INTERNATIONAL JOURNAL OF SYSTEMS APPLICATIONS, ENGINEERING & DEVELOPMENT
Issue 5, Volume 7, 2013

# Optimized Production-Ready Source Code Generation Based on UML

Michal Bližňák, Tomáš Dulík, Roman Jašek, and Pavel Vařacha

***Abstracts***—Automated source code generation is often an integral part of modern CASE tools. Unfortunately, the generated code usually 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 of complete C/C++ or Python applications from formal visual description based on UML diagrams. It shows how source state charts are preprocessed as well principles used for generation of optimized source code from the preprocessed diagrams.

***Keywords***—RAD, CASE, UML, source, code, generation, C/C++, production-ready, cross-platform, optimization, preprocessing, CodeDesigner

## I. Introduction

NOWADAYS, there exist many software development tools able to generate source code of basic application skeleton from its formal description (typically described by using UML diagrams). Unfortunately, in many cases these tools lack an ability to generate complete, production-ready source code defining both the application structure and logic suitable for building without need of any modifications done by a developer. This paper shows principles and algorithms used in cross-platform development tool called CodeDesigner RAD [2] aimed for production-ready source code generation which allows users to generate complete applications from their formal description.

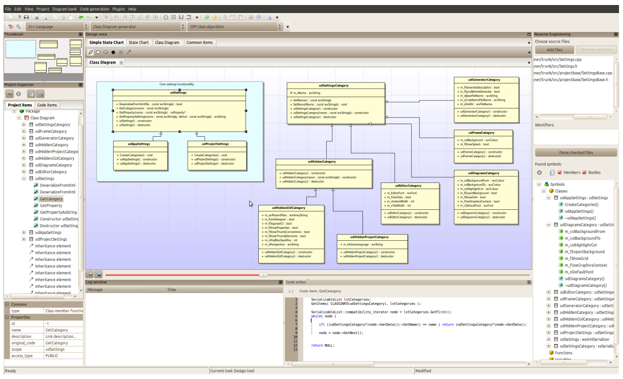

Fig. 1: CodeDesigner RAD

Michal Bližňák is with the Tomas Bata University, Faculty of Applied Informatics, Department of Informatics and Artificial Intelligence, Nad Stranemi 4511, 76005 Zlin, Czech Republic (corresponding author to provide phone: 00420-576035187; e-mail: bliznak@fai.utb.cz).

Tomáš Dulík, Roman Jašek and Pavel Vařacha are with the Tomas Bata University, Faculty of Applied Informatics, Department of Informatics and Artificial Intelligence, Nad Stranemi 4511, 76005 Zlin, Czech Republic ( e-mails: dulik@fai.utb.cz, jasek@fai.utb.cz and varacha@fai.utb.cz).

CodeDesigner RAD application show in Fig. 1 has been developed by using well known cross-platform programming toolkit called wxWidgets [8] together with its add-ons wxXmlSerializer [3] and wxShapeFramework [4]. 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.

The algorithms and principles shown in this article mostly don't depend on specific programming language (except a few specialized optimizations). This is important because formal application description is language-independent in its nature. It will be shown that presented principles can be used for generation of complete applications written in C/C++, Python or another object-oriented programming language.

## II. Code Generation in Detail

The code generation process consists of four steps as shown in Fig. 2. First of all a source diagram is *preprocessed* so its structure will change in order to be more suitable for further processing by the code generator. Preprocessed diagram must be *verified* to find possible inconsistencies in the diagram's topology. If the verification fails the code generation process is aborted. After that, a set of *optimization* procedures leading to various simplifications of the diagram's structure 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 *code generator*.

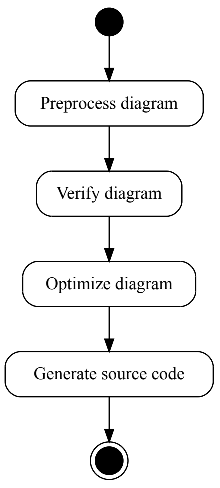

Fig. 2: Code generation process

The code 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). Generally, there are four basic types of code fragments:

- functions declarations and definitions,
- variables declarations and assignments,
- conditional statements,
- user-defined source code of methods/functions and the conditional statements.

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. Several language processors can be used at the same time so we can get set of source files in different programming languages during one code generation process. The complete structure of source code generator implemented in CodeDesigner RAD is shown in Fig 3.

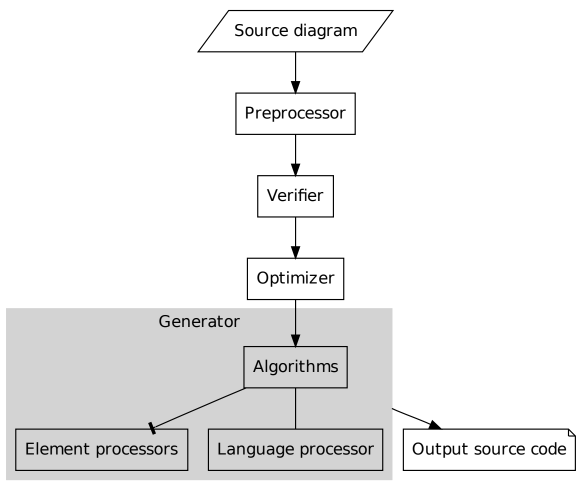

Fig. 3: Structure of code generator

## A. Diagram preprocessing

Preprocessing of source diagrams implemented in CodeDesigner RAD is used for deconvolution of hierarchical state charts [5] into classic Mealy state machines [1] further processed by state chart code generator.

The deconvolution process consists of four dependent steps:

1. **conversion of entry/exit state actions into transition actions** – the algorithm checks all states in the diagram whether they include entry/exit actions and assign those actions to all non-state-loop (i.e. transitions starting and ending in different states) incoming/outcoming transitions as shown in Fig. 4.

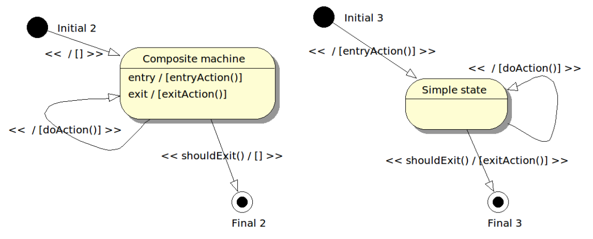

Fig. 4: Entry/exit actions conversion

2. **re-connection of state outputs** – the algorithm creates copies of conditional transitions starting in parent hierarchical state and connects them to all (next level) child states. The condition-less transitions starting in the parent hierarchical state will be connected to embedded final state. The child states must be processed from the top level to the bottom level so BFS algorithm [7] must be used for retrieving of the child states. The simple illustration of this modification is shown in Fig. 5.

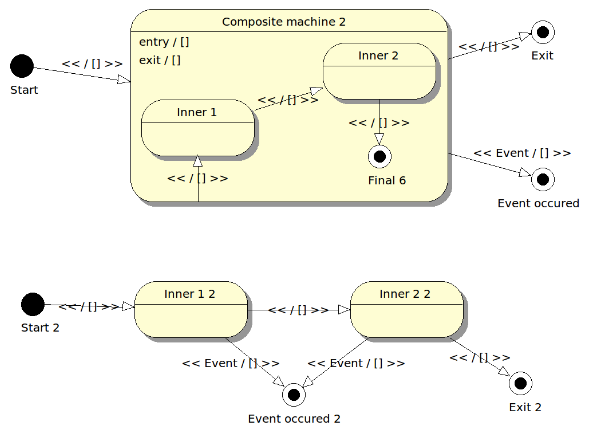

Fig. 5: Re-connection of output transitions

3. **re-connection of state inputs** – the algorithm re-connects all transitions starting in embedded initial states (i.e. initial states placed inside a hierarchical state) to their parent hierarchical state.

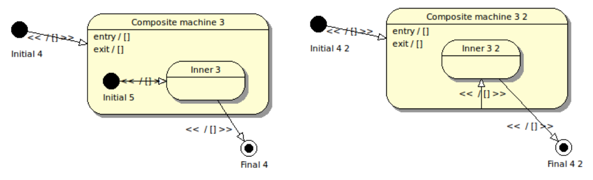

Fig. 6: Re-connection of input transitions

4. **state actions sorting** – state actions assigned to transitions must by sorted in such way that EXIT actions must be at the top of the list followed by ENTRY actions.

## B. Verification

Verifications done after the source diagram preprocessing ensure that there are no inconsistencies and other issues in the diagram structure so it can be successfully processed by next parts of code generator. The verification process of state charts consists of the following independent tests runnable in any order:

- **check initial states** – the algorithm tries to find whether there exist exactly one initial state in the diagram/parent hierarchical state
- **check history states** – check history states so:
  - history state must have a parent hierarchical state
  - there can be only one incoming transition in the history state which must be conditionless and must start in its parent or initial state
  - there can be only one outcoming transition in the history state which must be conditionless
- **check final states** – the algorithm tries to find whether there exist at least one final state in the diagram
- **check unconnected states** – the algorithm checks whether the diagram is fully connected
- **check transitions** – check transition paths between states so:
  - there are no transitions with duplicated guard condition per a state
  - there are no transitions with duplicated priority level per a state
  - there is just one conditionless transitions per a state
- **check input actions** – check whether a state chart with defined input actions has activated code generation algorithm supporting the input actions[1]

### *C. Optimization*

Optimization algorithms provided by state chart generator implemented in CodeDesigner RAD are aimed to reduction of used states and to simplifications of the diagram topology. Moreover, they allow code generator to produce source code easily readable by humans.

The diagram optimization process consists of three dependent steps which can be performed in several iterations until further optimization is required and a defined maximum iteration count is not reached as shown in Algorithm 1. The "hardcoded" maximum iteration count is used due to a possibility of main optimization loop to converge into infinite loop under certain conditions.

The optimization tasks are following:

1. **merge of direct diagram branches** – the algorithm tries to find direct condition-less branches inside the optimized diagram composed of several transitions and states and merge them into one condition-less transition with an action combined from all actions used in the branch as shown in Fig. 7. The optimization reduces overall number of the diagram states.

[1]The *input action* is an action/function invoked at every iteration of a main loop implementing the state chart behavior so it can be used for reading of *input symbols* as discussed in Chapter III.B. Each state chart can has got one input action defined.

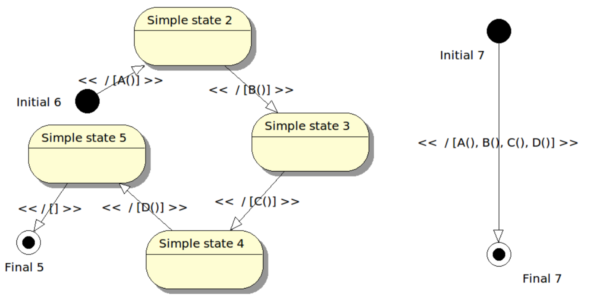

Fig. 7: Optimize direct branches

2. **merge of parallel transitions** – the algorithm tries to merge several parallel transitions with identical starting and ending states guarded by identical conditions into one transition with combined conditional statement as shown in Fig. 8. The optimization reduces number of transitions in the diagram.

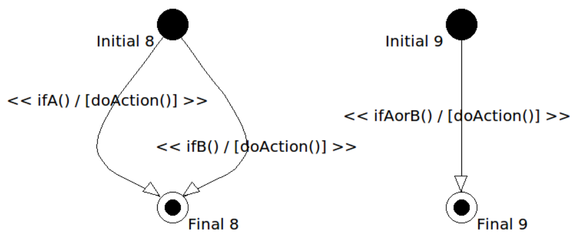
Initial 8
Initial 9
<< ifA() / [doAction()] >>
<< ifAorB() / [doAction()] >>
<< ifB() / [doAction()] >>
Final 8
Final 9

Fig. 8: Optimize parallel transitions

3. **inversion of conditions** – the algorithm optimizes subdiagrams composed of two transitions with identical starting state where the first transition contains a condition and has no actions while the second one contains action(s) and has no condition assigned so just one transition with inverted condition and defined action(s) is created as shown in Fig. 9. The algorithm optimizes program flow and makes the generated source code "prettier".

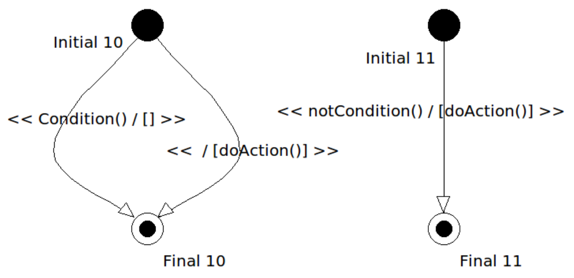

Fig. 9: Invert conditions

The main optimization loop is shown in Algorithm 1. Note that flag *again* used for determination whether further optimization is required is built as a logical sum of return values provided by the subsequent optimization tasks returning **true** if the task performed any change on the diagram topology.

## III. Generators, Algorithms and Element Processors

In general, any automated source code generator can produce just generic source code covering basic application structure and

**Algorithm 1** Optimize state chart

```
$i \leftarrow 1$
**repeat**
    $again \leftarrow$ **false**
    $again| = mergeDirectBranches()$
    $again| = mergeParallelTransitions()$
    $again| = invertConditions()$
    $i \leftarrow i + 1$
**until** $i \leq MAXITER$ **and** $again$ is **true**
```

logic, in other words the application skeleton. There are always minimal code fragments called *code snippets* that must be written by the programmer manually. These code snippets often implement platform/language specific operations which cannot be generated automatically like input/output operations, arithmetic calculations, etc. Therefore, CASE tools aimed for production-ready source code generation must allow users to manually define the code snippets and use them in the code generation process. This approach allows code generation of production-ready source code which can be built into full-featured executable.

Since every diagram type describes the application internals in a completely different way then specific code generators must be available for each diagram. Of course, there can exist more than one code generator suitable for one diagram type and the user can choose the one which fills all his needs.

Another aspect of source code generation process is usage of various code generation algorithms suitable for specific diagram types, output programming language and coding style. For example, CodeDesigner RAD provides three different code generation algorithms for state charts which are filtered by output programming language: Loop-Case, Else-If for C/C++ and Python and Go-To for C/C++ only, since Python programming language doesn't support direct jumps. The output of these algorithms differs not only in used command statements but also in the extent of produced source code. The differences between the algorithms is discussed in Chapter III.B.

Code generation algorithms implemented in CodeDesigner RAD aggregates set of so called *element processors* responsible for production of source code fragments based on processed diagram element. Generally, the element processors can be shared between several code generation algorithms or several diagram elements if they produce the same source code.

The next chapters deal with specific aspects of code generators/algorithms available in CodeDesigner RAD.

## 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 or 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 system 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 III.B.

In addition to the classic classes the class diagram can contain also elements representing *class templates* and *enumerations* as shown in Fig. 10.

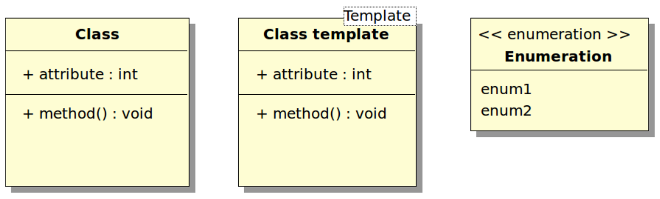

Fig. 10: Basic class diagram elements

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

- **inheritance/interface association** – defines inheritance relation between classes and interfaces
- **uni-directional/bi-directional association** – represents the static relationship shared among the objects of two classes
- **aggregation/composition aggregation** – association that represents a part-whole or part-of relationship
- **template binding** – used for specialization definition of template classes
- **include association** – used for inclusion of a class into another (parent) class which behaves as a *namespace*

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

**Class element** Fig. 11 shows basic class element with following members defined: **public** *attribute1*, **protected** *attribute2*, public *method1*, **private** *method2*, public *constructor* and public *destructor*.

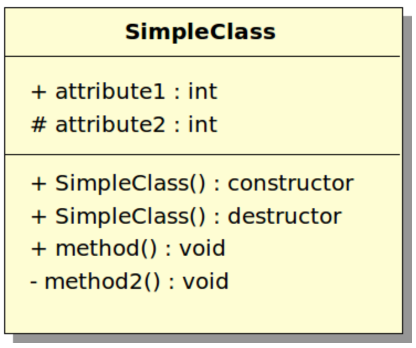

Fig. 11: Basic class element

Listing 1: C++ class declaration/definition

```
class SimpleClass {
  public:
    int attribute1;

    SimpleClass();
    ~SimpleClass();
    void method();

  protected:
    int attribute2;

  private:
    void method2();

};

SimpleClass::SimpleClass() {
}

SimpleClass::~SimpleClass() {
}

void SimpleClass::method() {
}

void SimpleClass::method2() {
}
```

Listing 2: Python class definition

```
class SimpleClass:
  # public data members:
  attribute1

  # protected data members:
  __attribute2

  # public function members:
  def __init__( self ):
    """ Constructor """
    pass

  def __del__( self ):
    """ Destructor """
    pass

  def method( self ):
    pass

  # private function members:
  def __method2( self ):
    pass
```

**Class inheritance** Fig. 12 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.

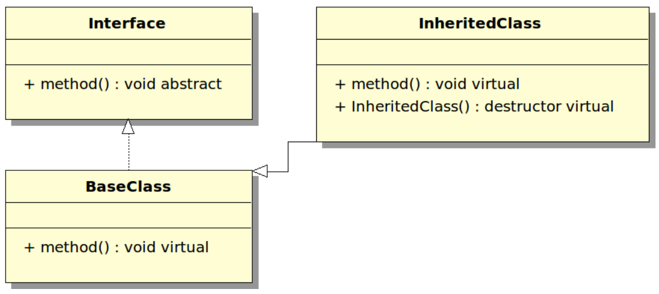
Interface
+ method() : void abstract
InheritedClass
+ method() : void virtual
+ InheritedClass() : destructor virtual
BaseClass
+ method() : void virtual

Fig. 12: Class inheritance

Listing 3: 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() {
}
```

Listing 4: Python class inheritance

```
class Interface:
  # public function members:
  def method( self ):
    pass

class BaseClass( Interface ):
  # public function members:
  def method( self ):
    pass

class InheritedClass( BaseClass ):
  # public function members:
  def method( self ):
    pass

  def __del__( self ):
    pass
```

**Class inclusion** Fig. 13 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 it parent element name.

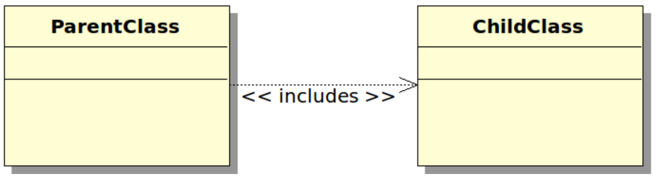

Fig. 13: Class inclusion

Listing 5: C++ class inclusion

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

Listing 6: Python class inclusion

```
class ParentClass:
  class ChildClass:
    pass

  pass
```

**Class template binding** Fig. 14 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.

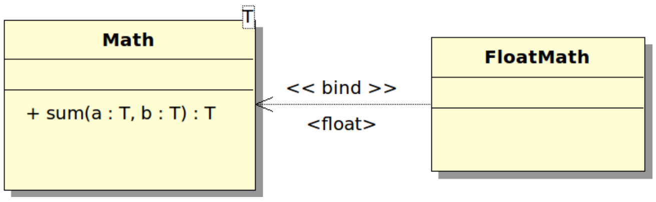

Fig. 14: Class template

Listing 7: 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 shown in Fig. 15 is generated in different ways for C/C++ and Python languages. C/C++ language processor uses standard *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.

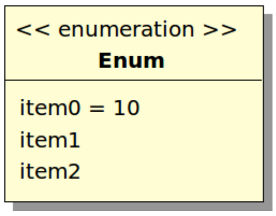

Fig. 15: Enumeration

Listing 8: C/C++ enumeration

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

Listing 9: Python enumeration

```
class Enum:
  item0 = 10,
  item1 = 11,
  item2 = 12
```

### B. *State Chart Code Generator*

UML state chart is a significantly enhanced realization of the mathematical concept of a finite automaton [6] 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 [5] 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** [6] defined in (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 ($\Sigma$)
- 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

$$(G : S \times \Sigma \rightarrow \Lambda) \quad (2)$$

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

$$(G : S \rightarrow \Lambda) \quad (3)$$

maps each state to the output alphabet.

Graphical representation of Mealy and Moore state charts visualized by using UML state chart notation illustrates Fig. 16. 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.

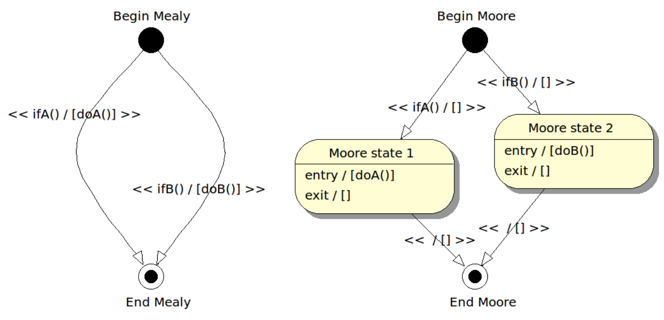

Fig. 16: Mealy vs Moore state charts

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 mentioned above. 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 other methods) can be regarded as symbols of $\Lambda$.

A projection of state chart describing an application logic into a source code is influenced 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 Fig. 17.

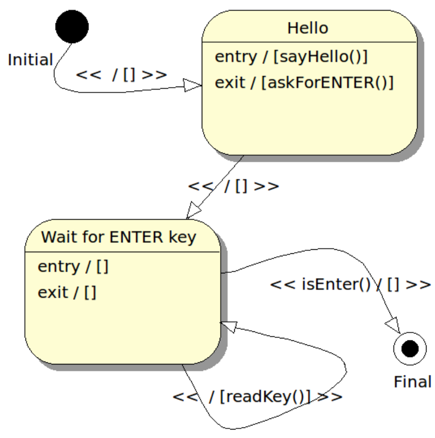

Fig. 17: State Chart

The Listing 10 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 missing *switch* command statement.

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

Listing 10: 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;
      }
    }
  }
}
```

**Else-If** algorithm's output based on the same input diagram is shown in Listing 11. 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 prevents better program flow optimization like done in Loop-case algorithm because just one state can be entered per one main state chart loop iteration.

Listing 11: 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 12. 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 the 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 12: 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 specific diagram.

## C. Simple State Charts

Lets observe how simple state chart elements can be mapped to C/C++ source code by using different algorithms. Both Loop-case and Go-To algorithms will be discussed in the following examples for better understanding of their main differences.

Consider one initial pseudo state as a source of two transitions leading to two different final pseudo states as shown in Fig. 18. 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.

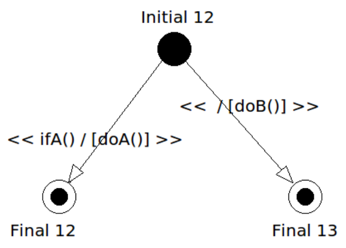

Fig. 18: Simple state chart

In this case the input alphabet as defined in (1) contains two symbols $\Sigma = \{ifA, \epsilon\}$ where $\epsilon$ 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 Fig. 18 if as follows:

Listing 13: Loop-case implementation of Fig. 18

```
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;
        }
      }
    }
}
```

Output of the Go-To algorithm implementing the same state chart is as follows:

Listing 14: Go-To implementation of Fig. 18

```
STATE_T Simple_State_Chart( )
{
  if( ifA() ) {
    doA();
    goto ID_FINAL_12_L;
  }
  else {
    doB();
  }

  return ID_FINAL_13;

  ID_FINAL_12_L:
  return ID_FINAL_12;
}
```

### D. *Hierarchical State Charts*

Hierarchical state charts [5] allow definition of complex application behavior in a simple way with reduced number of used states than would 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** are mapped into generated source code. Consider hierarchical state chart as shown in Fig. 19.

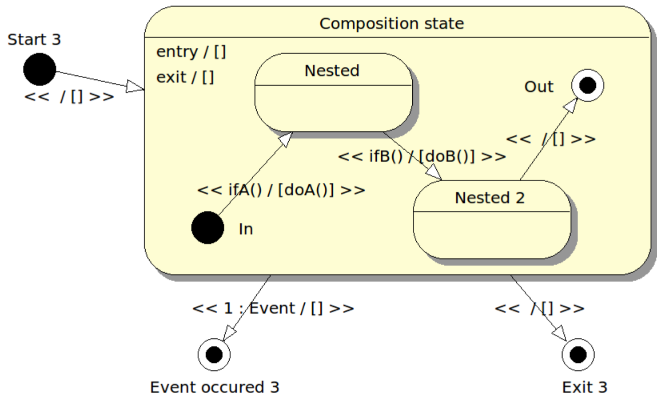

Fig. 19: Hierarchical state chart

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 15.

During the state chart preprocessing discussed in Chapter A. 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 its functionality is as follows:

Listing 15: Loop-case implementation of Fig. 19

```
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;
```

```
            }
        }
    }
}
```

Go-To algorithm's output implementing the same state chart is following:

Listing 16: Go-To implementation of Fig. 19

```
STATE_T State_Chart_2( )
{
  ID_COMPOSITION_STATE_L:
  if( ifA() ) {
    doA();
    goto ID_NESTED_L;
  }
  goto ID_COMPOSITION_STATE_L;

  ID_NESTED_L:
  if( Event ) {
    goto ID_EVENT_OCCURED_3_L;
  }
  else if( ifB() ) {
    doB();
    goto ID_NESTED_2_L;
  }
  goto ID_NESTED_L;

  ID_NESTED_2_L:
  if( Event ) {
    goto ID_EVENT_OCCURED_3_L;
  }

  return ID_EXIT_3;

  ID_EVENT_OCCURED_3_L:
  return ID_EVENT_OCCURED_3;
}
```

The **history pseudo state** can be used in conjunction with nested states as illustrated in Fig. 20.

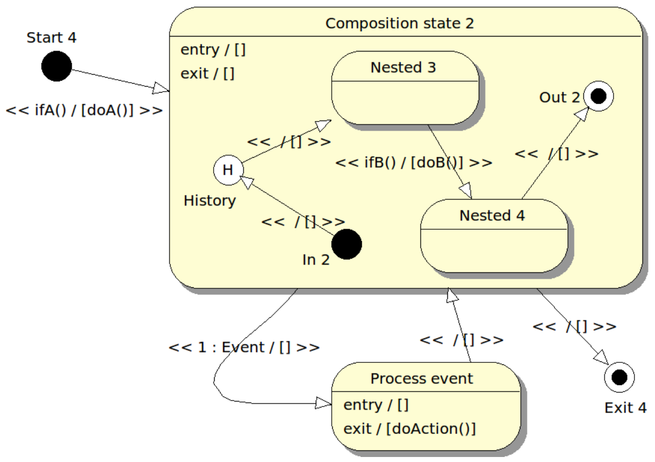

Fig. 20: Hierarchical state chart with history

History state behaves like a composition state 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 Fig. 20. A source code implementing the state chart in C/C++ by using Loop-case algorithm can be following:

Listing 17: Loop-case implementation of Fig. 20

```
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;
      }
      }
   }
}
```

Go-To algorithm's output implementing the same state chart is following:

Listing 18: Go-To implementation of Fig. 20

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

  ID_START_4_L:
  if( ifA() ) {
    doA();
    goto ID_COMPOSITION_STATE_2_L;
  }
  goto ID_START_4_L;

  ID_COMPOSITION_STATE_2_L:

  ID_HISTORY_L:
  if( history == ID_NESTED_3 ) {
    goto ID_NESTED_3_L;
  }
  else if( history == ID_NESTED_4 ) {
    goto ID_NESTED_4_L;
  }

  ID_NESTED_3_L:
  if( Event ) {
    history = ID_NESTED_3;
    goto ID_PROCESS_EVENT_L;
  }
  else if( ifB() ) {
    doB();
    goto ID_NESTED_4_L;
  }
  goto ID_NESTED_3_L;

  ID_NESTED_4_L:
  if( Event ) {
    history = ID_NESTED_4;
    goto ID_PROCESS_EVENT_L;
  }

  return ID_EXIT_4;

  ID_PROCESS_EVENT_L:
  doAction();
  goto ID_COMPOSITION_STATE_2_L;
}
```

## IV. Source Code Optimizations

Generated source code can be optimized at several different levels. Optimizations performed on processed diagrams discussed in Chapter II.C. lead to production of optimized source codes with reduced extent or better readability due to better source diagram's topology. Also some kind of low-level optimizations can be performed on the code: omitting of useless command statements as discussed in Chapter III.B. leads to better program flow, omitting of unused code labels in Go-To algorithm suppresses C compiler warnings, etc. Now, lets observe how specific optimizing and preprocessing algorithms influences properties of output source code.

At the first, **inversion of conditional statements** discussed in Chapter II.C. will be examined. Lets consider a simple state chart like in Fig. 21.

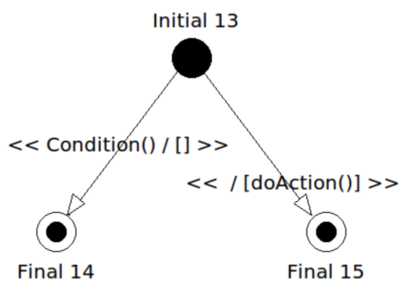

Fig. 21: Condition inversion

Source code generated by Go-To algorithm **without** optimized inversion of conditional statements can be seen in Listing 19.

Listing 19: Go-To algorithm without conditions inversion

```
STATE_T Inversion( )
{
  if( Condition() ) {
    goto ID_FINAL_14_L;
  }
  else {
    doAction();
  }

  return ID_FINAL_15;

  ID_FINAL_14_L:
  return ID_FINAL_14;
}
```

In this case the compound command enclosed by curly braces after positive condition test contains just one jump to relevant final state while the compound command invoked after the *else* keyword contains an action which will be called in the case the tested condition is not met. Lets compare the source with content of Listing 20 where the same input state chart is processed by using Go-To algorithm **with** optimized inversion on conditions.

Listing 20: Go-To algorithm with conditions inversion

```
STATE_T Inversion( )
{
  if( !( Condition() ) ) {
    doAction();
    goto ID_FINAL_15_L;
  }

  return ID_FINAL_14;

  ID_FINAL_15_L:
  return ID_FINAL_15;
}
```

As can be seen from the listing the inversion of generated conditions allows the algorithm to merge both contents of compound commands shown in Listing 19 so the whole *else* section of *if* clause can be omitted.

Another source code optimization which can be illustrated on the Fig. 21 is **suppression of unused *goto* labels**. Both Listings 19 and 20 show source code where the optimization is used. Now lets compare the Listing 20 with following one where the suppression of unused *goto* labels is not active.

Listing 21: Go-To algorithm without suppression of unused labels

```
STATE_T Inversion( )
{
  ID_INITIAL_13_L:
  if( !( Condition() ) ) {
    doAction();
    goto ID_FINAL_15_L;
  }

  ID_FINAL_14_L:
  return ID_FINAL_14;

  ID_FINAL_15_L:
  return ID_FINAL_15;
}
```

Note that in this code the labels *ID_INITIAL_13_L* and *ID_FINAL_14_L* without referencing goto jumps are generated which has two drawbacks: the extent of the source code increases and it can also produce warnings from a C/C++ compiler which can lead to complete fail of the build process if a compiler flag making the compiler treat warnings as errors is used (e.g. *-pedantic-errors* build flag in gcc/MinGW compilers).

## V. Conclusion

As shown in the paper, fully functional, production-ready source code can be generated by using nowadays modern CASE tools like CodeDesigner RAD. Moreover, the generated code can be optimized by using several preprocessing and optimizing algorithms so it has a form and structure near the source code written by human programmers. The paper also revealed how specific code generation algorithms influence overall properties of generated source code.

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

[3] 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.

[4] 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.

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

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

[7] Donald E. Knuth. *The Art of Computer Programming Vol 1.* Boston: Addison-Wesley, 3rd edition, 1997.

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