Journal of Achievements in Materials and Manufacturing Engineering
International Scientific Journal published monthly by the World Academy of Materials and Manufacturing Engineering

Introduction to solving task-level programming problems in logic programming language

K. Foit
Institute of Engineering Processes Automation and Integrated Manufacturing Systems,
Faculty of Mechanical Engineering, Silesian University of Technology,
ul. Konarskiego 18a, 44-100 Gliwice, Poland
Corresponding e-mail address: krzysztof.foit@polsl.pl

Received 17.03.2014; published in revised form 01.06.2014

ABSTRACT

Purpose: This paper illustrates one of the possibilities of using logic programming language to process task-level description of robot's program. The goal of such programming method is to achieve almost the same abstract level in human-machine communication as in human-to-human communication.

Design/methodology/approach: The task-level programming is very different from trajectory planning, although some algorithms from this field are used in subsequent stage of detailing of the program. At a higher level we only define the proper sequence of actions. This could be also done using logic processing languages (e.g. Prolog).

Findings: The approach shown in the paper allows to solve manipulation tasks at high level (the sequence of actions), but does not cover all the problems connected with manipulator movements like avoiding collisions or detailed description of the motion.

Practical implications: Due being immature, the mentioned method is not applicable in real world, but could be used as a base of further research.

Originality/value: Task-level programming and problem solving is a very current field of research and experiments in robotics. It is also in very early stage, so most of methods have only scientific mean, without wide application in the industry.

Keywords: Robotics; Task-level programming

Reference to this paper should be given in the following way:

K. Foit, Introduction to solving task-level programming problems in logic programming language, Journal of Achievements in Materials and Manufacturing Engineering 64/2 (2014) 78-84.

ANALYSIS AND MODELLING

1. Introduction

Modern industry is largely based on robotized production. There is no doubt, that most of the work on the production line is done by the programmed manipulators, without human intervention. For this reason, engineers are searching for efficient methods of robots' programming. The main goal is to achieve simplicity of programming while reducing the number of errors during programming process. This task is realized by wide spectrum of methods,

beginning from specialized high-level programming languages to the programming by demonstration. In addition to the trajectory planning the programmer must take into account the communication between robots and other machines [1]. The use of special interfaces (touch screens, force detectors, video devices etc.) could make the programming process more intuitive [2-4,6]. In order to help in everyday programming routines, so called code snippets are created. They are usually a small portion of the source code, which could be used in larger program (as a part of it). Snippets generally used to minimize repetition of the same code. In connection with code skeletons, snippets make programming easier [6]. This form of code developing is sometimes called "copy&paste programming" to emphasize the negative effect of this style. It has more in common with automatic program generation, using advanced graphics application to simulate and off-line programming of the robot [3,4,5]. The negative effects of such approach manifest themselves in the quality of resultant code: bad optimization, errors replication etc. Despite the fact that the rapid development of a program has several disadvantages, the method is regarded as most efficient way of programming.

Unlike the method mentioned earlier, an ideal solution could be to give the machine some short commands in order to achieve the desired effect. Since the ancient times, people dream of such possibility in relation to machinery or "artificial life form". The traces of this could be found in ancient writings (e.g. "Politics" by Aristotle), myths and legends ("The Golem of Prague"). The 20th century, with the development of science and industry, has brought new opportunities. In 1920's, Czech writer and playwright, Karel Capek defines the new word: "robot". Over the next decades, this word will be widely spread over the world by another science-fiction writer – Isaac Asimov.

The first attempt to create the task-level programmed machines was made by Westinghouse. In his factory several "robots" was produced, including "Mr Herbert Televox" and "Elektro the Moto-Man". Both mentioned machines were task-oriented ones and the proper action was activated by sound - technically, the "Elektro" robot was activated by human voice. In fact, the machine reacted to a number of impulses of light, which were created by every spoken word. Every command was connected with the proper number of impulses. It looks like a hoax, but in the late 1930's it was the fulfilment of the dream of mechanical servants. Nowadays, this trend continues. The home appliance market offers more and more devices controlled by voice commands or gestures; this includes smartphones, so called "smart tv", refrigerators etc. The voice control is also used in cars, where provides support for on-board equipment (radio, GPS navigation, air conditioning, telephone, etc.) without taking your hands off the steering wheel.

Contemporary industry is fairly conservative in the implementation of new IT solutions. On the one hand, this is due to economic calculations, on the other hand there is the need to preserve the safety, security and continuity of production. The voice control/programming, however, is for a long time known in robotics and mainly used in surgical robot control systems, where the operator should always keep hands on the joystick during the surgery. The voice control enforces brief commands that, in fact, provide sufficient quantity of information to complete the action described by the command.

Programming at the task level, poses major challenges for researchers dealing with this problem [7-9]. On the other hand, it can really simplify programming by making a programming language similar to natural language.

2. Information, meaning, completeness

The distinction between information and its meaning is particularly relevant at the task-level description. The description should be sufficiently exact in order to make interpretation clear and explicit. Natural, everyday language uses many simplifications, which leave a considerable margin for interpretation. For this reason, we should take a look at the language as an information medium, using certain principles adopted in the philosophy of information and communication theory [10-13].

First of all, we could look at the single word as a sign that carry some information. In general, it does not matter whether the sign is understandable – it is important to know that "it means something". For example, if somebody is familiar with the Latin alphabet, the words written in Russian or Chinese will be completely unreadable for him, but he will claim, but he will be right saying that they have a meaning. In order to determine meaning of these words, this person needs to know a foreign alphabet and a language. We can also assume a slightly different situation. We are on holiday in some exotic country, but we do not know the local language. If someone says something to us in this language, we automatically reject the information that cannot be interpreted (language) and look for other signs, e.g. the behaviour of the person, trying to figure out what are the intentions. Let's have another example: there is a forest and some traces indicate that a boar is prowling around. The ranger immediately identifies the signs and will be able to visualize an animal, while the random person could not even notice the traces.

The given examples point out three important things. First of all, the information is something that independently exists in the environment. The second important thing is the fact, that information could be correctly interpreted only when the "receiver" has the proper level of knowledge. Eventually, the information could be ignored, when there is no enough knowledge to interpret it.

Another issue is the completeness of the information. The incomplete information are very common in everyday life, for example the sentence:

Give me some milk, please. (1)

carries only the information about what somebody wants. There is no data about amount, type of milk or about container. In the typical life situation, we could complement the missing information using our knowledge and experience: if we know that person, who wants the milk, we could fulfil the wish. The knowledge could be supplemented by requesting the missing data, for example the waiter in the restaurant could ask the client some question, which will ensure that the dish will be served in accordance with the guest's order.

It could be seen, that the knowledge plays the main role in realization of defined task. Using the given example, we could define the needed sets of data: the type of milk, amount and container. In order to reduce the complexity, we could use some "standard variables". The general declaration for this type of variables could be written in pseudocode, as shown in Fig. 1.

```
type
    objects = (light, fat, semi-skimmed,
    jug, glass, cup);
    milk_type = light..semi-skimmed;
    container = jug..cup;
    amount = (full, half);
```

Fig. 1. The declaration of types (sets) in pseudocode

The procedure, which satisfies the demand "*Give me some milk*", could be written in pseudocode as it has been pictured in Fig. 2. This is very general approach, but it could be seen, that the parameters are required. The interpreter does not know what means "*some*" or what kind of milk should be selected. The properly formulated command should be in form:

Give me half glass of fat milk, please. (2)

The given sentence contains values of all variables used by the procedure. Using the parameters from the sentence (2), the procedure call should have the form like

Give_milk(fat,glass,half); (3)

```
Procedure Give_milk(
    milk: milk_type;
    selected_container: container;
    amount_of_milk: amount);

    begin
        Select(milk);
        Select(selected_container);
        Pour(milk, selected_container,
          amount_of_milk);
        Serve;
    end.
```

Fig. 2. The program in pseudocode

After substituting the parameters (passed in the procedure call) in place of the variables, the code takes the form shown in Fig. 3.

```
    begin
        Select(fat);
        Select(glass);
        Pour(fat, glass, half);
        Serve;
    end.
```

Fig. 3. The main part of program using actual parameters

The given example illustrates the need to precisely define the tasks and completeness of the provided information.

As it was mentioned earlier, the correct interpretation of the information requires a proper level of knowledge. This general term stands for the ability of the correct interpretation of a message, as well as the set of proceedings rules used during the realization of the task.

In the case of a program or procedure, the knowledge should be available in the form of data permanently registered in the program code, or in the form of references to external sources (file or database). The same applies to the rules and subroutines that can be available in the form of a universal library, dynamically linked with the code.

3. Splitting a task into parts; levels of detail

When editing a program at the task level, we do not focus on each step separately. Referring to the examples shown in Figs. 2 and 3, we do not analyse what is hidden under the names of procedures such as "*Select*" and "*Pour*".

In everyday life, when sending a certain command to a specific person, we assume that this command is understood and the person is able to interpret it correctly (Fig. 4). In fact, the specific name of procedure encodes the set of activities that concrete the action. For example, it is possible to write the procedure "*Select*" as follow in Fig. 5.

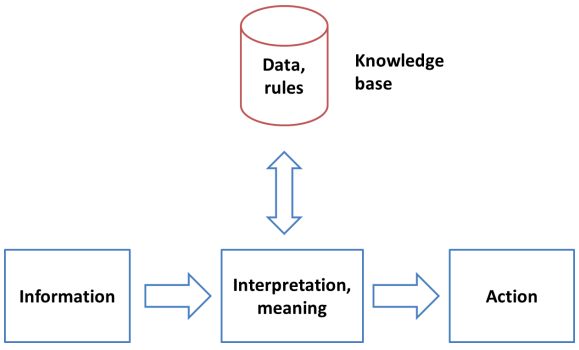

Fig. 4. General scheme illustrating data flow during interpretation of information

```
Procedure Select(obj: objects);
    begin
        ReachFor(obj);
        Grab(obj);
        GoBack;
        PutOnTable(obj);
    end;
```

Fig. 5. Hypothetical code of the procedure "*Select*"

The mentioned procedure consists of four another procedures: "*ReachFor*", "*Grab*", "*GoBack*" and "*PutOnTable*". The passed parameter "*obj*" is the type of object to select. As it was shown in Fig. 1, the "*objects*" is the enumerated type consisting of all possible types of milk and containers. This enables to reach both the specific type of milk and the container and makes the procedure "*Select*" more universal.

What makes this example a very important one is moving to a different level of detail. In fact, with the knowledge of the details of the procedure "*Select*", we can replace any of its occurrence in the code shown in Figs. 2 and 3, with respectively customized code from Fig. 5. This, however, would lead to the conclusion of quite detailed information at the highest level of generality. On the other hand there is no information about the commands "*Grab*", "*ReachFor*", "*PutOnTable*" and "*GoBack*" used in the code shown in Fig. 5. What we can do is to step down to another level of details and write the mentioned procedures.

A very important conclusion drawn from these considerations is the fact that the task level programming resembles the structure "from general to specific". The definition of the task itself is not enough to implement it without the completion of the lower level procedures. This results in a tree structure (see Fig. 6), where different levels represent different quantity of details. This eventually leads to the definition of elementary, indivisible actions, which form the lowest level of the program's structure. This is important from the point of view of further code processing (for example into a form of robot's program), but on the general level of considerations, this may lead to unnecessary obfuscation of code. The task-level programming should give the fast and clear solutions, which takes into account the imposed conditions and restrictions. The examples of such programming tasks could be famous "Towers of Hanoi" problem or "River crossing puzzle". These puzzles could be easily solved and described using task-level approach with the support of logic programming.

4. The Prolog programming language

The Prolog language was developed in early 1970's. It is hard to say that it is one of the programming languages used every day. The main difference between Prolog and other programming languages is its declarative nature. This means that there is no algorithm that solves the problem. Instead, there is a description of a problem, written in a special manner (according the rules of the Prolog language), so the system can deduce the solving of that problem [14-15].

The source code of Prolog program consists of logical formulas describing the properties of the problem. Prolog uses the method of resolution in order to determine the right answer to the problem given by the description. Programming in Prolog consists in expressing the relationship as opposed to functional programming, where the expressions (functions) are evaluated. It should be kept in mind that every function is a relation, but not vice versa.

Coding in the Prolog involves creating the program structure, which consists of predicates and clauses. The desired solution is called goal. The final appearance of the source code largely depends on the type of dialect. The Borland's Turbo Prolog requires exact definition of program structure as it is shown in Fig. 7, while some of the others (e.g. SWI Prolog) are more tolerant.

The source code of the Prolog program looks very different in comparison with source codes written in other programming languages. First of all, there is no "data flow", no sequential execution of commands, no iterations. The main body of the source code is built on so called clauses, which could consist of facts and rules.

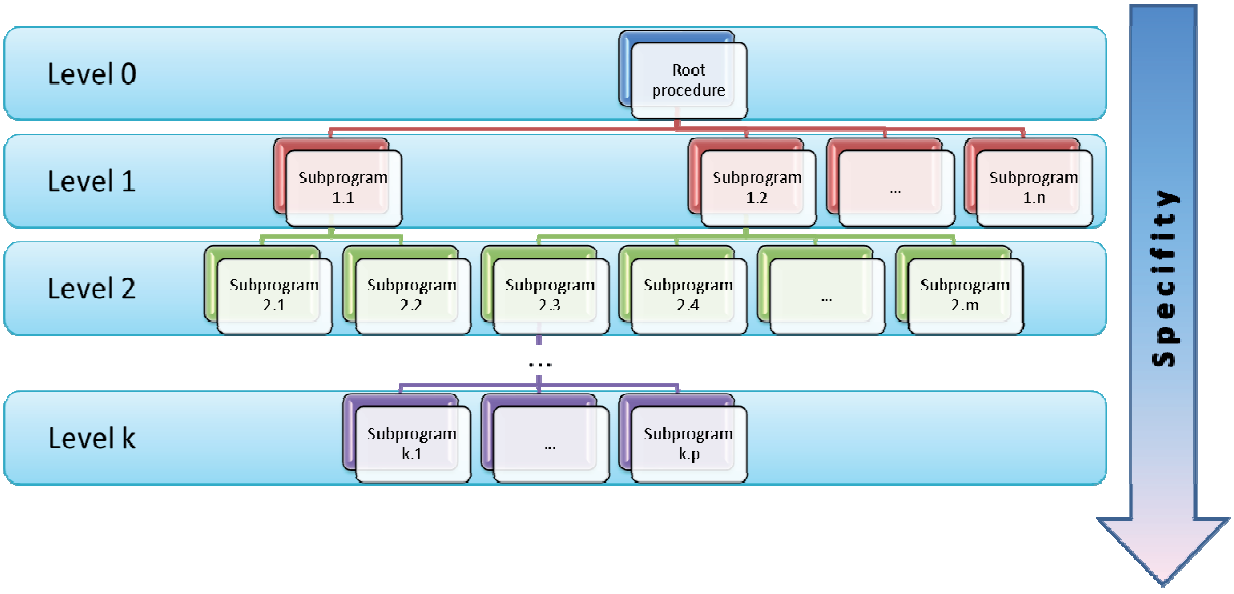

Fig. 6. The tree structure of a program. Different levels represent different quantity of details

```
domains
    /* domain statements */

predicates
    /* predicate statements */

goals
    /* goal_1, goal_2... etc. */

clauses
    /* rules and facts */
```

Fig. 7. The structure of the source code in the Turbo Prolog language

In order to fully understand what the programming in Prolog is, first of all we should answer to the question: what is the predicate, clause, fact and rule. The predicate could be seen as a declaration of the object and its properties, for example

car (brand, color, age). (4)

defines a hypothetical object, which is a car of some brand, some colour and age. The expression (4) could be used to determine a concrete car – then it could have form of (5):

car (fiat, blue, 3).
car (ford, yellow, 5). (5)
car (toyota, white, 2).

The expressions listed above, describes the concrete cars. In this way the predicate has been treated as a template, and on this basis the specific vehicles and their properties have been defined. The expressions (5) are called facts and are included in the set of clauses.

The other subset of clauses is formed by rules. Let's assume that car painted in bright colour is better visible at night. From the clauses mentioned in (5), we could select the yellow and white colour and write the following rules (6):

visible (X) :- **car** (X, yellow). (6)
visible (X) :- **car** (X, white).

Gathering the (5) and (6) together and running the code in SWI-Prolog gives the results shown in Fig. 8. The subsequent numbers indicate questions (goals), while "?-" is the standard prompt of the SWI-Prolog system.

```
11 ?- visible(fiat).
false.

12 ?- visible(ford).
true .

13 ?- visible(toyota).
true.
```

Fig. 8. The results of "asking questions" in the SWI-Prolog

The above example illustrates the general principle of creating a program in Prolog. Despite some similarities to the description of the algorithm by using pseudocode, the source code written in Prolog does not describe the subsequent steps that lead to completion of a specific task.

5. The logic programming as a support in planning of robotic task

As it was mentioned earlier, the Prolog language is characterized by a declarative form of source code, as a natural way of the creation of the program in that language. A certain level "freedom" when defining the constants and variables, no need to preserve the characteristic structure of the functional programming (associated with the order of coede execution) and the form of source code that is similar to natural language, make the Prolog a good tool for solving problems related to the robotic task planning [16].

5.1. The simple example

Let's assume that there is a simple robotic assembly cell (Fig. 9). It consists of a robot, conveyors as input and output buffers, temporary buffer, and the assembly holder. The robot performs the assembly operation of a car lamp. There are four characteristic position of the robotic arm, which are indicated in Fig. 9 by the numbers 1, 2, 3 and 4. In the input buffer the three components are arranged in random order – they are the mirror with the housing (body), bulb and glass. The task of the robot is to assemble the lamp by placing all of the components in the correct order, starting with the body, which should be placed in the holder, and then installing the bulb and glass. The ready lamp should be placed in output buffer. Because the arrangement of components in the input buffer may be random, there is another buffer, where any component could be stored for a while. It has the capacity of one element. We assume that there could be any combination of components, but there must be possibility to assemble the lamp from the following three parts taken from the conveyor.

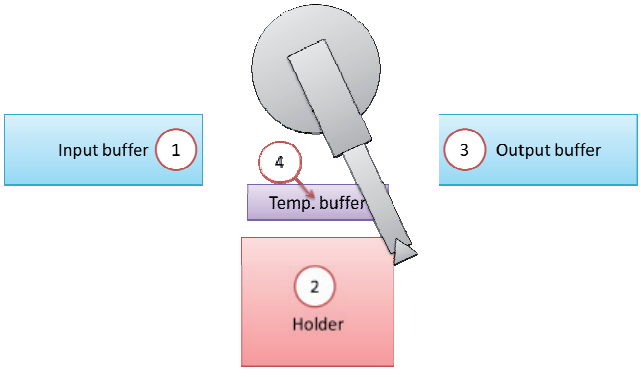

Fig. 9. The model of hypothetic assembly cell

The aim is to create a simple program, which describes the actions on the task level. In order to do that it is necessary to create the "*input_buffer*" predicate (7):

```
input_buffer (part1, part2, part3).    (7)
```

where the *part1*, *part2* and *part3* are components used to build a lamp. In this manner there could be six combination of the element sequence in the input buffer (8-13).

```
input_buffer (bulb, glass, body).      (8)
input_buffer (bulb, body, glass).      (9)
input_buffer (glass, bulb, body).      (10)
input_buffer (glass, body, bulb).      (11)
input_buffer (body, bulb, glass).      (12)
input_buffer (body, glass, bulb).      (13)
```

The goal is to find the sequence of robot's arm movements according to the configuration of the input buffer. The assembly process should fulfil the following rules:

- the body should be placed in holder,
- the bulb should be mount inside the body,
- the glass should cover the body with the mounted bulb,
- the temporary buffer is intended for short-time storage of only one part.

The search for solution is initiated by entering the goal in the form of (8-13). For example the goal (12) generates the following solution:

```
Move body from in_buffer to holder
Move bulb from in_buffer to holder     (14)
Move glass from in_buffer to holder
Move lamp from holder to out_buffer
```

The "*holder*" statement stands for arm position – this is a simplification, which unambiguously determines the position for further processing at higher level of detail. As it can be seen, the lamp could be assembled without the use of temporary buffer, because all of the components are in order of use. In turn, the goal (9) generates the following solution:

```
Move bulb from in_buffer to temp_buffer
Move body from in_buffer to holder
Move bulb from temp_buffer to holder   (15)
Move glass from in_buffer to holder
Move lamp from holder to out_buffer
```

Here the temporary buffer is used, because it is need to change the order of components. The goals (8) and (10) cannot be solved because of limited capacity of temporary buffer.

6. Conclusions

For centuries the man is accompanied by dreams of machines that he can control by giving the commands in natural language. With the development of robotics, these dreams have become a necessity, especially in applications where the man is accompanied by a robot during his work, like for example heart surgery or space mission. The development of task-level programming is currently in the early stage, but gives a chance to clearly define the top level actions that may be further "clarified" by developing the subsequent levels of the code. The restrictions associated with the particular task can cause problems even at the highest level of generality. This paper has shown the possible use of logic programming language as a method of supporting the task planning for robotic systems. At the present stage of development, the presented method cannot be considered as independent and competent tool, but in some cases may significantly facilitate the programmer's work.

References

[1] D. Reclik, G. Kost, J. Świder, The signal connections in robot integrated manufacturing systems, Journal of Achievements in Materials and Manufacturing Engineering 26/1 (2008) 89-96.

[2] K. Foit, J. Świder, The project of a platform-independent, off-line programming system for industrial robots, Proceedings of the 7th International Scientific Conference "Computer Integrated Manufacturing – Intelligent Manufacturing Systems", CIM'2005, 62-65.

[3] J. Świder, K. Foit, G. Wszołek, D. Mastrowski, The off-line programming and simulation software for the Mitsubishi Movemaster RV-M1 robot, Journal of Achievements in Materials and Manufacturing Engineering 20/1-2 (2007) 499-502.

[4] K. Foit, The web-based programming interface for the Mitsubishi Movemaster robot, Journal of Achievements in Materials and Manufacturing Engineering 27/2 (2008) 183-186.

[5] G. Kost, R. Zdanowicz, Modeling of manufacturing systems and robot motions, Journal of Materials Processing Technology 164-165 (2005) 1369-1378.

[6] K. Foit, Mixed reality as a tool supporting programming of the robot, Advanced Materials Research (2014) (in print).

[7] R. Johansson, Sensor integration in task-level programming and industrial robotic task execution control, Industrial Robot: An International Journal 31/3 (2004) 284-296.

[8] T. Lozano-Pérez, Task-level planning of pick-and-place robot motions, IEEE Computer 22/3(1989) 21-29.

[9] E. Coste-Mainere, B. Espiau, E. Rutten, A task-level robot programming language and its reactive execution, Proceedings of the IEEE International Conference "Robotics and Automation" IEEE, 1992, 2751-2756.

[10] C.C. Kuhlthau, From Information to Meaning: Confronting Challenges of the Twenty-first Century, Libri 58 (2008) 66-73.

[11] C. Menant, Information and Meaning, Entropy 5 (2003) 193-204.

[12] L. Floridi, Is semantic information meaningful data?, Philosophy and Phenomenological Research 70/2 (2005) 351-370.

[13] J.C. Mingers, Information and meaning: foundations for an intersubjective account, Information Systems Journal 5/4 (1995) 285-306.

[14] Turbo Prolog – Owner's handbook, Borland International Inc., 1986.

[15] U. Nilsson, J. Maluszynski, Logic, Programming and Prolog, John Wiley and Sons Ltd, 1995.

[16] B.G. Batchelor, R. Hack, Robot vision system programmed in Prolog, Proceedings of the Conference "Machine Vision Applications, Architectures and Systems IV", Philadelphia, 1995, 239-252.