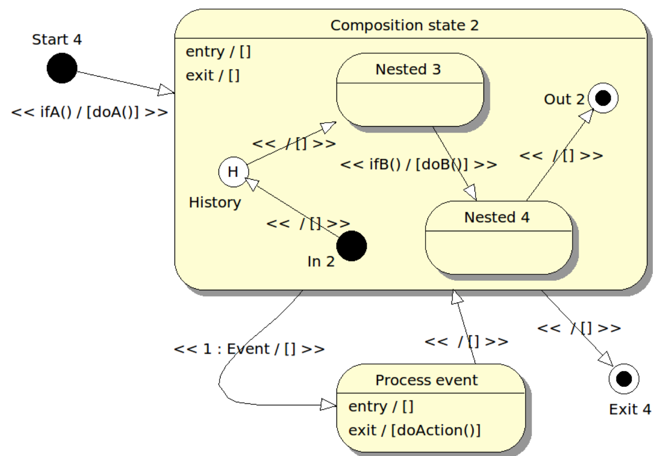

Fig. 14 Hierarchical state chart with history

History state behaves like a composition state's local memory where information about currently processed state is kept. The information is used later for restoration of run point at which the composition state was leaved (in a case the *Event* has occurred) after the state is entered again like shown in Figure 14. A source code implementing the state chart in C/C++ by using Loop-case algorithm can be following:

Listing 10 Loop-case implementation of Figure 14

```
STATE_T State_Chart_3( )
{
  STATE_T state = ID_START_4;
  STATE_T history = ID_NESTED_3;

  for( ;; ) {
    switch( state ) {
      case ID_START_4: {
        if( ifA() ) {
          doA();
          state = ID_COMPOSITION_STATE_2;
        }
        break;
      }
      case ID_COMPOSITION_STATE_2: {
        state = ID_HISTORY;
      }
      case ID_HISTORY:
      /* call entry actions of possible target
         states */
        switch( history )
        {
        }
        state = history;
        break;

      case ID_NESTED_3: {
        if( Event ) {
          history = ID_NESTED_3;
          state = ID_PROCESS_EVENT;
        }
        else if( ifB() ) {
          doB();
          state = ID_NESTED_4;
        }
        break;
      }
```

```
        case ID_NESTED_4: {
          if( Event ) {
            history = ID_NESTED_4;
            state = ID_PROCESS_EVENT;
          }
          else {
            state = ID_OUT_2;
          }
          break;
        }
        case ID_OUT_2: {
          history = ID_OUT_2;
          state = ID_EXIT_4;
        }
        case ID_EXIT_4: {
          return ID_EXIT_4;
        }
        case ID_PROCESS_EVENT: {
          doAction();
          state = ID_COMPOSITION_STATE_2;
          break;
        }
      }
    }
}
```

IV. Reverse Source Code Engineering

In addition to discussed source code generation capabilities modern CASE tools like Visual Paradigm [8], Enterprise Architect [2] or CodeDesigner RAD offers also source code reverse engineering functionality. As stated in [6] 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" [9].

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 to reverse engineering capabilities provided by CodeDesigner RAD tool.

A. *Source Code Analysis*

Static source code analysis can cover both source code structure and application logic. The code structure can be visualized by using UML class diagrams and the application logic by using UML/standard state charts as shown in Chapter II..

There exist plenty of software tools able to perform those analysis. For instance, Exuberant CTAGS [7] command line tool can be used for generation an index (or tag) file of language objects found in source files. A tag signifies a language object for which an index entry is available (or, alternatively, the index entry created for that object).Software tools able to analyze behavioral aspects of examined source code are for instance Ablegold Computer's Easystructure tool [13] and C Algorithm Viewer tool [12]. Both of them can visualize the program flow as a tree view or a flowchart.

In the current version, the CodeDesigner RAD v 1.5.4 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 CASE tools which is crucial for complete round-trip code engineering implementation as discussed in Chapter B..

CodeDesigner RAD reverse code engineering plug-in provides C/C++ and Python classes import with:

- support for multiple inheritance
- support for include associations
- support for uni-directional class associations based on defined class members
- import of stand-alone and included enumerations
- import of stand-alone functions and variables
- import of complete function bodies

The reverse engineering process can be stared from the panel shown in Figure 15:

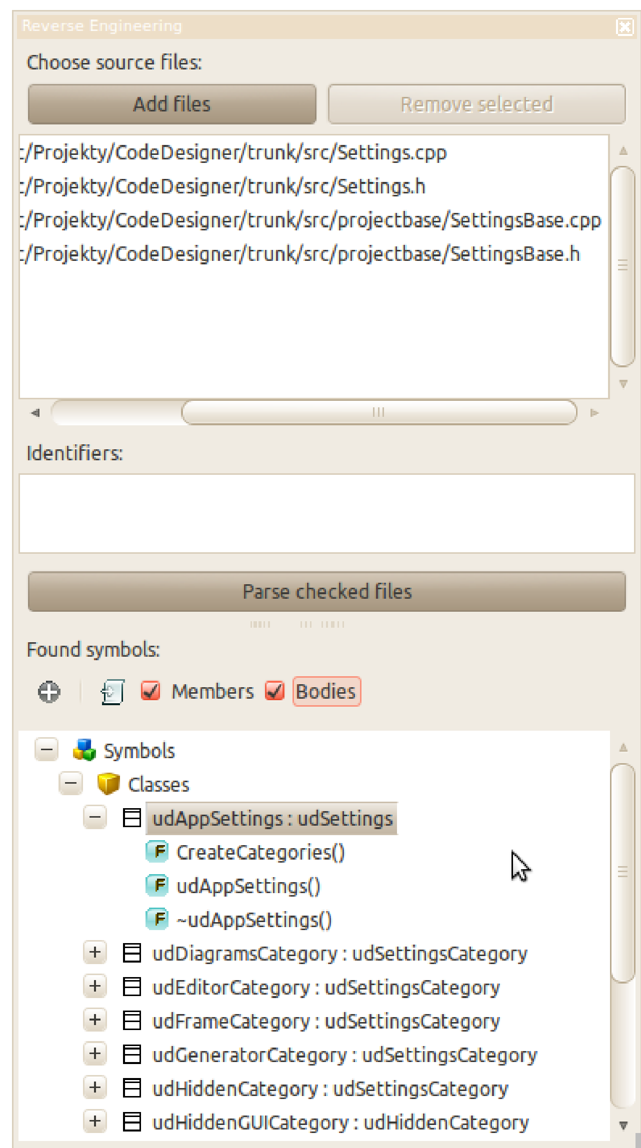

Fig. 15 CodeDesigner RAD Reverse Engineering panel

The upper part of the panel contains list box where analyzed source files are listed. The selected files are processed by the

CTAGS utility after clicking the *Parse checked files* button placed under the list box. All found symbols like **variables**, **functions**, **classes** and **enumerations** are displayed in the tree control placed under the parse button. User must select which found symbols should be imported into newly created class diagram and also set whether class members and function bodies will be imported as well. The new class diagram with selected symbols will be created after clicking the button placed next the check boxes displayed above the symbol tree control.

The following source code listing and class diagram show an ability to import classes where multiple inheritance is used. It also illustrates how referenced classes are associated in the class diagram.

Listing 11 Class sample 1

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

class Being {
};

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

  const char* GetSpecie() const {;}

protected:
  const char *m_specie;
};

class Butterfly : public Animal, public Being {
public:
  Butterfly() {;}

protected:
  Data *m_pData;
};
```

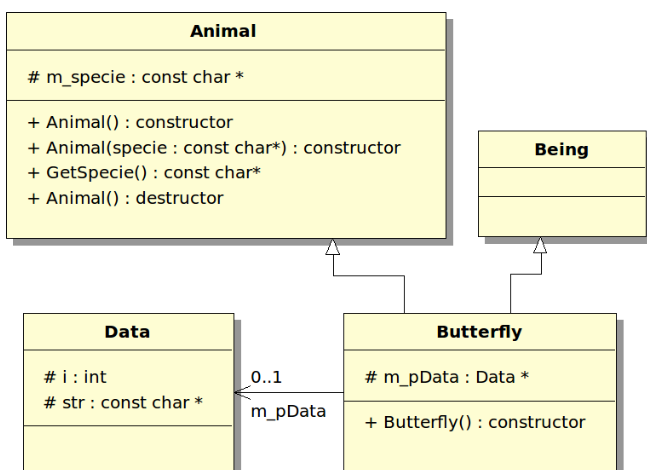

Fig. 16 Class sample 1

As can be seen in Figure 16 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.

The next sample shows how embedded classes are recognized and how C/C++ enumeration is imported into a class diagram.

Listing 12 Class sample 2

```
class OuterClass {
public:
  class InnerClass {
  public:
    enum InnerEnum {
      itemONE = 0,
      itemTWO
    };
  };
};
```

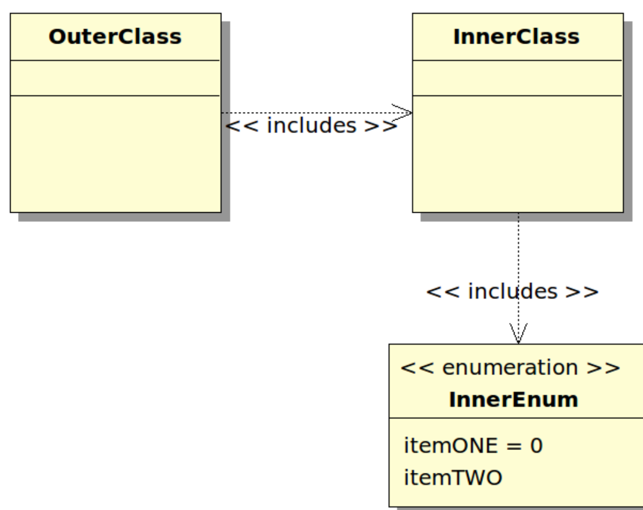

Fig. 17 Class sample 2

In the Figure 17 two classes *OuterClass* and *InnerClass* are shown together with the include association connecting them. There is also one enumeration element called *InnerEnum* defined as a part of *InnerClass* so the same association element is used with the enumeration as well.

B. 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 are source code, models, configuration files, and other documents. The need for round-trip engineering arises when the same information is present in multiple artifacts and therefore an inconsistency may occur if not all artifacts are consistently updated to reflect a given change. For example, some piece of information was added to/changed in only one artifact and, as a result, it became missing in/ inconsistent with the other artifacts.

Round-trip engineering process consist of several repetitive tasks as illustrated in Figure 18.

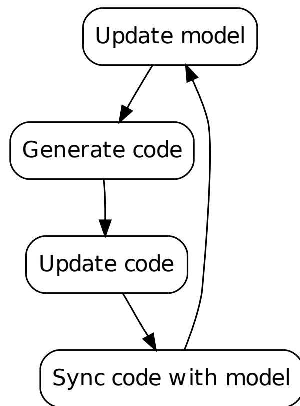

Fig. 18 Round-trip engineering

Now lets look how the round-trip 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 13 Round-trip step 1

```
#include <stdio.h>

int main(int argc, char **argv)
{
  return 0;
}
```

2. Create empty CodeDesigner project and save it in the same location like the C++ source code. The CodeDesigner RAD project settings should be as shown in Figure 19

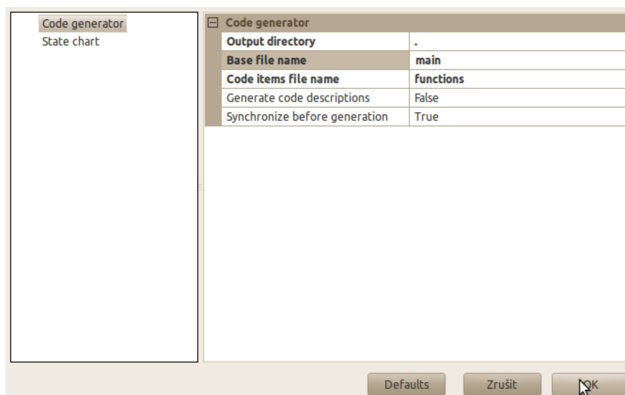

Fig. 19 CodeDesigner RAD project settings

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 function body from within the CodeDesigner RAD. The class diagram should look like shown in Figure 20.

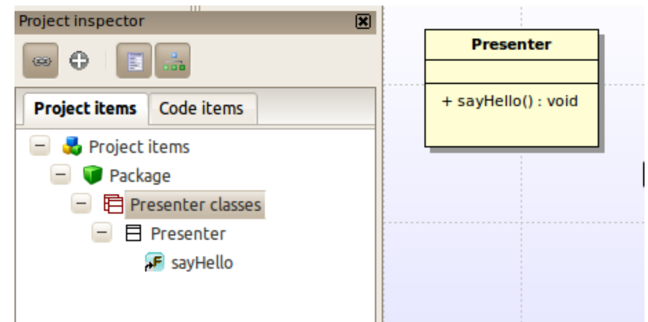

Fig. 20 CodeDesigner RAD project settings

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 bellow.

Listing 14 main.h

```
/* ['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 15 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)
{
  return 0;
}

/* ['Presenter classes' begin (DON'T REMOVE
   THIS LINE!)] */
```

```
/* public function members of 'Presenter'
    class */
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 invoked from the *Presenter* class instance created in the *main()* function as shown in Listing 16:

Listing 16 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 and run the application. The output should look like illustrated in Figure 21.

Fig. 21 Sample 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 (this functionality is optional and must be enabled in the project settings). A dialog window is displayed during the synchronization process; it allows user to select which modified code should be imported into CodeDesigner RAD project as shown in Figure 22.

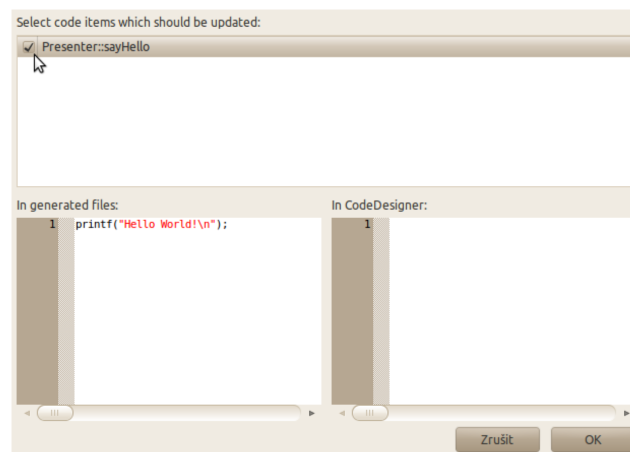

Fig. 22 Synchronization dialog

At the moment, the CodeDesigner RAD project is fully synchronized with the application source code and user can proceed with the round-trip engineering like described in step 3. 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.

V. 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. Moreover, the generated code can be optimized by using several algorithms as discussed in the paper. 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

[1] UML 2.2 infrastructure. http://www.omg.org/spec/UML/2.2/ Infrastructure/PDF/, 2011.

[2] Enterprise Architect. UML design tools and UML CASE tools for software development. http://www.sparxsystems.com/products/ ea/index.html, January 2012.

[3] Michal Bližňák. CodeDesigner RAD homepage. http://codedesigner.org/, 2011.

[4] Michal Bližňák, Tomáš Dulík, and Vladimír Vašek. A persistent Cross-Platform class objects container for c++ and wxWidgets. *WSEAS TRANSACTIONS on COMPUTERS Volume 8, 2009*, 8(1), January 2009.

[5] Michal Bližňák, Tomáš Dulík, and Vladimír Vašek. wxShapeFramework: an easy way for diagrams manipulation in c++ applications. *WSEAS TRANSACTIONS on COMPUTERS Volume 9, 2010*, 9(1), January 2010.

[6] E. J Chikofsky and J. H. Cross. Reverse engineering and design recovery: a taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[7] Exuberant Ctags. Exuberant ctags. http://ctags.sourceforge.net/, January 2012.

[8] Visual Paradigm for UML. Round-trip engineering. http://www.visual-paradigm.com/product/vpuml/provides/ roundtripcodeengine.jsp, January 2012.

[9] P. A. V. Hall. *Software Reuse and Reverse Engineering in Practice*. Chapman & Hall, 1st edition, March 1992.

[10] David Harel. Statecharts: A visual formalism for complex sysytems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[11] John Hopcroft and Jeffrey Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company, 1979.

[12] Rhee J.D. C algorithm viewer. http://sourceforge.net/projects/algoview/, January 2012.

[13] Ablegold Computers Ltd. Easystructure. http://www.ablegoldcomputers.com/index.html, January 2012.

[14] Julian Smart, Kevin Hock, and Stefan Csomor. *Cross-Platform GUI Programming with wxWidgets*. Prentice Hall, August 2005.