2007 IEEE International Conference on Intelligent Robots and Systems, San Diego/USA

Automatic adaptation of sensor-based robots

Jan DEITERDING and Dominik HENRICH

Lehrstuhl für Angewandte Informatik III
Universität Bayreuth, D-95445 Bayreuth, Germany
E-Mail: {jan.deiterding, dominik.henrich}@uni-bayreuth.de, http://ai3.inf.uni-bayreuth.de

Abstract—**We present a framework for an adaptive system capable of automatically optimizing a given sensor-based program for an industrial robot to facilitate the development of fast and robust applications. We present a systematic overview of the layers of the robot controller in which a speed-up may be gained by incorporating adaptation techniques. We then measure the potential for optimization at a specific level to determine the improvement yielded with this optimization.**

Keywords: Manipulation and Compliant Assembly; Learning and Adaptive Systems; Sensor-based robot programming; Optimization methods

I. INTRODUCTION

An emerging interest in robotics is the use of external sensors such as cameras or force-/torque-sensors to provide an intuitive way to develop robot programs. Schraft [11] and Ehrmann [4] have identified this as a major factor for small- and medium-sized enterprises. Thus, one of the next steps in robot programming is to develop programs that are capable of acquiring knowledge about their environment by using external sensors to allow for a more flexible and robust execution of the task at hand.

However, the utilization of external sensors presents two problems: First, acquiring data from the sensor can be complex. The type of information acquired is strongly dependent on the sensor itself and the environment it is used in. Additionally, the sensor data must be processed to extract the relevant information necessary for the control flow of the program. Second, the execution speed is significantly lower than that of a robot program which uses no sensors at all. The data must be processed during execution, slowing the process significantly. This problem can be compensated for if sensors are used in a preparatory manner. However, this is only feasible if the sensor can be placed in such a manner that the information can be acquired while the robot is performing some other task and the information itself is independent of the robots behavior. Because of these problems the use of external sensors in industrial robot applications is still not very widespread and limited to areas where it is absolutely necessary.

Here, we present a systematic approach for the development of sensor-based robot programs for industrial applications that are able to cope with both problems. The sensor is used to automatically acquire knowledge about the environment, which is subsequently used to reduce execution time.

The rest of this paper is organized as follows: Section II gives a brief overview of existing work in this area. In Section III we explain how we classify changes that occur during multiple executions. Then we show how adaptive strategies can be used to deal with these changes, both during development and execution of the task. Since our goal is to reduce execution time, we categorize various layers of abstraction on which a given robot program can be optimized. These levels are independent from the task at hand and can be applied to all applications. In Section IV we describe some basic experiments that were conducted to measure the optimization potential at a given layer of abstraction.

II. RELATED WORK

Although there exist numerous papers on the optimization of trajectories and acceleration profiles, e.g. [7], [8], [10] and [14], all of these generate an optimal trajectory for an invariant position and its related trajectory. These strategies are employed outside of the actual task execution and need no sensory information. The developer must identify the position or trajectory to be optimized and then check the validity of the proposed optimization by hand.

There has been some work done in the area of learning optimization strategies for industrial robots. Dauster [2] has developed a learning method for closed-loop control of assembly processes that is capable of determining optimal control parameters for a given contour using multiple executions. Chhatpar [1] deals with the problem of varying hole positions in a peg-in-hole task by means of a map of the environment. This map must be pre-acquired either analytically or be sampled systematically to ensure fast execution. Simon [12] has proposed a strategy whereby the robot program incorporates control primitives with adjustable parameters and an associated cost function. A search algorithm uses experimentally measured performance data to adjust the parameters to seek optimal performance and track system variations. Unfortunately, this proposal has apparently never been developed further.

Kahl [6] proposed facilitating intuitive robot programming by means of virtual robot environments. To avoid the absolute positions employed in robot programs without any sensors, Xiao [15] has developed a model of contact-states to characterize an assembly process solely by describing

the face- and edge-contacts of the objects involved. While this model allows for a very intuitive description of the task, there is neither an optimization strategy involved nor is it clear how the different states can be measured using sensors.

Various approaches such as [3], [5] and [13] employ learning strategies to teach skill primitives to a robot in order to facilitate intuitive robot programming. However, all of these methods are aimed at teaching a skill as flexibly as possible so it can be used in a broad range of situations, but the methods are not geared towards fast execution.

In summary, while various interesting approaches to either optimize sensor-based robot programs or alleviate the task of development exist, all of them focus on a specific task area and are not universally applicable. In this paper we will explain a systematic approach for optimization of adaptive robot programs and show how adaptive strategies can be employed to maximize automation of this optimization process.

III. Optimization of Sensor-based Robot Programs

The main idea is to reuse knowledge gained in previous executions to carry out the task at hand. Because application in industrial settings is the goal, the task itself will be always the same, but we allow indeterminacies during development and variations between different executions.

In the following, we classify the types of changes that can occur between multiple executions (Section A). In a second step we show how these changes can be accounted for to by employing suitable strategies (Section B). Finally, we show how a given robot program can be optimized with regard to its execution speed using the knowledge gained in previous executions (Section C).

A. Workspace changes

In a first step we classify the changes that a robot should be able to deal with and explain which of these can be covered using sensors. Table 1 shows how these changes can be subdivided into four groups with two characteristics: If the change is caused by the task itself or by abrasion of the environment and if the change can be dealt with in one step or if a continuous adaptation strategy is required.

		Origin of change	
		Caused by the task	Caused by abrasion
Reaction to change	One-step adaptation	Indeterminacies	Faults and Errors
	Continuous adaptation	Variations	Drifts

Table 1. Classification of changes that can occur between two executions of the same program.

Changes that are brought about by the task itself are indeterminacies and variations. The difference between these definitions is that an *indeterminacy* is something we are not aware of at this moment, but once we have learned about it, it will remain constant for a prolonged period of time and will not change during subsequent executions. One example is the position of a table in the workspace. When developing the program, the programmer usually only knows that there will be a table but not its exact location and orientation in Cartesian coordinates. The robot must be calibrated to learn this indeterminacy (usually by teach-in). *Variations* on the other hand occur every time the robot performs the task at hand. They are intentional and occur due to the heterogenic nature of the manipulated objects. This is the main reason why external sensors are used for industrial robot applications. For example, when handling food or other natural materials no object is exactly the same – the objects differ in size, weight, form, etc. In this case, the robot must be able to act flexibly enough to deal with these variations. This can either be accomplished with compliant mechanical devices or by incorporating sensors to measure the objects and act accordingly.

Changes caused by abrasion differ from those caused by the task in the sense that they are unintentional and usually undesirable; still, they must be dealt with. Here, we can subdivide the changes into two groups as well: *Faults and errors* occur when a sudden change in the workspace takes place. For example a fault would occur if the table in the workspace falls apart due to wear and tear or is moved away by a human. Faults denote an abrupt change in the environment. In this case the robot should stop execution and alert a human supervisor. *Drift* is a problem caused by gradual changes within the workspace, i.e. the settings of machines and tools changes over time. While this dislocation is minimal from one execution to the next, it can amount to a significant shift when it occurs over multiple executions. Drift can be modeled as a continuous change within the system. Classical robot programming with fixed positions is unable to deal with this problem. After a certain number of program cycles, either the machines/tools or the program itself must be re-calibrated by a human operator. Using a continuous adaptation strategy we can tutor the robot to react to this change without the need for recalibration.

Variations and drifts can both be handled with a continuous adaptation strategy and can be characterized by the same set of parameters describing the extent to which they are allowed to occur during execution. All changes exceeding this threshold are classified as an indeterminacy or fault, respectively. In sensor-based robot programs, we should be able to deal with indeterminacies and variations. Variations and drifts can be characterized by the same set of parameters, so we can include these drifts to allow the robot to tolerate them, making the program more robust. However, all faults require human intervention to resolve the problem.

B. Learning to deal with changes in the environment

In sensor-based robot programs, it is impossible to model the drift in the environment since it is unknown. Variations will be modeled explicitly by the programmer. The parameters for these models are extracted by a long series of tests until they are good, i.e. fast enough to warrant industrial application. In our system, one model describing both variations and drift is empirically acquired and learned by the robot during multiple executions of the program. To achieve this, we incorporate adaptation strategies into the system.

Initially the robot has little or no knowledge about the workspace. Execution will therefore be successful but very slow because the variation/drift model is very vague. Sensor information gained during the execution is used to update the world model of the workspace. Afterwards optimization strategies are employed that use the updated world model to reduce uncertainties and speed up execution (Figure 1). This process will be repeated in every execution.

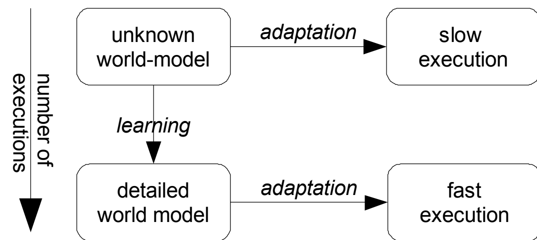

Figure 1. Structure of the learning and adaptation cycle. Initially, the world model is unknown so adaptation strategies can only facilitate slow execution. After multiple executions the world model becomes more and more detailed, producing enough knowledge to enable the adaptation strategies to allow for fast execution.

There are two important things a robot can learn: The first is the general position and the trajectory leading to the target position. In Table 1 these are identified as indeterminacies. The second is the (allowed) variations and the drift it should be able to deal with. We do not require the robot to learn high-level concepts such as a general peg-in-hole skill.

To ease programming, the developer should not have to deal with specifying indeterminacies but rather should give general instructions on how to resolve these indeterminacies, for example using commands like “touch the table, then search for the hole on the table”. This results in a calibration; the robot will search for this hole during the first execution and store its location in a database so this search will be performed only once. This calibration can be regarded as a kind of one-step-adaptation. In an open-loop system, this is all that has to be done as no sensors are needed to further control the movement of the robot unless we assume that there may be drift in the workspace.

In a closed-loop system additional variations will occur. Here, movements can be triggered and stopped based on sensor information. In this case the robot can learn optimal threshold values for the sensor data to start and stop the movement as, proposed by Schlechter [9]. Another property the robot can learn is how to minimize the uncertainty model for a given location, so that the search for that location is sped up. During the first iterations these models will cover a large area. Accordingly, the time required for the search is extensive. The more exact the model becomes, the faster the search can be executed.

Before we show how to use this knowledge to actually optimize the robot program, we classify optimization strategies according to the influence they have on the robot's behavior during execution.

C. Adaptation strategies

For any given robot program, we can divide the task of optimizing this program into four layers of abstraction with respect to the execution speed (Table 2). The higher the layer, the more the internal structure of the program will be changed. Note that we only deal with optimization of the program code itself. Neither do we re-arrange the workspace nor do we introduce new mechanical devices, such as better sensors or intelligent robots.

Layer of abstraction	Type of optimization	
	Open loop	Closed loop
4	Execution order of logical task-steps	
3	Interval points of transfer movements	Sensor-dependent positions
2	Trajectories	Controlled movements
1	Acceleration and velocity profiles	
0	Unoptimized robot program	

Table 2. Layers of abstraction for the optimization of industrial robot programs. On the lowest layer we have the program “as it is”. No optimization has taken place yet. The increasing layers imply more and more significant changes to the behavior of the robot.

The lowest layer (zero) represents the robot program in its original state, i.e. no optimization has taken place yet.

In the first layer, we can optimize the acceleration and velocity profiles along the trajectories of the robot. Usually every (transfer-)movement can be separated into three parts: The departure from the environment, the transfer itself and finally the approach to the environment. Various methods exist to determine the optimal trajectory (and subsequently the corresponding acceleration profile) for two given positions, e.g. [14] and [8].

In the second layer we deal with the task of calculating the “best” - either the shortest or the fastest - route to a given goal position. Here, we must distinguish between two categories: Open-loop movements and closed-loop movements. In the case of an open-loop movement, the

optimal route is the best trajectory between the actual position and the goal position, as there are no external constraints on the trajectory or the goal position. Once again, if the positions are known, we can use existing methods such as in [14] to calculate the optimal trajectory. However, if we need sensor information to accomplish the movement, as is the case in a closed-loop program, we need to optimize the control itself. This is accomplished by minimizing the “overshoot” in every execution cycle of the controller.

In the third layer of abstraction we optimize the positions themselves. Once again it is useful to distinguish between open-loop and closed-loop programs. If the application-specific positions are fixed and will not change between repeated executions of the program, there is no possibility for optimization. However, it may be useful to determine better interval positions that may result in a shorter overall trajectory to the goal position. In the case of a closed-loop program, the position is dependent on the sensor information available. A typical example is the hole in a peg-in-hole task. Here the robot is guided to the exact location of the hole by a sensor. Another example is movements that are started or stopped by the sensor to compensate for environmental uncertainty. Two possible approaches for optimization are to either re-calculate the starting position of the search to place it closer to the goal position - thus reducing the time required for the search - or to determine better-suited threshold values for sensor-controlled movements to prevent errors, i.e. a false decision to start or stop a movement.

In layer four of the abstraction diagram we can reorder certain logical parts of the program to reduce waiting time or generate better paths along multiple positions. It may also be possible for the robot to execute another task while waiting for a machine to finish processing of an object. While this level still defines an optimization on the software-side of the system, it requires external knowledge about the structure of the task, e.g. which robot commands must be treated as an entity and pre-conditions that must be met before another part of the task can be executed.

It should be noted that the optimization at a high layer of abstraction produces the possibility for optimization at lower layers as well. For example, if we compute a new position which is somehow better suited for the task at hand, we can also re-calculate the trajectory leading to that position and subsequently the acceleration profile for that new trajectory.

This layering of optimization strategies is purely theoretical so far. Up to now the decision to optimize the program is nearly always made by a human supervisor who selects a certain strategy out of the various algorithms mentioned in Sections II and III and applies them to a designated part of the program. In this system we propose that this optimization be performed automatically every time the task has been completed. The developer only marks the areas and layers of abstraction where optimization is

Length of motion	Factor by which acceleration is scaled		
	1	0.9	0.8
1cm	0.271 (100 %)	0.287 (106 %)	0.288 (106 %)
10 cm	0.736 (100 %)	0.751 (102 %)	0.767 (104 %)
50 cm	1.696 (100 %)	1.728 (102 %)	1.760 (104 %)

Table 3: Required time for a motion of a given length and a set maximum acceleration. The values in brackets denote the change in percent to the reference motion (factor 1).

	Factor by which trajectory is scaled		
	1	1.1	1.2
Time elapsed	2.57 s (100 %)	3.16 s (123 %)	3.40 s (132 %)
Total of joint-rotations	36.1° (100 %)	39.2° (109 %)	41.8° (116 %)

Table 4: Required time and total sum of degrees covered by all robot joints during a PTP motion between two points. The values in brackets denote the change in percent to the reference motion (factor 1). The length of the trajectory is scaled by adding a virtual interval position which is covered by the trajectory.

permitted while the actual optimization itself is done by the computer.

We thus provide an easier and faster method to develop adaptive robot programs that can reach acceptable execution times.

IV. Measuring the Potential for Adaptation

In this section, we will explain some experiments conducted to examine to which degree a given robot program may be optimized along the layers described in Section III.

When deciding to optimize a given robot program the programmer must decide which parts of it should be improved. Usually he will choose those parts in which he believes execution time can be gained. To be able to deal with this situation analytically, we have set up some basic experiments for each layer described in section III. The idea is to define a typical situation and a very good (if not the best) solution to it. We will then downgrade this solution by a certain measure and see how much longer the robot requires to reach its goal. We think that in most cases this reasoning works the other way round as well, that is, the same amount of time is gained when the solution is improved by the same measure. The goal is to have some kind of chart that tells us for which layers an optimization is auspicious and in which cases the amount of time required to find an optimization may not offset the benefits. All experiments were conducted on a Stäubli RX130 robot with the monitor speed set to 10. The sensor we have used is a wrist-mounted force/torque sensor 90M31A from JR3.

For the lowest layer we set up an experiment where the robot moves along a given trajectory using the square wave

acceleration profile provided by the manufacturer. The only parameter modifiable for this profile is the maximum allowable Cartesian acceleration. In the experiment, we perform a straight line motion between positions A and B. The goal position B is set to be either 1cm, 10cm or 50cm from the the starting position A, respectively. For each of these three motions, we use the maximum allowable acceleration and an acceleration speed reduced to 90% and 80% of the maximum value, respectively. We measure the time it takes the robot to complete the motion and calculate the percentage by which this differs from the fastest possible motion (Table 3) .

To measure the impact of a trajectory optimization in Layer 2 we set up a PTP trajectory between the given positions A and B. (Cartesian motions are omitted since they are always slower than PTP motions.) We then scaled the length of this trajectory in joint space by adding a virtual interval position C on a continuous path and moving this position so that the resulting overall trajectory from A to B in joint space is stretched by a factor of the original length. We measured the time the robot required to complete the motion and the overall sum of degrees that all joints rotated during the motion execution (Table 4).

The experiment used to measure the impact of an optimization of interval points for complex trajectories (Layer 3) is similar to the one described above. However, instead of using a PTP motion, which essentially moves all joints of the robot regardless of external constraints (Layer 2), here we typically employ straight-line motions to navigate around obstacles. A typical example is a pick-and-place operation where the robot has to move the grasped object around an obstacle to reach the goal position (Figure 2). The better the intermediate position C is chosen, the shorter the overall trajectory will be. Thus, for this experiment we set up a pick-and-place operation and measured the time it takes the robot to move from starting position A to goal position B using a continuous path motion covering the intermediate position C. We scaled the length of the trajectory by shifting position C and measured the time it takes to complete the motion as well as the total number of joint rotations of the robot (Table 5).

To measure the impact of sensor-controlled movements (Layer 2, closed-loop), we set up an experiment that moves the robot a set distance along the surface of a table. For simplicity's sake, we used a PID controller and determined reasonable values for the proportional, integral and derivative parts, so that the movements appear fluid to the human eye. We then scaled this value by 10% and 20% respectively and measured the time and the total number of joint rotations that occurred while performing this sensor-controlled movement (Table 6).

In the case of sensor-dependent positions (Layer 3, closed loop), there are actually two parameters we can vary: The first is the speed of the movement itself and the second is the threshold value of the sensor, which tells us if we have reached the goal position. As in the other experiments, we

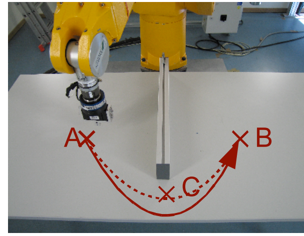

Figure 2: Experimental setup for Layer3. The robot moves from A to B, avoiding the obstacle by employing the intermediate position C. For this experiment, we moved C outwards from the dotted trajectory so that the resulting overall trajectory from A to B was elongated by a set factor.

	Factor by which trajectory is scaled		
	1	1.1	1.2
Time elapsed	2.57 s (100 %)	3.34 s (130 %)	3.47 s (135 %)
Sum of joint-rotations	36.5° (100 %)	38.9° (107 %)	41.0° (112 %)

Table 5: Required time and total sum of degrees covered by all robot joints during a pick-and-place motion between two points A and B with intermediate position C. The trajectory-length is scaled by moving C. The values in brackets denote the change in percent to the reference motion.

	Factor by which PID parameters are scaled		
	1	1.1	1.2
Time elapsed	16.98 s (100 %)	17.12 s (101 %)	17.20 s (101 %)
Sum of joint-rotations	37.0° (100 %)	37.2° (100 %)	37.2° (100 %)

Table 6: Required time and total sum of degrees covered by all robot joints for a PID controlled move along a surface of X centimeters. The values in brackets denote the change in percent to the reference motion (factor 1).

	Factor by which movement speed is scaled		
	1	1.1	1.2
Time elapsed	62.81 s (100 %)	62.85 s (100 %)	62.83 s (100 %)
Overshoot	-	0.1mm	0.2 mm

Table 7: Required time and overshoot of the the tool-tip compared to the original motion for a force-guarded move onto a flat surface. The values in brackets denote the change in percent to the reference motion (factor 1).

	Factor by which route is worse than optimal route		
	1	1.1	1.2
Time elapsed	50.81s (100 %)	60.20 s (118 %)	59.20 s (117 %)
Sum of joint-rotations	308.8° (100 %)	498.3° (161 %)	530.2° (172 %)

Table 8: Required time and total sum of degrees covered by all robot joints for a route along all corners of a cube with a side length of 20 cm. Three different routes were used: The shortest, and two routes which are among the top ten percent and top twenty percent of all possible routes respectively. The values in brackets denote the change in percent to the reference motion.

<table>
<tr><th></th><th colspan="4">Layer of optimization</th></tr>
<tr><th></th><th>1</th><th>2</th><th>3</th><th>4</th></tr>
<tr><td>Open loop</td><td rowspan="2">3 %</td><td>22 %</td><td>30 %</td><td rowspan="2">18 %</td></tr>
<tr><td>Closed loop</td><td>1 %</td><td>0 %</td></tr>
</table>

Table 9: Possible gain on each layer of optimization if the corresponding parameters are optimized by 10 %.

employed a force-guarded movement on a flat surface that stops when the force along the z-axis of the tool-tip exceeds 10 N. In the first experiment we scaled this threshold-value and measured the time it takes to execute the motion and the overshoot to the original motion. We repeated each motion ten times and computed the average of both time and overshoot. In this experiment there were no measurable differences between the original motion and the two scaled motions, so we conducted a second experiment in which the threshold value remains unchanged and we scaled the motion speed with which we approached the surface to 90% and 80% of the original velocity, respectively (Table 7).

In the highest layer, execution order of logical task steps, we defined eight positions forming a cube with a uniform edge length of 20 cm. The task is to move to all corners of the cube starting at one corner and then back to the start. This experiment is a practical implementation of the traveling salesman problem. For eight points there are 5040 different routes presenting a valid solution. We determined the shortest route and two routes that are among the top 10% and 20% of all routes with respect to their length. As in all other experiments, we measured the time and the total sum of joint rotations occurring along the route (Table 8).

We summarized the possible gain yielded by all experiments in Table 9. Only the time potential for an optimization of 10% is displayed. (For the potential at Layer 1, we used the average of three results.)

We can see that an optimization by 10% in a given layer seems to be very profitable for the open-loop parts of the program. However, two things have to be considered: First, the results for Layers 1 and 2 are of theoretical value at best. An optimization of a robot's acceleration profile usually requires a new robot with a higher rate of acceleration, because robots usually employ the best - that is fastest – acceleration available. In Layer 2 the best motion from one position to another is a PTP motion, if there are no external constraints such as obstacles etc. This motion can be regarded as a straight line motion in the joint space of the robot and is already the shortest joint motion. While the results for the closed-loop parts of the program seem to be disappointing, it must be kept in mind that a good choice of parameters for a given sensor is tricky at best. Unless the programmer is highly experienced in this field or a very thorough calculation has been done, the choice of these parameters is usually trial-and-error. While the gain is minimal if an already well-chosen parameter is moved closer to the optimal value, the gain will be much higher if the same parameter was poorly chosen to begin with. We believe that the potential for optimization in these layers is much higher than these first experiments predict. The potential for optimization in Layers 3 (open loop) and 4 is significant and it might be worth the effort to attempt an optimization for these layers.

V. Conclusion

We have presented here a systematic approach to the optimization of a given robot program for industrial applications. We have classified workspace changes by two characteristics: The origin of the change and the robots reaction to it. We have argued that, in principle, we can deal with all of these changes except for faults and errors. Based on this, we have shown how adaptation strategies can be employed to deal with these changes. All information gained in previous executions is stored in a database to be used for a better-adjusted execution of the task at hand. In the next step we developed a scheme to assign optimization strategies to a distinct layer of abstraction from the original program. We found that there are exactly four layers that are independent of the actual task. Finally, we presented a set of experiments to measure the potential for each layer of optimization. An analysis of the experiments has shown that the most promising aspects for optimization are found in the high layers of abstraction.

Further work needs to be done in several areas. A complete mathematical model should be derived to model the variations and drifts adequately. For a given optimization layer from Table 2, we will test and evaluate various adaptation strategies to see if they can be used to automatically optimize a certain aspect of a robot program.

References

[1] S. R. Chhatpar, „Localization for robotic assemblies with position uncertainty“, Intelligent Robots and Systems, 2003. (IROS 2003). Proc.. 2003 IEEE/RSJ Int. Conf. on, Vol. 3, Issue , 27-31 Oct. 2003 Page(s): 2534 - 2540 vol.3

[2] K. Dauster, „Prozessangepasste, lernende Roboterregelung für Montageprozesse“, VDI Verlag Düsseldorf, ISBN 3-18-392508-7

[3] R. Dillmann, „Building elementary robot skills from human demonstration“, in Proc. of the IEEE Int. Conf. on Robotics and Automation, v. 3, pp. 2700-2705

[4] M. Ehrmann, M. Seckner, D. Zuehlke, “Simplified programming of robot assembly cells by using motion oriented elements”, Proc. of the 37th Int. symposium on robotics (ISR) 2006

[5] T. Hasegawa, „A model-based manipulation system with skill-based execution“, Advanced Robotics, 1991. 'Robots in Unstructured Environments', 91 ICAR., Fifth Int. Conf., page(s): 970-975 vol.2

[6] B. Kahl, D. Henrich „Virtuelle Roboter Programmierung: Konzept und prototypische Implementierung“, VDI Robotik 2004, München, Deutschland, June 17.-18., 2004

[7] K. Marti, “Path Planning for Robots by Stochastic Opt. Methods”, Jour. of Intelligent and Robotic Systems, Vol. 22 , Issue 2, pages 117 – 127, 1998

[8] W. Neubauer, M. Moller „Learning systems behaviour for automatic correction of off-line robot programs“, Intelligent Robots and Systems, 1992., Proc. of the 1992 lEEE/RSJ Int. Conf. on, Vol. 2, Issue , 7-10 Jul 1992 Page(s):1355 – 1362

[9] A. Schlechter, D. Henrich, „Discontinuity Detection for Force-based Manipulation“, IEEE Int. Conf. on Robotics and Automation, Orlando, Florida, USA, May 15 - 19, 2006

[10] M. Schlemmer, G. Grübel, „Real-Time Collision- Free Trajectory Optimization of Robot Manipulators via Semi-Infinite Parameter Optimization“, The Int. Jour. of Robotics Research.1998; 17: 1013-1021

[11] R.D. Schraft, „The need for an intuitive teaching method for small and medium enterprises“, ISR 2006 - ROBOTIK 2006 : Proc. of the Joint Conf. on Robotics, May 15-17, 2006, Munich

[12] D.A. Simon, „Self-tuning of robot program primitives“, Proc. of the 1990 IEEE Int. Conf. on Robotics and Auto. (ICRA '90), Vol. 1, May, 1990, pp. 708 – 713

[13] U. Thomas, „Error-tolerant execution based on robot skills“, Robotics and Automation, 2003. Proc.. ICRA apos;03. IEEE Int. Conf. on, Vol. 3, Issue , 14-19 Sept. 2003 Page(s): 3069 - 3075 vol.3

[14] B.Tondu, “The Three-Cubic Method: An Optimal Online Robot Joint Trajectory Generator under Velocity and Acceleration Constraints”, The Int. Jour. of Robotics Research, Vol. 18, No. 9, 893-901 (1999)

[15] J. Xiao, „Automatic Determination of Topological Contacts in the Presence of Sensing Uncertainties," Proc. 1993 IEEE Int. Conf. on Robotics and Automation, pp. 65-70, Atlanta, May 1993.