Structured Robot-Control Language on Dolittle Syntax

Toshiyuki Kamada, Hiroyuki Aoki, Shuji Inoue, Shuji Kurebayashi

Aichi University of Education, Tokyo Gakugei University, Studio MYU, Shizuoka University

tkamada@auecc.aichi-edu.ac.jp, aokih@cs.u-gakugei.ac.jp,

shu_inoue@hotmail.com, eskureb@ipc.shizuoka.ac.jp

Abstract

Control programs for autonomous robots have to handle sensor inputs for condition determinations. Hence, such programs tend to become complicated. We have developed a robot controller board which can be programmed with virtual CPU instructions. The virtual CPU encapsulates details of I/O and irregular instruction sets, to make robot programming easier. However, virtual CPU instructions are still based on conditional jumps and thus was difficult to program for novice programmers. To overcome the problem, we have implemented a robot control language with structured syntax on top of Dolittle [1, 2, 4] environment. In this paper, we first explain firmware of our robot control board "MYUROBO" [5], and previous Dolittle-based programming envirionment for the board. Then we discuss needs for structured control flows and explain our new (structured) language constructs, along with its implementation.

1 Introduction

Control programs for autonomous robots have typical structure of branching according to mutiple senser inputs (including changing of internal status values of the robot according to its behavior) and taking corresponding actions (Fig. 1). However, many robot programming environment uses conditional jump instructions to control program flow, and the structure explained above is difficult to construct with such primitive instructions. The situation actually holds for MYUROBO [5] robot controller board we have developed.

An answer to the problem is to prepare higher-level language with structured control flow, which is translated to the robot instructions by the compiler. We have first done that with MYU BASIC processor, then with Dolittle [1, 2, 4] language.

Previously, simple translator written in Dolittle were used to generate MYUROBO instructions and send them to the controller board, but the translater had not supported structured control flow. To overcome the problem, we have revised the previous implementation and developed a new structured robot control language on Dolittle. The strong points of our new language is that the language largely uses Dolittle syntax and dolittle environment, so users do not have to learn new programming environment or new syntax (although some differences exist between the robot language and original Dolittle).

In this paper, firstly the firmware part of MYUROBO robot controller board that we have developed is described. Secondly, the way of utilize the MYUROBO from some high-level languages is shown. Thirdly, the programming environment based on structured Basic in order to make sure the benefit of structured programming language for robot programming is introduced. Lastly, the design and implementation of structured robot-control language on Dolittle syntax is explained.

2 The firmware of MYUROBO

The MYUROBO controller board is composed of a microcontroller, 4 input ports, and 6 output ports. The output ports have three pairs of connectors; each pair controls one DC motor for forwared and reverse rotation. In the microcontroller, the flash memory and the EEPROM are embedded. Flash memory includes MYUROBO firmware

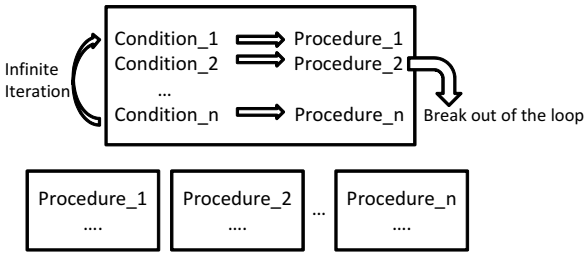

Fig. 1 Typical architecture of robot-control program with multiple sensors

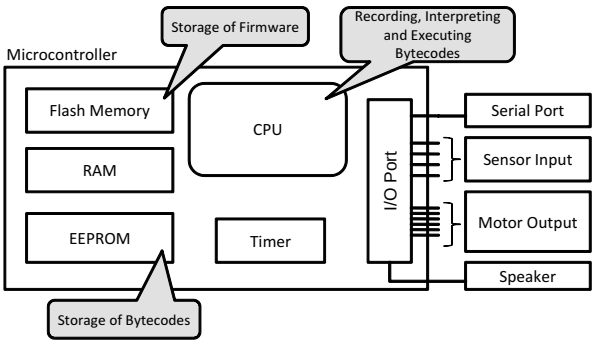

Fig. 2 The microcontroller of MYUROBO and the role of each part

which implements virtual CPU interpreter and program loader. Acutal robot programs are written as virtual CPU instructions (bytecodes); they are loaded into the EEPROM and executed by the interpreter. Fig. 2 shows the architecture of the microcontroller in MYUROBO and the role of each component.

2.1 Bytecode interpreter of MYUROBO

The bytecode interpreter of MYUROBO is not only a sequencer of recorded instructions. It offers a virtual CPU which has functionalities of memory transfer and calculating operations. The conceptual diagram of the implemented virtual CPU is shown in Fig. 3. The virtual CPU is modelled after a register machine. The summary of registers in this CPU is shown in Table. 1. The size of each register is 8-bit. The numerical and logical operations which are executed on the A register changes status flags (C for carry and Z for zero). The status flags affect the behavior of conditional branch instructions.

Table. 1 Registers of virtual CPU

Name	Function
A	Accumulator
IX	Index register
T1H, T1L	Timer value (16 bit)
PA	Input port
PC	Output port

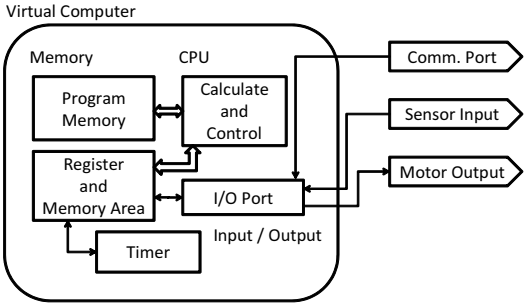

Fig. 3 The conceptual diagram of virtual CPU realized by the MYUROBO firmware

2.2 The benefit of virtual CPU

Instead of downloading actual machine code, we have used virtual CPU and its programs (bytecodes), due to the following reasons:

- Register, memory and I/O architecture of microcontroller are complex; we wanted to hide such details.
- By using abstract instruction, we can freely move to different models of microcontrollers or add new instructions in a backward compatible manner.

From the above reasons, by designing virtual CPU properly, it can offer an abstract simple architecture and an instruction set to the programmer. This makes the programmer write programs which is suitable to the robot control easily.

Generally, abstraction of the hardware is offered by libraries, but there is an advantage to detach functions from the language by offering abstracted instructions of the virtual CPU as bytecodes. In the case of using libraries, the programmer has to keep their APIs in mind, but in the case of using a virtual CPU, the programmer can write robot

Table. 2 Summary of MYUROBO instructions (excerption)

Instruction	Function
startrobot	start of MYUROBO program
endrobot	end of MYUROBO program
n forward	rotate both wheels forward (in $n \times 0.1$ seconds)
n back	rotate both wheels backward
n turnright	turn the robot to the right
n turnleft	turn the robot to the left
n stop	stop the robot
n m tone	play a sound of m-width pulse
i label	label with i
i jump	jump to label i
i j jumpifhigh	jump to label i if the status of sensor j is high
i j jumpiflow	jump to label i if the status of sensor j is low
i blockstart	start of the block i
blockend	end of the block
i executeblock	execute block i
i n repeatblock	repeat block i in n times
exitblock	exit from a current block
n A	set n to A register
n ADD	add n to A
n SUB	substitute n from A
INCA	increment A
n CMP	compare A with n
n AND	bitwise AND
n OR	bitwise OR
JZ	jump if zero
JNZ	jump if not zero
JC	jump if carry is set
JNC	jump if carry is not set

control codes naturally by using a compiler or a converter from a well designed high-level language to the bytecodes.

2.3 Instructions defined in the bytecode

The implemented bytecode has mainly two category of instructions; one is optimized instructions for robot control such as motor control or branch by the status of sensor input, and the other is instructions the same as which in other general purpose CPUs such as transferring data between registers or numerical and logical operations.

Table. 2 is a summary of typical MYUROBO instructions.

The allocation of bytecodes and naming of mnemonics corresponding to them are carefully defined as upper compatible to our previous "biaxial robot controller board" [3]. All bytecodes allocated in previous board are included in the new bytecode system.

3 Programming environment for MYUROBO controller board

Every bytecode have corresponding mnemonic. Then the most primitive method to program the MYUROBO board is manually translating mnemonics (operation names) to corresponding numbers (op-codes), representing them as series of numerical data and prepare a software that reads numerical data and sends them to the board. However, this procedure is tedious and error-prone task. Therefore, there is a need to develop a integrated programming environments that can translate MYUROBO instructions to bytecodes automatically and send them to the board.

3.1 Programming environment with Dolittle

Dolittle is an educational, object-oriented programming language. we were using Dolittle for elementary, junior-high and high school programming education. Then it is natural for us to use Dolittle system also for robot programming.

In order to alleviate problem which have mentioned above, we have developed simple robot language (mnemonic) translator on top of Dolittle. The main idea is to define method for each of the mnemonic, and the method outputs corresponding op-code to the serial port, optionally with accompanying operand data.

The actual sample code is shown in Fig. 4. In our Dolittle envitonment, "serialport" object is predefined, which handles serial port of the PC. The "serialport" object has "open", "close", and "write" methods. They correspond to "open the specified port", "close the port", and "send a data to the port" functionality. We defined additional methods which names are the same as mnemonics. In these methods, we wrote codes to send correspondent bytecodes and accompanying operand data to the serial port. Then, the programmer can write a robot control code as three steps like (1) opening the serial port, (2) doing some mnemonics method invocations, (3) closing the port. In Fig. 4, mnemonics are written in separate method

```
robo=serialport!create.
robo:transfer=[!
  startrobot
    20 forward
    10 stop
    20 back
    10 stop
  endrobot
].
robo!"com1" opensesame.
robo!transfer.
robo!closesesame.
```

Fig. 4 Example program of MYUROBO code in Dolittle

named "transfer". By using this technique, the programmer can concentrate on writing mnemonics because the other part of the program (before "startrobot" and after "endrobot") is completely the same in every program.

3.2 Structured control and MYU BASIC

Dolittle environment explained in the previous section was quite usable for simple program targeted to two-motor robots. However, when the programmer starts to handle multiple sensors in the code, it becomes complicated and the readability of it degrades. The problem resembles to that causes in general assembly code; "spaghetti program".

Robot programs with mutliple sensors must have series of conditional dispatch and corresponding actions, and its complex structure is difficult to describe naturally in primitive conditional jumps. To overcome the problem, programming language with structured control flow was necessary.

As an asnwer to the problem, we first developed "MYU BASIC" [6] by modeling after the structured Basic language. MYU BASIC is implemented by Visual Basic and it is a complete system that can edit source file, compile it, and send bytecodes into the controller board. Table. 3 shows the structured syntax adopted by MYU BASIC.

Another noteworthy feature of MYU BASIC is a declaration of variable. In the MYU BASIC program, single byte sized integer variable can be declared. By using this, the programmer can access memory area of the controller board without handling index register directly. Thanks to this feature, the user can easily program the robot behavior which responds to a status variable. For example, the robot can recognize how many times it turns at corners.

Table. 3 structured syntax of MYU BASIC

Function	Syntax
Condition	If ... Then ... [Else ...] EndIf
	IfHi(n) Then ... EndIf
Iteration	Do ... Loop
	For ... Next
Procedure	Proc ... EndProc
Function	Function ... EndFunction

3.3 Design of structured robot-control language on Dolittle syntax

From the experience of MYU BASIC, we have learned there is a good effect in the readability of the code by using the structured syntax. Then, we return to Dolittle. Because the syntax of Dolittle is simple and easy to understand, we think it is worth challenging to develop structured robot-control language on Dolittle syntax.

At first, we examine the same syntax with original Dolittle as robot-control language, but we abandon this idea because we notice that there is a semantic difference between the concepts of object-oriented programming in Dolittle and the autonomy of the robot. In the program of Dolittle, multiple objects communicate each other with message passing. On the other hand, in the robot-control program, a robot object describes entire robot. If the robot object communicate to other objects in Dolittle, it means that the robot has to communicate with Dolittle system while its execution. This contradicts the autonomy of the robot.

Consequently, we have determined "MYU" object which inherits original "serialport" object and designed to implement all structured commands as methods of this object. The designed structured syntax is shown in Table. 4.

The syntax of the conditional branch and the iteration resemble to original Dolittle except for the position of "!" sign and the absence of a period at the end of the line. In addition, the condition part is limited to testing the status of the sensor

Table. 4 Structured robot-control language on Dolittle

Function	Syntax
Condition	[!n ifhigh] then [...] execute
	[!n iflow] then [...] else [...] execute
Iteration	[!n ifhigh] whilerepeat [...] execute
	[...] n repeat
Terminating Iteration	break

```
robo=MYU!"com1" create.
robo:collision_avoidance=[!
    [!1 iflow] whilerepeat
      [!10 back 10 turnright] execute
].
robo:program=[!
  startrobot
  [!
    forward
    [!2 ifhigh] then [!collision_avoidance] execute
  ] repeat
  endrobot
].
robo!program.
```

Fig. 5 Example program of structured robot-control language on Dolittle

input. For this purpose, only "ifhigh" and "iflow" methods can be used in the condition part.

Furthermore, in order to enable composing compound conditions, we introduced "and" and "or" methods.

The example program using this structured language is shown in Fig. 5. This program includes a definition of a subroutine by method definition, a method invocation, two conditional branches, and iteration.

4 Implementation of converter to MYUROBO instructions with Dolittle

In this section, the implementation detail of above structured language on Dolittle system is described.

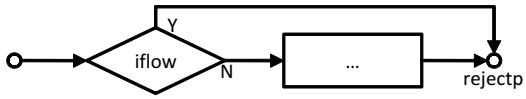

Fig. 6 Simple conditional branch

4.1 Simple conditional branch

As described above, the condition part of a conditional sentence is limited to "ifhigh" and "iflow" methods. These methods test the status of the sensor input by specifying a number of input ports as an argument of the method.

For example, in order to execute the statements when the status of sensor 1 is high, write like this:

```
[!1 ifhigh] then [...]  execute
```

This also means that "*when the status of sensor 1 is low, skip the statements and goes to the next line of the statements*". Therefore, in converting to MYUROBO instructions, "ifhigh" method of structured language should convert to "jumpiflow" of MYUROBO instruction and "iflow" method should convert to "jumpifhigh".

As for jump target, the converter decides the label value and stores in the internal variable "rejectp". The "then" method have a role to evaluate the block of condition part like "`[!1 ifhigh]`". Then "execute" method evaluates the argument block and outputs "label" instruction with its value is "rejectp" at the last part of processing. Fig. 6 shows the diagram of this conversion procedure.

4.2 Conditional branch including else clause

The example of conditional branch including else clause is as follows:

```
[!1 ifhigh] then [...]  else
                [...]  execute
```

In this case, after processing "then" method, "else" method evaluates argument block first, decides new label value and stores in the internal variable of "(new) rejectp" and outputs "jump" instruction to there. Then, the converter outputs "label" instruction which value is "(old) rejectp" and disables this old value. As a result of this process, the "execute" method outputs "label" instruction with new value of "rejectp". Fig. 7 shows the diagram of this conversion procedure.

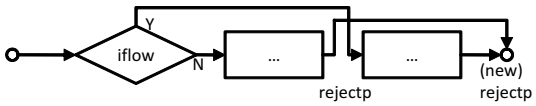

Fig. 7 Conditional branch including else clause

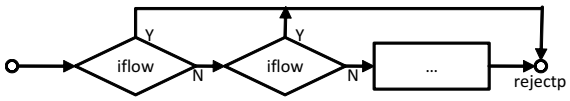

Fig. 8 Compound conditional sentence using "and"

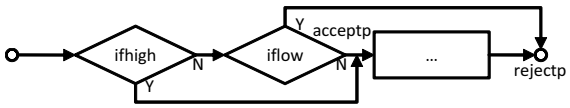

Fig. 9 Compound conditional sentence using "or"

4.3 Compound conditional sentence

The compound conditional sentence uses "and" and "or" methods.

The conversion procedure of "and" method is the same as "then" method. The diagram of Fig. 8 shows the correctness of this.

On the contrary, "or" method needs more complex conversion process.

In converting "or" method, the condition part does not invert the logic. Then "ifhigh" method in the condition part of "or" method is converted to "jumpifhigh" instruction. In addition, the name of internal variable is "acceptp" instead of "rejectp". With this difference, the "execute" method should have an additional process to output "label" instruction with its value of "acceptp" at first. Fig. 9 shows the diagram of this conversion procedure.

4.4 Iteration

The "whilereapet" method realizes an iteration using a condition. In order to prepare startup point of the loop, the converter first decides the value of internal variable "loopp" and outputs "label" instruction with the value. Then by processing the condition part with the same manner of "then" method, it enables to jump to the "rejectp" point when the loop condition becomes false. In order to form the loop, the "execute" method should have an additional conversion process to output "jump" instruction to the point of "loopp" between the evaluation of the argument block and outputting "label" instruction of "rejectp" point. Fig. 10 shows the diagram of this conversion procedure.

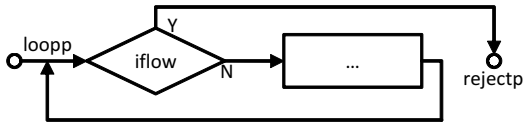

Fig. 10 Iteration using a condition

The "repeat" method realizes an iteration using a count. In converting this method, the "block" functionality of MYUROBO instructions are used. As the "block" of MYUROBO can iterate with the "repeatblock" instruction, the converter defines the value of the internal variable "blocknum" and outputs "repeatblock" instruction with the value. However, because the nesting of MYUROBO "block" might cause a problem of ambiguity, the evaluation of the argument of "repeat" method (body of the iteration) is delayed by storing the evaluation code into the "blockname" array as a string and actually evaluated in the execution of "endrobot" method by sequentially processing the member of the "blockname" array. Then all of the blocks are sequentially aligned at the end of resulting instructions.

The termination of the iteration is processed as in the case of "whilerepeat", outputting "jump" instruction to the point of "rejectp" and in the case of "repeat", outputting "exitblock" instruction.

4.5 Subroutine by the method

Because the content of a defined method is actually a block of Dolittle, invoking the method in robot-control program causes an expansion of the content of the method. This mechanism is the same as macro expansion in general purpose programming languages. The method definition is effective to raise the visibility of the code, but because of this mechanism, numbers of method invocations enlarges the size of converted MYUROBO codes.

Then, we prepare "blockize" method to realize the original meaning of the subroutine by using the "block" functionality of MYUROBO. The usage of "blockize" is writing code as following instead of writing method invocation:

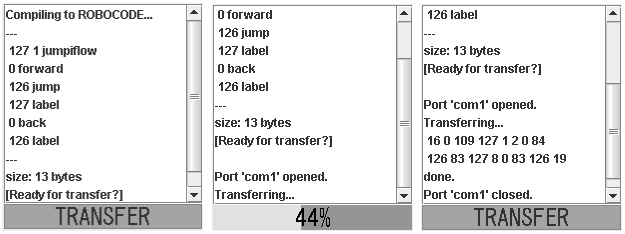

Fig. 11 Visual elements of MYU object

`!"method_name" blockize`

The "blockize" method outputs "executeblock" instruction here and store "method_name" into the "blockname" array. With this mechanism, the content of the defined method is expanded as the "block" of MYUROBO in processing the "endrobot" method.

4.6 Visualization of MYU object

The conventional "serialport" object does not have any visual element which every other Dolittle objects have. Then we added visual element to "MYU" object in order to the user can figure out the status of the object. In the process of conversion, the "MYU" object stores converted MYUROBO code into an array and when the user presses the "TRANSFER" button, it starts sending bytecodes to serial port. Additionally, by invoking "robocode" method, the converted MYUROBO code appears in the visual element and the user can ensure the correctness of the converted code. Fig. 11 shows the visual element of "MYU" object. From the left, the first is an appearance at the moment of the converted code is shown, the second is an appearance of under transferring bytecodes, and the last is an appearance at the moment of the transferring have finished.

5 Discussion

Because every statements of the structured robot-control language on Dolittle syntax are implemented as methods of "MYU" object, they are interpreted and executed by Dolittle system. From this reason, the language is a part of Dolittle system and the user does not need to learn any other operations than original Dolittle system. However, there are some unnatural limitations caused by this implementation.

5.1 Unnatural description of receivers

Because each statement of the structured language is a method of Dolittle, it need to specify receiver. In Dolittle syntax, each statement have to start with receiver name and "!" sign. But if the receiver of the message is the same object, the receiver name can be eliminated. In addition, Dolittle have "cascade" notation and if the receiver of continuous messages are the same, even "!" sign can be eliminated. In consequence, the robot-control program on Dolitte syntax can write with single cascading messages as shown in Fig. 4. The robot-control program in Fig. 4 does not include any receiver name or "!" sign except for the beginning of the block (line 2). Indeed, there must specify receiver at the beginning of the block. Another example of robot-control program in Fig. 5 shows this fact. Because each statement of structured robot-control language has the form of block and method name, there are numbers of "`[!`" notation in the program. Since understanding the exact meaning why this "!" sign is required in robot-control program is difficult to novice Dolittle programmers, they are tend to forget to write it. Even worse, statements in which the evaluation of argument block are delayed such as "execute" does not cause Dolittle syntax error even if the "!" signs are missing. In such a case, whole of the block is ignored in converting to MYUROBO instructions. For example, the statement as following:

```
[!1 ifhigh] then [forward] else
                    [back] execute
```

does not raise syntax alert from Dolittle system but in the converted code, "`forward`" and "`back`" are not included. This seems to be an error which is difficult to notice for the user.

5.2 Difficulty in checking errors of a word order

Because each statement of the structured language is independent Dolittle method, even if the user makes a mistake of the word order or combining unrelated statements, the Dolittle system does not notice any errors to the user. For example, the following sentence:

```
[!1 ifhigh] whilerepeat [...]  repeat
```

is not an error in original Dolittle syntax. In this case, user's mistake can be detected by checking the value of "loopp" in the process of converting "repeat" statement because "repeat" statement does not need "loopp" variable. These kinds of

checking are included in current implementation of structured language, but there may be unnoted illegal combination.

5.3 Confusable with original Dolittle syntax

Because each word of the statement of structured language is the same as original Dolittle, users who are familiar with original Dolittle tend to confuse syntax. Inserting "!" sign like "`[···]!then`" and attaching periods at each line are typical confusion.

5.4 Unable to handle variables and expressions

The current implementation of structured language cannot handle variables. The only way to handle registers and the memory area of MYUROBO from Dolittle is writing MYUROBO instructions directly. Because any variables defined using Dolittle syntax are statically determined at the time of converting, those variables and expressions are converted as fixed number. We have no idea to handle expressions in the current implementation of structured language.

6 Conclusion

Because the firmware of our target robot controller board "MYUROBO" realizes a virtual CPU which runs its bytecode interpreter, the user can program with any kind of high-level language by developing the compiler or converter from the language to the bytecode.

We found there is a difficulty in productivity of programming by using the mnemonic which directly corresponds to the bytecode. Therefore we designed and implemented structured robot-control language on Dolittle syntax. Current implementation utilizes block of Dolittle and realizes control structures by inserting "label" and "jump" instructions of MYUROBO mnemonic before and after the block.

However, there are problems such as unnatural syntax or difficulty in handling variables and expressions in the implementation using Dolittle system. Therefore enhancing Dolittle system or developing independent compiler would be needed to overcome these problems. We are going to design new structured robot-control language based on a fully object-oriented model.

References

[1] Susumu Kanemune and Yasushi Kuno : Dolittle : An Object-Oriented Language for K12 Education. *EuroLogo2005*, Warszawa, Poland, pp. 144–153, 2005.

[2] Susumu Kanemune, Takako Nakatani, Rie Mitarai, Shingo Fukui and Yasushi Kuno : Dolittle — Experiences in Teaching Programming at K12 Schools. *Proc. of the Second International Conference on Creating, Connecting and Collaborating through Computing (C5)*, IEEE, pp. 177–184, Kyoto, Japan, 2004.

[3] Shuji Kurebayashi, Toshiyuki Kamada and Susumu Kanemune : Learning Computer Programming with Autonomous Robots, *Lecture Notes in Computer Science, Vol. 4226*, Springer, pp. 138–149, 2006.

[4] Dolittle Programming Language. http://dolittle.eplang.jp/

[5] MYUROBO Controller Board. http://www.geocities.jp/shuinoue/myurobo/ (Japanese)

[6] MYU BASIC COMPILER. http://www.geocities.jp/shuinoue/myurobo/myubasic.html (Japanese)