

Grape – Graphical Robot Programming for Beginners

Stefan Enderle

University of Applied Sciences Isny, Germany
qfix robotics GmbH, Senden, Germany
enderle@qfix.de

Abstract. Using robot kits for education in schools and universities, we found that there is a lack in tools for teaching the structures of an object oriented programming language. Thus, we decided to develop a graphical programming environment for the beginner, using procedural concepts together with given objects.

The described tool is able to produce C++ code from the graphical user input. So, the mapping between a flow chart and the syntax of the programming language is directly visualized. Additionally, the environment can easily be extended by the user to use additional C++ classes or to create code for different controllers or PC processors.

1 Introduction

The author works in the field of robot kits, which are modular kits (see [1]) consisting of mechanical and electronical components which can be used to build an autonomous mobile robot. After physically building up the robot platform, the created vehicle must be programmed by some programming environment, mostly running on the PC. These robot kits are used for example in school tournaments, like RoboCupJunior [2] where young students from an age of 10 years start to work with and to program robots.

Advanced students have no problem using a real programming language like C/C++ or JAVA. This is because the structural programming entities, like commands, loops, if-then-else-clauses, etc. which are the same for all procedural programming languages, are well-known. Thus, in this case learning a programming language means learning the *syntax* of a programming language. However, for younger students who do not know the structures of programming, it is important to introduce these entities in a graphical way (like flow-chart elements), and, to make them understand the mapping from the graphical design to the program source code.

Companies developing toy robot kits, like LEGO [3] or Fischertechnik, also provide graphical programming environments for their kits. However, these environments are all dataflow oriented, i.e. the boxes used in these systems are *processes* according to a “input-process-output- model”, or simpler *functions* with input and output. The ins and outs of such functional boxes can be combined to larger systems and e.g. the sensor data flows along the described channels being modified by the boxes until it flows into a last box, e.g. a motor, where they performs some action.

This approach is also used in sophisticated programming systems like labView or Simulink where even embedded realtime systems can be programmed graphically.

However, due to their complexity these systems cannot be used for absolute beginners. And, we think that the procedural approach where a program is firstly considered as a sequence of statements is much easier to learn than the data flow or functional approach.

This led to the development of *Grape*, an easy-to-use graphical programming environment with which a flow chart (representing the program flow) can be built and the meaning of the individual elements of the flow chart are defined. The complete steps of creating a *Grape* program are explained in the following.

The main design issues for *Grape* are the following:

- **Extensibility:** The set of classes that is given for the user should be easily extended by additional classes. Thus, a simple class description language must have been created.
- **Generic program representation:** There should be a generic XML representation of the graphical program to be able to write additional tools, like translators to different programming languages.
- **Automatic code generation:** The tools should be able to generate C++ (or any other OO language) source code in order to directly see the translation from the graphical program identities to the respective lines of code.
- **Platform independent:** The tool should run on at least Windows and Linux. So, we chose to use the qt toolkit ([4]) for implementation.

Most of these design issues are discussed in more detail in the following.

2 Grape

The easiest way to introduce *Grape* is to look at a small standard project and how it is implemented using *Grape*. First, have a look at a typical “hello robot-world” program in C++:

```
#include "qfixSoccerBoard.h"

SoccerBoard robot;

int main()
{
    robot.motor(0,255);
}
```

This will let one motor of the robot turn with full speed. In the short example you can see a typical sequence of four basic actions:

- Including a (class) library
- Declaring an object
- Defining the main function
- Using the object by calling its methods

The next sections show how these basic actions are performed in *Grape* leading to a graphical program. After that we briefly look at the process of automatically transforming the graphical program to C++ code.

2.1 Classes and Objects

Grape comes with a list of predefined classes for robot programming. Figure 1 shows the tab “Classes and Objects” where classes can be included and objects of the included classes can be declared.

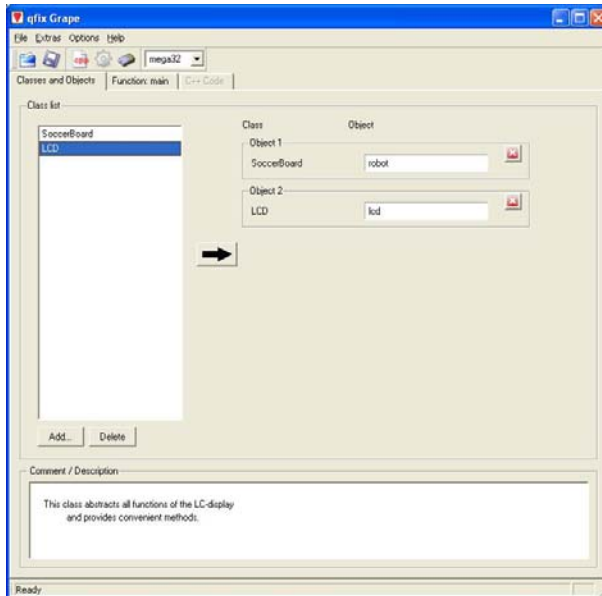


Fig. 1. Two included classes and objects thereof

In the Figure 1 the two classes (`SoccerBoard` and `LCD`) are included and an object of each class (called `robot` and `lcd`) are declared.

This list of predefined classes can be extended by own classes using a simple XML syntax which is described in Section 3.1.

2.2 Graphical Programming

The second tab “Function: main” consists of an almost empty grid holding only the stub of the main program represented by a `Start` and a `End` node (see Figure 2).

Here, to program graphically means to arrange symbolic blocks to yield the desired flow chart. The available building blocks are represented on the left side of the window and the user has to drag-and-drop them over an already existing arrow in the program. Figure 3 shows a first program containing an infinite loop.

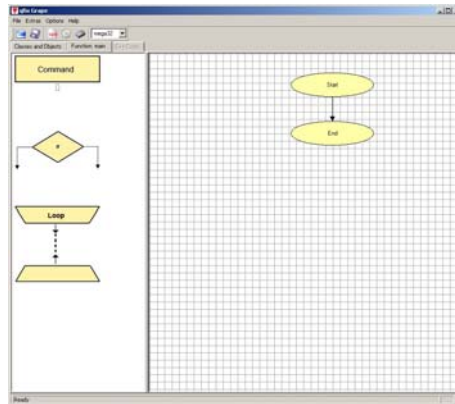


Fig. 2. Main window with program stub

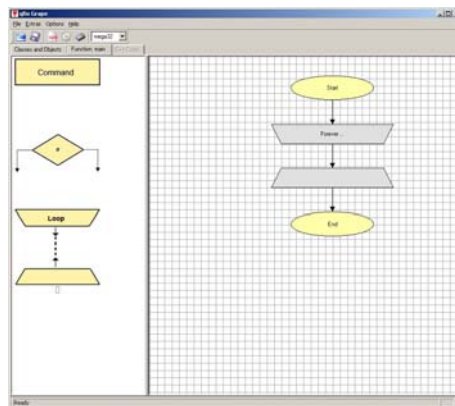


Fig. 3. Infinite loop as first element

The symbolic building blocks currently include the following programming concepts:

- commands (or statements)
- if-statements with else-clause
- loops

Figure 4 shows a graphical program including all three concepts.

Further programming concepts that are not implemented yet are

- own functions or procedures
- variables

Note that the concept of *dragging over an arrow* does not allow the user to produce a logically incorrect program! So, for example, it is not allowed to have an if-statement with a *true*-path ending at another place as the *else*-path. This would result in a “goto”-functionality (as it is done in other graphical programming systems in which boxes and arrows can be placed arbitrarily).

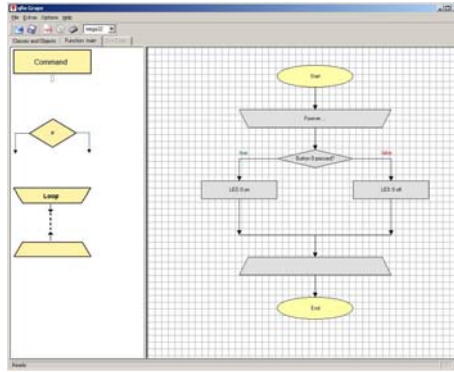


Fig. 4. Graphical program including commands, if-statement and loop

2.3 Dialog-Based Implementation

In the last figures, we see that the selected programming elements are grey. This means that the logical structure is existing, but their meaning (semantics) is not defined, yet. The user can decide if he first finishes the graphical program and then implements the individual boxes, or if he performs the two steps in parallel. However, before being able to generate source code, all boxes have to be implemented with the desired semantics.

The indicator for an element being defined or not is its color. A grey box indicates the pure element without meaning, a yellow box indicates an implemented command, loop, etc.

In order to define the semantics of a specific element it can be double-clicked which opens the respective dialog for the implementation. Figure 5 shows the dialog belonging to a command box.

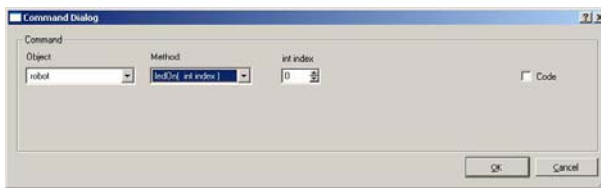


Fig. 5. Command dialog

You can see that one of the declared objects can be selected and one of the selected object's methods can be chosen. According to the method's signature the respective arguments can then be specified. With this approach it is not possible to produce any syntactical error.

A respective dialog is used for loops and if-then-else clauses.

3 Internal Representations

3.1 Class Representation

The classes that can be used in the *Grape* environment are regular C++ classes which are represented by an XML file describing the following properties of a class:

- the class name
- the C++ header file with the class definition
- the class’ methods including their parameters (name, type, and value range)
- descriptions of the class and the methods

See the following XML code for a representation of the class `BobbyBoard` which represents a robot controller board (the `DOCTYPE` header is left out for convenience):

```
<qfixClass>
  <header>
    <includefile name="qfixBobbyBoard.h" />
  </header>
  <class name="BobbyBoard">
    <description>
      This class abstracts all functions of the
      BobbyBoard and provides convenient methods
      to access all input and output channels.
    </description>
    <method type="void" name="motor" >
      <params>
        <param type="int" name="index"
          min="0" max="1" />
        <param type="int" name="speed"
          min="-255" max="255" />
      </params>
      <description>
        Sets the motor with the given index to
        the given speed.
      </description>
    </method>
    <method type="int" name="analog" >
      <params>
        <param type="int" name="index"
          min="0" max="3"/>
      </params>
      <description>
        Returns the value of the given analog
        input port.
      </description>
    </method>
  </class>
</qfixClass>
```

A class description file is read when the user switches to the first tab "Classes and Objects" and presses the "Add.." button in order to add a new class.

3.2 Flow-Chart Representation

In Figure 4, we saw a graphical program consisting of commands, an infinite loop and an if-then-else clause. Now, we describe the internal representation of this flow-chart, firstly regarding only the structure of the program without code.

Displayed graphically, the XML structure looks like displayed in Figure 6 (note, that we left out several nodes for better overview).

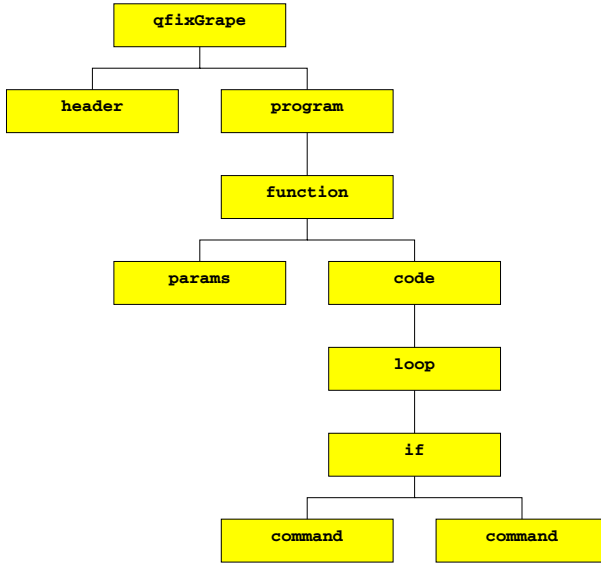


Fig. 6. Rough XML structure of the program in Figure 4

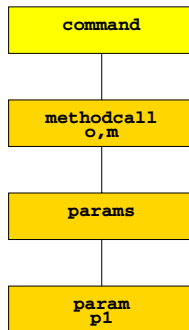


Fig. 7. XML structure of a command with implementation

3.3 Code Representation

When the user uses the dialog-based implementation to fill a graphical element with its semantics, the internal XML representation is updated in the way that the respective graphical element receives a child node describing the implementation of the concept.

For example, when one of the `command` nodes above is filled with the implementation *call object o , method m with parameters $p=\{p1\}$* , the command node expands to the one in Figure 7. Thus, one can say that the semantics information is *embedded* into the graphical representation.

4 Code Generation

The XML representation discussed above can be automaticall translated into C++ (or any other object oriented programming language). This is done by a relatively simple mapping schema from XML to C++ which is explained in this section.

As an example, mapping the graphical program from Figure 4, which ist internally represented by the XML code in Section 3.1, produces the following C++ source code:

```
#include "qfixSoccerBoard.h"
#include "qfixLCD.h"

SoccerBoard robot;
LCD lcd;

int main()
{
    while (true) {
        if (robot.button(0)==true) { // forever ...
            robot.ledOn(0); // Button 0 pressed ?
        } // LED 0 on
        else {
            robot.ledOff(0); // LED 0 off
        }
    }
}
```

This mapping is performed by applying the following schema:

- First, the `<header>` node is visited:
 - nodes `<includefile>` expand to


```
#include "<name>"
```
 - nodes `<global>` expand to


```
<type> <name>;
```
- Then, the `<program>` node is visited:
 - nodes `<function>` expand to


```
<type> <name> (<params>)
{
    <code>
}
```

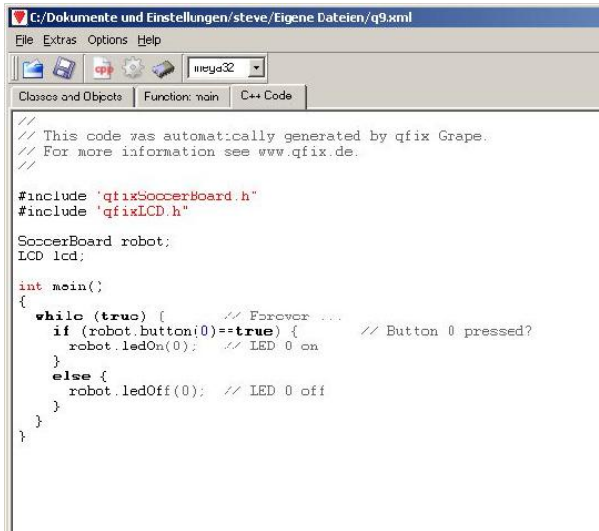

- nodes `<loop>` of type `forever`¹ expand to


```
while(true) {
    <code>
}
```
- nodes `<if>` expand to


```
if(<left-term><operand><right-term>)
{
    <true>
};
else {
    <false>
}
```
- nodes `<command>` of type `methodcall`² expand to


```
<object>.<method>(<params>);
```

The user, obviously, is not bored with these mapping details. When having finished his graphical and dialog-based development – i.e. all graphical elements turned to yellow – he simply presses the function key F4 and receives the respective C++ source code in the third tab “C++ code” (see Figure 8).



```

C:/Dokumente und Einstellungen/steve/Eigene Dateien/q9.xml
File Extras Options Help
meya32
Classes and Objects | Function: main | C++ Code

// This code was automatically generated by qfix Grape.
// For more information see www.qfix.de.
//

#include "qfixSoccerBoard.h"
#include "qfixLCD.h"

SoccerBoard robot;
LCD lcd;

int main()
{
    while (true) { // Forever ...
        if (robot.button(0)==true) { // Button 0 pressed?
            robot.ledOn(0); // LED 0 on
        }
        else {
            robot.ledOff(0); // LED 0 off
        }
    }
}

```

Fig. 8. Respective code in GRAPE

5 Tools

Working with mobile robot kits, what you always have to do despite coding is to compile your program and download it to the robot controller. Thus, it is desirable to have

¹ Other loop types including conditions are possible.

² Other command types are possible.

the compile and download tools integrated into the programming environment. However, since there are often multiple controllers that must be supported at the same time, the environment must be kept flexible in order to integrate multiple tool sets according to the required task.

In *Grape*, we use a schema-based configuration concept to import additional tools and being able to activate them by a shortcut key. A tool is represented by a command-line (e.g. a compiler call or a shell script) with additional arguments e.g. for the program name or directory.

in the XML tools configuration file you can add your own tools to the desired schemas and configure a hot-key which shall be used to invoke the respective tool. See the following code for an exemplary configuration file with three schemas and two tools for compilation and download in each schema:

```
<qfixGrapeConfig>
  <schemes>
    <scheme name="mega32">
      <tool name="compile" key="F5"
        command="c:\WinAVR\compile-mega32.bat"
        params="%f" />
      <tool name="download" key="F6"
        command="c:\WinAVR\download-mega32.bat"
        params="%f" />
    </scheme>
    <scheme name="mega128">
      <tool name="compile" key="F5"
        command="c:\WinAVR\compile-mega128.bat"
        params="%f" />
      <tool name="download" key="F6"
        command="c:\WinAVR\download-mega128.bat"
        params="%f" />
    </scheme>
    <scheme name="can128">
      <tool name="compile" key="F5"
        command="c:\WinAVR\compile-can128.bat"
        params="%f" />
      <tool name="download" key="F6"
        command="c:\WinAVR\download-can128.bat"
        params="%f" />
    </scheme>
  </schemes>
</qfixGrapeConfig>
```

In this example, you can see three schemas “mega32”, “mega128” and “can128” which correspond to three different microcontroller settings. For each schema, the tools “compile” and “download” are defined which are bound to the hotkeys F5 and F6. Pressing one of these keys, the respective command line is executed with the current filename (see the “%f”) as argument.

Note that this solution is not only flexible enough for compiling and downloading programs for a family of similar robots with similar controllers. It is also possible to

call completely different tools for other robots with different controllers or even to call an independent program, like LaTeX or a compiler for a PC GUI applications.

6 Experiments

In order to demonstrate the feasibility of the *Grape* system, a development task was given to five groups of students. They should developed a graphical program with *Grape* and use the additional tools for compiling and downloading to the robot. The task was to have a small robot equipped with an analog light sensor to follow a black line. At the beginning, the program should wait for a start button. While moving along the line, it should check if a stop button was pressed and in that case stop the motors and wait for the start button again. Figure 9 displays the used robot.

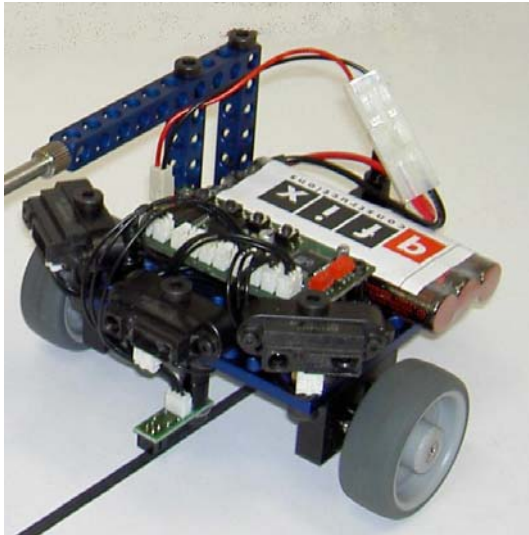


Fig. 9. Differential drive robot with two driven wheels, three IR distance sensors and a analog light sensor for ground line detection

The solution program that all students could easily solve is shown in Figure 10. After waiting for the start button it runs into the main loop which checks if the robot is on the line. If so, a slight left curve is driven, if not, a slight right curve is driven. After that selection, the stop button is checked. If it is not pressed, nothing happens. If it is pressed, the motors are stopped and the program waits until the start button is pressed.

From top to bottom, the program entities are filled with the following semantics:

```
- robot.waitForButton(0)
- forever
- if (robot.analog(0) <= 120)
- robot.motors(-100,255) / robot.motors(-255,100)
- if (robot.button(1) == true)
- robot.motorsOff()
- robot.waitForButton(0)
```

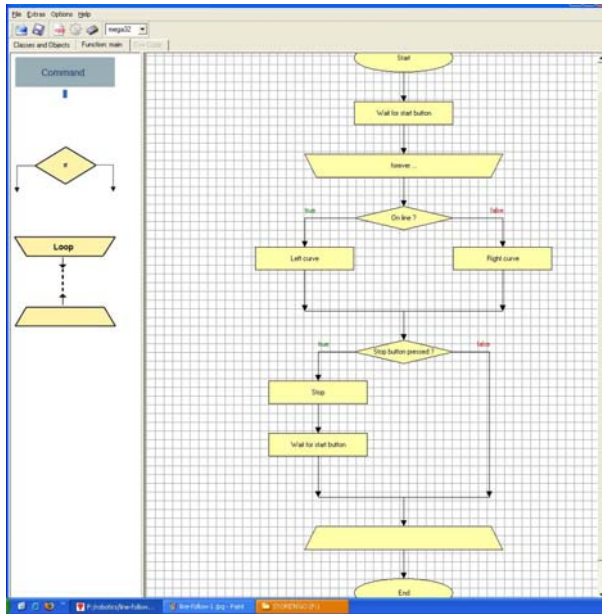


Fig. 10. Solution program for line following

Generating C++ source code for this program generates the following code which can be compiled and downloaded to the robot controller.

```
#include "qfixBobbyBoard.h"

BobbyBoard robot;

int main()
{
  robot.waitForButton(0); // Wait for start button
  while (true) {          // forever ...
    if (robot.analog(0) <= 120) {
      // On line ?
      robot.motors(-100, 255); // Left curve
    }
    else {
      robot.motors(-255, 100); // Right curve
    }
    if (robot.button(1) == true) { // Stop button pressed ?
      robot.motorsOff();          // Stop
      robot.waitForButton(0);     // Wait for start button
    }
  }
}
```

Playing around with the threshold value (here: 120) the robot is indeed able to follow a black line by wobbling along the black/white edge of the line. The experiment showed that the students could easily use the Grape tool for developing such behaviour.

7 Conclusion

We presented *Grape*, a graphical programming environment for object oriented programming. The main goal of *Grape* is the students to understand the *concepts* of programming languages including commands or statements, loops, if-clauses, etc. With *Grape*, the student can play around with these concepts in a graphical way.

To go further into programming, the student has to define the *semantics* of each graphical entity with respect to his program in mind. From this, *Grape* can generate C++ code which can also be studied in order to learn the mapping between the flow-chart and the source code. This is enforced since the user can immediately see how a change on the graphical side affects the generated code.

Thus, *Grape* is a powerful tool for educational purposes and has a good chance for being used in other areas despite programming small mobile robots, too.

8 Open Work

Note that currently, only a subset of object-oriented programming is supported by *Grape*, namely the usage of existing predefined classes. It is not supported to define own classes within *Grape*, yet. However, for a novice of programming, this is sufficient.

The next programming concepts to be included into *Grape* are subroutines/functions and variables. Subroutines and functions can be easily integrated since the function concept is already defined (see the main function in the examples above). The representation of variables is still open.

Especially for robotics applications, the concept of multiple processes/threads is important, since the robots can be programmed much easier using a behaviour-based approach including multi-threading.

Acknowledgements

This work is sponsored by qfix robotics, Germany (www.qfix-robotics.com). We also thank Bostjan Bedenik who mainly implemented the Grape software.

References

1. Enderle, S.: The robotics and mechatronics kit qfix. In: Lakemeyer, G., Sklar, E., Sorrenti, D.G., Takahashi, T. (eds.) RoboCup 2006: Robot Soccer World Cup X. LNCS, vol. 4434, pp. 134–145. Springer, Heidelberg (2007)
2. RoboCupJunior, <http://www.robocupjunior.org/de>
3. Baum, D., Gasperi, M., Hempel, R., Villa, L.: Extreme Mindstorms – An Advanced Guide to LEGO Mindstorms. Apress (2000)
4. Trolltech: Qt c++ library, <http://www.trolltech.com>