A GOTO-based Concept for Intuitive Robot Programming

Katharina Barth and Dominik Henrich

***Abstract*— This paper proposes a new concept to augment direct robot programming with sensor-based branching and looping. The objective is to derive a new robot programming paradigm in order to enable non-programmers to use robots for different tasks. Inspired by the ancient programming style based on the GOTO-statement, this approach requires the user to deal with only two basic concepts: Intuitive Control Expressions and Spatial Labels. This simplicity yields a very intuitive robot programming interface. We introduce these new concepts and derive some statements about the power and the limitations of this approach.**

I. Introduction

CONTEMPORARY robot manipulators are typically employed in large scale industry, where they are programmed by dedicated robot experts now and then. To make reasonable use of robots in circumstances where there are no programmers available and reprogramming is needed often, it is necessary to develop really intuitive programming interfaces that do not require much more than common sense. From this demand, some **design goals** for such a language can be derived:

- The programming paradigm should be 100% direct, i.e. all programming action is conducted directly on the robot (e.g. by moving it or by activating sensors or actuators) and not on any external programming system. The user should regard the robot as an intelligent tool.
- There should be no textual programming and no visual programming involved, that could draw off the user's attention from his main businesses. By "visual programming" we understand systems where the flow of control is defined by graphical elements that are arranged and/or connected by the user. Famous examples comprise the visual programming language NXT-G for Lego Mindstorms and the LabVIEW programming language.
- There should be as little extra hardware needed as possible since the process of robot programming should take place in between other activities. A carpenter, for example, who wants to use a robot in his production, might find it weird and tedious to put on a head-mounted display for programming his robot.
- The language should be based on very few basic principles since every basic concept a user has to know and remember tends to make the programming process less intuitive.

K. Barth is with University of Bayreuth (e-mail: katharina.barth@ uni-bayreuth.de).

D. Henrich is with University of Bayreuth (e-mail: dominik.henrich@uni-bayreuth.de).

Considering the last point, we introduce two basic principles: Spatial Program Labels (**SPL**s) and Intuitive Control Expressions (**ICE**s). The former are derived from the GOTO statement, the latter from conditional branching. The main thesis of this paper is that the idea of programming with "GOTO" is not just not harmful but even useful in the context of intuitive robot programming.

A coarse overview of the targeted system is depicted in Fig. 1. It comprises a sensor-equipped robot, a simulated robot (in virtual reality) and the demonstration observer (a software running on a computer). The user moves the robot (in gravitation compensation mode) while activating sensors or buttons. This demonstration is recorded and analyzed online and a corresponding robot program is generated. The simulation of the robot is an important part of the user interface — it shows active and activatable components (e.g. spatial labels), the movement paths that were already programmed and other useful feedback. It also can visualize partial program executions before their execution on the real robot and thereby enhance safeness.

If there is at least one branch for which the corresponding actions are not yet specified and a SPL is snatched, the unfinished program is partly executed in order to take the robot to a part of the program, where the programming process can be continued.

The paper is organized as follows. Section 2 shows the most important related work with regard to simplifying robot programming. In Section 3, we explain our motivation to base our intuitive approach on the GOTO language. The resulting concepts are presented in Section 4, followed by a description of the program generation in Section 5. In Section 6, the power and limitations of the approach are examined. Finally, Section 7 summarizes the conclusions.

II. Related work

Different approaches for the simplification of robot programming have been examined. Visual programming languages are applied in the context of robot programming (e.g. the Microsoft Visual Programming Language [5]), because they are expected to be superior to textual programming languages in terms of accessibility for non-programmers. Though, this programming method contradicts the first two design objectives from Section 1. A discussion about the assumed superiority of visual programming languages over textual languages can be found in [6]. While

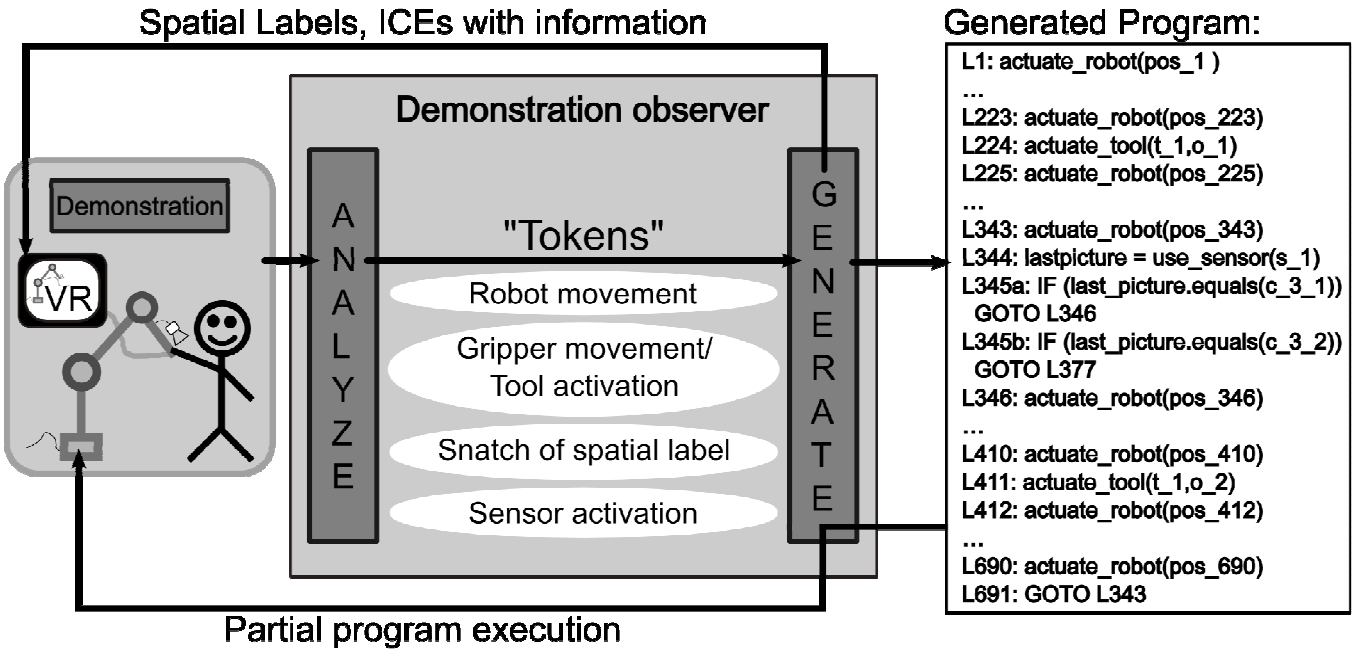

Fig. 1. System Overview: Intuitive robot programming system

text is aligned in a one-dimensional manner, the two-dimensionality of visual languages is considered to be more natural. The proposed concept goes one step further: the programming takes place directly in the three-dimensional working space of the robot.

Another approach, which is absolutely in line with those design goals, is called Robot Programming by Demonstration (RPD). A survey about its different variations can be found in [2]. With RPD, the user demonstrates a task several times and the robot generalizes those (different) demonstrations. In [3], for example, the task of pouring a glass of water into a bowl is represented by a Gaussian Mixture Model, and Gaussian Mixture Regression is used to execute the task. The demonstrations vary in the glass's and the bowl's start position and with a good set of demonstrations, the system can generalize to situations with different start positions that have not been demonstrated before.

In RPD, Policy Learning is used, i.e. the programming results in a mapping from observation states to actions [2]. Since the same state always causes the same action, this approach does not allow encoding the temporal ordering of the task. Usually, there is no "memory", and approaches to extend Policy Learning with states are task specific ([2], p. 480).

The foundation of the new robot programming paradigm is the well known playback method. While the user moves the robot by direct physical contact, the positions are recorded in small time steps. When the program is executed, the robot moves from one stored position to the next [9]. This method has already been used in early robot applications, especially for spray painting. Modern robots that are equipped with force/torque sensors alleviate the formerly physically demanding procedure. Although this method is easy to understand, it lacks mechanisms for loops and conditionals. It is not possible to make use of extern sensors without applying an additional programming interface, typically textual programming. In the next section, the basic approach of extending playback programming will be motivated and in Section 4 the derived mechanisms will be described.

III. Derivation of a Robot Programming Language

Basically there are two alternatives for designing an intuitive direct robot programming system: to design it from scratch or to derive it from an existing robot programming language. In this paper we choose the second alternative because the generated program should be easily transferable to a program that is executable on a real robot.

Typical robot programming languages mainly consist of movement instructions, control structures, signal processing, digital and analogue signals and so forth [1]. The most obvious way to derive an intuitive robot programming language from a textual robot programming language is to find an intuitive match for every concept in the original language. This approach shows some disadvantages: First of all, this will yield a set of concepts that have to be learned by users of the system. Since many concepts in the original language are redundant, this set might result larger than needed. (Modern programming languages contain many structures that do not extend the power of the language but rather make it more convenient and more readable. This is called syntactic sugar.) Furthermore, programming such a system would presumably be even more complicated than textual programming since the only difference would be an abstraction layer that transforms movements and button

presses into a robot program written in the original language.

In this section we present the derivation of an intuitive robot programming language from the theoretical "GOTO language". The latter is a simple programming language that is investigated in computability theory [8]. It was shown to have the same computability like Turing machines. However, since this notion of computability refers to functions and since the computation of functions is very far from the main focus of robot programming, there is little practical use of this Turing completeness (This holds particularly true, due to our approach's poor transfer of the concept of variables, as will be shown.). However this approach has the advantage that we can show that all kinds of nested conditional loops and branches can theoretically be programmed. The definition of the GOTO language is shown in Definition 1, which originates from [8].

Definition 1. A GOTO program is a sequence of pairs consisting of labels L_i and statements S_i:

$$L_1 : S_1; L_2 : S_2; \ldots ; L_k : S_k;$$

The following statements are allowed with x_i being variables and c being a constant:

Unconditional branch:	GOTO L_i
Conditional branch:	IF x_i = c THEN GOTO L_j
Halt instruction:	HALT
Value assignment:	$x_i := x_j \pm c$

In the early times of computer programming, control structures like loops were implemented with the means of the GOTO statement. Later it fell into disrepute since using it imprudently results in code that is difficult to read and maintain (This insight was made popular by Edsger W. Dijkstra in his famous letter "Go To Statement Considered Harmful", see [4].). For this reason it was replaced by more sophisticated control structures, e.g. different types of loops. Our rationale to do this seemingly step backwards is the intuitiveness of the GOTO statement in the programming process. Its use resembles the linear way people think. To make this point clearer we have a look at another small theoretical language – the WHILE language from Definition 2, also coming from [8].

Definition 2. The syntax of the WHILE programming language is inductively defined as follows (with x_i being variables and c being a constant):

Every value assignment in the form

$$x_i := x_j + c \text{ or } x_i := x_j - c$$

is a WHILE-Program. If P_1 and P_2 are WHILE programs, so is the sequence

$$P_1; P_2.$$

If P is a WHILE program, so is

$$\text{WHILE } x_i := 0 \text{ DO P END.}$$

This language resembles much more a modern programming language than the GOTO language does, and it has the same computability. Its disadvantage from the perspective of intuitive robot programming is its recursive nature. In the context of direct robot programming this would mean that the programmer either has to be aware of the different levels of recursion (if he starts the process from the beginning) or he has to build up the program by combining smaller programs.

Both variants have weaknesses regarding their intuitiveness: In the first case, it is important to make explicit when the body of a control structure ends. Experience from our own work on spatial counter-controlled loops suggests that forgetting to set a boundary while demonstrating is a frequently occurring mistake [10]. Whereas in textual programming this does not cause trouble, since inserting a bracket or a key word is done easily afterwards, the issue is completely different in direct robot programming.

In the second case, composing a program from smaller programs means that the order of programming differs significantly from the natural order of the task. This yields not only a lack of intuitiveness, but also a very practical problem: programming a task e.g. where an object is machined might require the object to be in a certain processing condition for a subroutine of the program. The object could be processed manually before starting to program but the result might differ from the result of the part of the program that still has to be programmed.

IV. Intuitive Mechanisms for Control of the Program Flow

In this section, we assign an intuitive mechanism to each of the four statement types from Definition 1. Actually, an unconditional branch is not necessary for the GOTO language to be Turing-complete; the other three statements would be sufficient. We decided to include unconditional branching since the concept of variables is not transferred very well to the intuitive paradigm and we consider the "simulation" of an unconditional branch by a variable with value 'true' in a conditional branch not as intuitive. In our approach, from the view of the user the concept of GOTO is strictly separated from the concept of conditional branching.

A. Spatial Program Labels

In a textual program, labels mark those lines to which the program execution can jump. There are two variants of labels: ascending numbers in every line or the explicit definition of labels wherever necessary. We restrict ourselves to the former variant since explicit labels would require the user to prospectively identify positions that will be jumped to in the subsequent demonstration. Such a mechanism would be error-prone and annoying. The first time the user thinks about a Spatial Program Label (SPL) should be the moment, when he actually wants to do the "jump". Thereby locality is fostered, a desired characteristic of novice programming systems meaning that related program components are kept together [6], but here it is locality in matters of time.

While the user moves the robot, the positions are recorded and corresponding SPLs are created. At the same time, the

current robot position (in Cartesian space) is compared to all previous SPLs. If the current position is very close to a SPL and some other predefined consistency conditions are satisfied, e.g. the gripper having the same opening or the robot's tool having the same velocity, the SPL is highlighted in the simulation and can be activated by pressing a button. In most cases this will result in the robot executing parts of the program until it encounters an ICE.

The predefined consistency conditions narrow down the activatable SPLs so that the user is not confronted with too many possibilities. Another important effect of the consistency conditions is that ambiguities are resolved by them. Without them, inconsistent situations may occur, for example, if the gripper is open at the moment of the SPL's definition and closed at the moment, the SPL is activated, this might mean that the gripper is opened in an extremely fast movement or the opening of the gripper extends to parts of the program around the jump. By the premise of the same velocity sudden changes in the velocity of the resulting program are avoided. Consistency conditions force the user to specify how the last state of the robot previous to the jump can be transformed to the state at the moment, the label was created.

B. Intuitive Control Expressions

As we have seen in the previous section, Spatial Program Labels augment playback programming by looping. Intuitive Control Expressions (ICEs) add conditional branching. They are needed, if the program execution depends on environmental conditions that have to be determined by the use of sensors. As a simple example, we consider in the following the case of sensing the colour in the middle of a camera picture. The basic principle should be transferable to other types of sensors and to other image recognition methods.

For ICEs we allow more than two cases. In the GOTO language this effect can be achieved by nesting conditional branches into each other.

Basically, the user needs to specify the position in the program where the picture should be taken, the different cases of the control expression, and how the execution should be continued for each case. The user can specify the position by pressing the camera's button at the corresponding position (regarding time and place) in the demonstration. The picture is stored (and given a variable name for referencing it at execution time) and the subsequent actions are considered to form the actions for the first case. Later, further cases are specified in a similar way: the picture pertaining to the next case is taken and then the actions are demonstrated with the robot. In contrast to the first time a picture is taken, for all other pictures the robot moves to the corresponding position automatically after a SPL was activated. The user must notify the system of the last time the ICE will be demonstrated. All ICEs where at least one case has not yet been defined are considered unfinished.

When the program is executed and reaches the position of the ICE, a picture will be taken and compared to all pictures of the ICE. The continuation of the program is that one, whose picture is closest to the currently taken picture, provided that the distance does not exceed a predefined value.

C. Program exit

Not all kinds of robot programs are supposed to run in an infinite loop. In some cases it is desired to stop the robot if the task has been accomplished or if certain conditions hold. For this purpose there is a special built-in SPL. Its position is the start position (with the gripper closed).

To program the execution stop, the robot is moved to the start position and the special SPL is activated. When there are unfinished ICEs left (see Subsection B), the robot will start the execution again and move to the position of the next unfinished ICE.

D. Variables and value assignments

The last statement type from Definition 1 we need to transfer is also the most difficult: value assignment to a variable. A program's variables can be distinguished into two groups: implicit and explicit variables. Here, we consider those variables that are not directly changeable by the user as implicit variables. Implicit variables require dedicated mechanisms for their definition in the intuitive programming paradigm. We do not want to deal with implicit variables in this paper, but an approach for counting loops was examined in [10]. As explicit variables, on the other hand, we consider states of the environment that are sensed within the ICEs. We also want to avoid most explicit variables and value assignments in our concept. The only occasion where the user has contact with a kind of variable is when he uses ICEs. By defining a new ICE, a piece of code for a new variable is generated.

Theoretically, variables could be simulated in the real environment by placing or manipulating objects and using ICEs to determine their states (e.g. by incrementing a manual, mechanical counter and observing its display for a certain number). Of course, this method is only practicable for very small examples. We do not consider the lack of a good transfer of variables to the intuitive programming paradigm as a severe restriction since in the context of robot programming the explicit computation has much less importance than in standard programming.

V. Program Generation

This section details how the system works. A coarse overview was already given in Figure 1. The finite-state automaton in Figure 2 shows how the different inputs of the demonstration (called "Tokens" in Figure 1) change the internal state of the system and how the program code is generated.

Generally, when the robot is neither being programmed

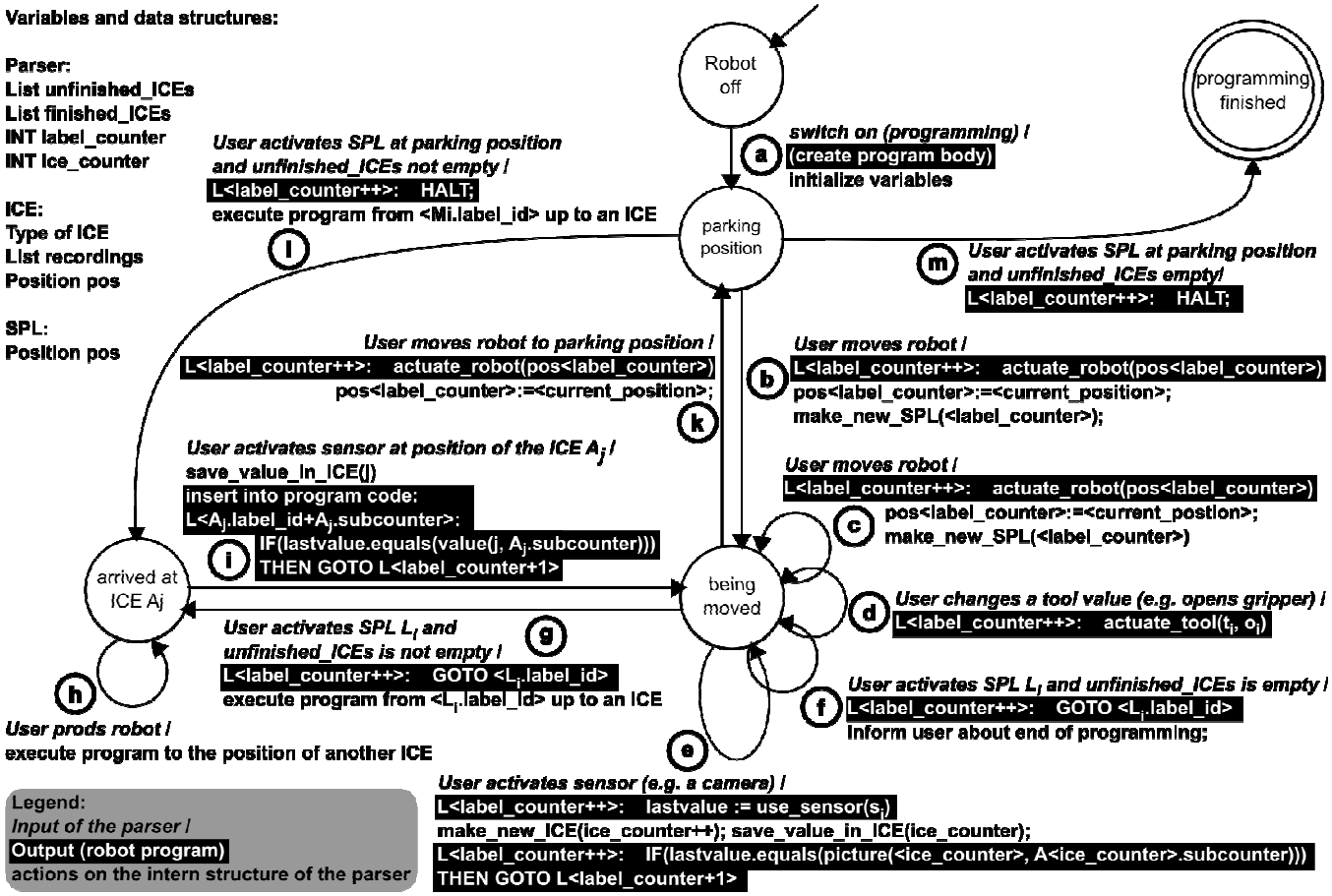

Fig. 2 FSA for the generation of the program (Intuitive Language); the most important datastructures are listet on the left side.

nor executing a program, it is in its parking position (cf. Subsection 4.3). First of all, the user switches the robot in programming mode. Hereon the program body is created (if necessary) and variables for the program generation are initialized (a, cf. Figure 2). Now the user can grab the robot and move it (b and c). While doing so, the system generates code lines that consist of a label and a movement command. In addition, the current position of the robot is recorded, so that it can be approached when the program is executed, and a new SPL is introduced. When the user changes the tool value, e.g. the gripper's opening, this action is recorded in the program (d). A new ICE can be defined at the current position by pressing the camera's (or another sensor's) button (e). A picture is stored that will serve as reference sample in the execution and the program is supplemented with code for taking and comparing a picture (or other sensor data). Also, the ICE is put into the list of unfinished ICEs.

When a SPL is activated, it depends on the list of unfinished ICEs, how the robot will react. If it is empty (f), the user is asked to move the robot to the parking position to finish programming. Although this motion will be recorded like all the other motions, this part of the program will never be executed. It can be regarded as dead code. If the list is not empty (g), the generated code is the same, but the program is executed onwards from the snatched label until it encounters an unfinished ICE. In contrast, if this is not the ICE where the user wants to continue programming, he can prod the robot to make him move on to another unfinished ICE (h). (Please note, that the order of the ICEs is not specified here.) When the desired ICE is reached, the next branch can be inserted by activating the camera (i). At this point the user has to decide whether this is the last picture (and case) of this ICE and if so, the ICE will be transferred from the list of unfinished ICEs to the list of finished ICEs. This transition's code generation deviates a bit from the code generation of all other transitions since the code is not attached to the end of the program but it is inserted after the position where the code was inserted when the ICE was newly defined (i.e. with transition e). After the picture is taken, the demonstration is continued. At this point, it is essential that the function equals() always returns true for at most one stored sensor value of the ICE. Otherwise, the programming order of the different cases would influence the execution of the program.

When the robot is moved to the parking position (k) a special SPL becomes available. When this SPL is activated and there are still unfinished ICEs left, a HALT-command is

generated. Then the robot executes the program, beginning at the first line until it reaches one of the unfinished ICEs (l). However, if the list of unfinished ICEs is empty, the programming procedure is finished (m).

VI. Power and Limitations of the Intuitive Approach

In this section we compare our approach to intuitive robot programming with a fictitious textual robot programming language based on the GOTO language from Definition 1. A direct comparison with the GOTO language would neglect characteristics that are specific for robot programming, in particular movement. Definition 3 shows the modified version. The main difference lies in the specialization of the variables' types and the addition of movement commands. It purposely resembles the output of the finite state automaton in Figure 2 to simplify the comparison. The most obvious difference between general textual (robot) programming languages and the intuitive approach arises from the relinquishment of explicit variables: no explicit computation can be performed.

Definition 3. A GOTO robot program is a sequence of pairs consisting of labels L_i and statements S_i:

$$L_1 : S_1; L_2 : S_2; \ldots ; L_k : S_k;$$

The following statements are allowed with x_i being variables for sensor measurements (i.e. they are initialized and evaluated when the program is executed), c_i being constant sensor measurement values (i.e. they are determined when the robot is programmed), s_i denoting the attached sensors, p_i being constants for robot movements (e.g. joint angles) and o_i being constants for the tool states (e.g. the gripper's opening):

Unconditional branch:	GOTO L_i
Conditional branch:	IF x_i.equals(c_j) THEN GOTO L_k
Halt instruction:	HALT
Sensor value assignment:	x_i :=use_sensor(s_i)
Movement assignment:	actuate_robot(p_i) | actuate_tool(o_i)

We also demand the last statement of the program to be either the HALT command or an unconditional branch.

The language that is defined through the FSA from Fig. 2 (henceforth called Intuitive Language) is obviously a subset of the GOTO robot programming language from Definition 3 (henceforth called Textual Language): every output of the FSA has a counterpart in Definition 3. On the other hand, there are programs that can be generated by the grammar in Definition 3, but not by the FSA. For the remainder of this section we investigate, which programs cannot be generated, and whether this is a disadvantage (since it restricts the power of the approach) or an advantage (since it restricts the generable programs to reasonable programs).

A. Evaluation of ICEs

The Textual Language permits the formulation of a program where the evaluation of an ICE does not follow directly after the corresponding use of the sensor:

```
...
//use sensor(s23)
L100: x42 := use_sensor(s23)
//do other things
L110: x43 := use_sensor(s13)
L120: actuate_robot(p120)
//evaluate sensor variable
L130: IF x42.equals(c10) THEN GOTO L220
L140: IF x42.equals(c11) THEN GOTO L320
...
L220: HALT
...
L320: actuate_tool(o22)
L330: HALT
```

In contrast, the Intuitive Language only allows the evaluation of an ICE directly after the sensor is activated. This does not imply a restriction to the power of the language, but it is necessary to duplicate the part of the code between the use of the sensor and the evaluation. In the intuitive paradigm this means duplication of demonstrations. The upper code can be transformed to the following, executionally equivalent code:

```
...
L100: x42 := use_sensor(s23)
L110: IF x42.equals(c10) THEN GOTO L220
L120: IF x42.equals(c11) THEN GOTO L320
...
L220: x43 := use_sensor(s13)
L230: actuate_robot(p120)
L240: HALT
...
L320: x43 := use_sensor(s13)
L330: actuate_robot(p120)
L340: actuate_tool(o22)
L350: HALT
```

B. Forward and Backward Jumps

In contrast to the Textual Language, only backward jumps can be programmed explicitly in the Intuitive Language (f and g in Fig. 2), since the Spatial Label has to be defined before it can be activated. Forward jumps are defined implicitly with every ICE (e and i). Thus, all unconditional branching produces backward jumps and all conditional branching produces forward jumps, whereas the Textual Language allows both kinds of jumps for both kinds of branches. This does not imply a limitation of the new approach's power: backward jumps can be added to conditional branching by simply adding an unconditional branch (and thereby a backward jump) at the very end of the program (via transitions e/i and f/g). When we express the desired behaviour with a backward jump as a textual program, this would look like the following:

```
...
L100: ...
...
L150: IF x42.equals(c23) THEN GOTO L100
```

We can simulate this program with only conditional forward jumps and unconditional backward jumps:

```
…
L100: ...
...
L150: IF x42.equals(c23) THEN GOTO L999
...
L999: GOTO L100
```

Since we demand a valid program to end with HALT or an unconditional branch, the extensions at the program's end do not change the program's behaviour. For the other way round, a dummy ICE could be used, that always evaluates to the same value, but this probably would contradict intuitiveness. For this reason, we show with a constructive proof, that every unconditional forward jump can be avoided by a very natural permutation of the code fragments. The following remarks are not important for the implementation or the understanding of the approach, but they show that there is no limitation of the language's power because of the limitations that were discussed at the subsection's beginning.

At first, we divide the code into jumpless sections that are rearranged in the following step. Considering the sequence of pairs of labels Li and statements Si from Definition 3,

$$L1 : S1;L2 : S2; \ldots ;Lk : Sk;$$

we identify those indices where the cuts have to be made:

Every GOTO statement (regardlessly of its conditionality) is the last statement of a section and every label that is referred to by a GOTO statement is the first label of a section. This step can be expressed in pseudo code:

```
INPUT: list L of label-statement pairs representing the
general program
OUTPUT: list T of lists of label-statement pairs representing
the jumpless sections
// break L into jumpless sections
FOR i = 1 TO k:
   IF (L_i is referred by any GOTO statement) {
      T.add(list of pairs in L before (L_i,S_i))
      L.remove(list of pairs in L before (L_i,S_i))
   }
   IF (S_i is a GOTO statement) {
      T.add(list of pairs in L up to (L_i,S_i))
      L.remove(list of pairs in L up to (L_i,S_i))
   }
END
```

Like the original program, the rearranged program starts with L1 : S1 and the related section. The next section to be taken from the temporary list T and added to the output program N is determined as follows:

```
INPUT: list T of lists of label-statement pairs representing
the jumpless sections
OUTPUT: list N of label-statement pairs representing the
transformed program
// rearrange sections
programEnd // collection of last program parts
lastSection := T.pop(0) // start with first section
WHILE (T.notEmpty()) {
   N.add(lastSection)
   lastStatemnt := lastSection.lastStatement()
   IF (lastStatemnt is not a branch) {
      oldFollower := lastSection.follower()
      IF (T.contains(oldFollower)) {
         lastSection := oldFollower
      }
      ELSE { // old Follower already in N
         jmpLbl := oldFollower.firstLabel
         N.add((newLabel, GOTO jmpLbl))
         lastSection := T.pop(0)
      }
   }
   ELIF (lastStatemnt is a conditional branch) {
      oldFollower := lastSection.follower()
      jumpSection := lastStatemnt.gotoSection()
      // destination already used
      IF (N.contains(jumpSection)) {
         jmpLbl := jumpSection.firstLabel
         tmpLbl := createNewLabel()
         lastStatemnt.changeDestination(tmpLbl)
         programEnd.add((tmpLbl, GOTO jmpLbl))
      }
      IF (T.contains(oldFollower)) {
         lastSection := oldFollower
      }
      ELSE { // old Follower already in N
         jmpLbl := oldFollower.firstLabel
         N.add((newLabel, GOTO jmpLbl))
         lastSection := T.pop(0)
      }
   }
   ELSE { // lastStatemnt unconditional branching
      jumpSection := lastStatemnt.gotoSection()
      IF (N.contains(jumpSection)) {
      lastSection := T.pop(0)
      }
      ELSE {
         delete lastLabelStatementPair()
         lastSection := jumpSection
      }
   }
}
N.add(lastSection)
N.add(programEnd)
```

Basically, there are three possibilities for the last statement of the last section: it can be a conditional branch, an unconditional branch, or another statement. For all cases we now describe how the next section has to be chosen so that there are only unconditional backward and conditional forward jumps left and the order in which the statements are executed does not change.

If the last statement of the current section is **not a GOTO** and the following section in the original program (delivered by the function follower() in 4) has not yet been inserted into

the new program, insert it. Otherwise (if the section is addressed from more than one GOTOs), attach an unconditional GOTO to the end of the program that directs the control flow to the corresponding section. Since the section is already part of the new program, this results in an unconditional backward jump. The next section to insert is the one from the left sections that has the smallest label number (the first in list T).

The most complicated case is **conditional branching**. The reason for this is that there are two possible next sections and both can still be in T or already in N. We call the section that is executed, when the condition is fulfilled, the jumpSection. If this section is still left, there is no need to do anything about it, but otherwise, the procedure to resolve conditional backward jumps, that was introduced previously, has to be used. Regarding the oldFollower, the same actions have to be taken like when the lastStatemnt is not a branch.

If the last statement of the current section is an **unconditional GOTO** and the jumpSection is already inserted into the new program, this is a backward jump, like it is required. Otherwise, the GOTO statement is deleted and the jumpSection is inserted directly behind the current section, thereby avoiding the GOTO completely.

At the end, the labels have to be changed to make sure, that they are increasing over the program.

One can see that the lack of conditional backward jumps and unconditional forward jumps in the intuitive paradigm does not imply any limitation in principle, since every GOTO robot program can be transformed to an equivalent one that obeys these restrictions.

VII. Conclusions

In this paper, we tried to take the first steps toward the extension of playback robot programming by sensor-based branching and looping. The rationale for orienting ourselves towards the old-fashioned GOTO statement is its purity resulting in only two basic programming concepts, which the user has to be aware about: ICEs as the intuitive counterpart of conditional GOTOs and SPLs for unconditional GOTOs. We showed how the program can be generated depending on the programmer's actions. We also showed that the power of the approach is not limited in comparison with a fictitious textual robot programming language that was deduced from the GOTO language. The findings suggest that GOTO is indeed useful in the context of intuitive robot programming.

There still are some issues that need to be cleared, before the concept can be implemented. For example, the mechanism to choose the next unfinished ICE that should be moved to, is not yet specified. Another interesting question is whether and how intuitive counting loops (cf. [10]) could be added to the concept in a natural way.

References

[1] A. M. Al-Qasimi, M. Akyurt, and F. Dehlawi, “On robot programming and languages,” in *Proc. 4th Saudi Engineering Conference*, vol. 4, pp. 249-258, Nov. 1995.

[2] B. D. Argall, S. Chernova, M.Veloso, and B. Browning, “A survey of robot learning from demonstration” in *Robotics and Autonomous Systems*, 57:469-483, 2009.

[3] S. Calinon and A. Billard, “A probabilistic programming by demonstrationframework handling constraints in joint space and task space,” in *2008 IEEE/RSJ International Conference on Inelligent Robots and Systems*, pp. 367-372. IEEE, 2008.

[4] E. W. Dijkstra, “Go to statement considered harmful,” in *Communications of the ACM*, 11(3):147-148, 1968.

[5] Microsoft. Visual programming language. http://msdn.microsoft.com/en-us/library/bb483088.

[6] J. F. Pane and B. A. Myers. “Usability issues in the design of novice programming systems,” *Technical Report CMU-CS-96-132, School of Computer Science*, Carnegie Mellon University, 1996.

[7] L. Ramshaw. Eliminating go to’s while preserving program structure. Journal of the Association for Computing Machinery, 35(4):893-920, 1988.

[8] U. Schöning. Theoretische Informatik – kurzgefasst (in german). Spektrum Akademischer Verlag, Berlin, 1997.

[9] R. D. Schraft and C. Meyer. “The need for an intuitive teahing method for small and medium enterprises” in *IRS 2006 – ROBOTIK 2006: Proceedings of the Joint Conference on Robotics*, 2006.

[10] K. Soller and d. Henrich. „Intuitive robot programming of spatial control loops with linear movements, “ in T. Kröger and F. M. Wahl, editors, Advances in Robotics Research, pp. 147-158. Springer Berlin Heidelberg, 2009.