A System to Automate the Generation of Program Variants for Industrial Robot Applications

Eckhard Freund¹, Bernd Luedemann-Ravit²

¹Institute of Robotics Research, University of Dortmund, Germany, freund@irf.de
²Institute of Robotics Research, University of Dortmund, Germany, luedemann@irf.de

Abstract
This paper presents a novel system that allows to generate variants of industrial robot programs. Such a system is widely needed in different areas of application. First, product scenarios change often slightly and thereby cause high programming costs. Using our system reduces these costs, because non-specialists can create a specific program variant by themselves. Second, an engineer needs numerous program variants to find an optimized cell layout, e.g. regarding robot execution time. Present systems require too much manual effort and they are very limited in considering application-specific requirements. Consequently, we have implemented a system that contains an interpreter to execute user-defined scripts. A script contains the information on how to generate the program variants. Each change of the parameter values inside the script results in a different program. We simulate the program execution and use the simulation results to further improve the programs. We prove the usefulness of our system by industrial applications.

1 Motivation
There are different methods to program a robot: online and offline programming methods. In industrial environments, the online teach-in method is still the one that is mostly used, because it is easy to learn and application-independent. However, the teach-in times are long and therefore expensive, especially for products with complex geometries or with many variants [1]. Therefore, modern offline programming systems offer modules to ease the programming of special applications such as spot or arc welding [2]. With these systems, skilled users can program each product variant in an interactive manner. However, the deficiency of present offline programming systems is the missing ability to rapidly adapt to requirements that arise from scenario changes or customer requirements [3]. Consequently, a highly-qualified programmer is required to modify the generated code manually before executing it on an industrial controller [4]. Therefore, present offline programming methods are unsuited to rapidly program a robot for numerous product variants, because for each variant, the engineer has to repeat all necessary programming steps manually. We present a typical example to demonstrate the practical problems that our system can solve. A Mitsubishi robot must polish the hatched border of many different metal parts at the contact point of the polishing machine (Fig. 1). These parts vary in their size and their shape of the border. A new robot program is necessary for each different metal part. The first problem, the programming problem, is to enable a non-specialist to program the robot for all possible product variants in minimal time. With our system, an engineer can easily implement a program generator that creates a polishing program for these parts. With this generator a non-specialist can then create a new program only by setting a few parameters.

Figure 1: Example "Polishing a Metal Part Border"

Besides for the programming of robots, companies use offline systems to plan robot work cells. Their engineers must find solutions for complex layout planning problems. In our example, the second problem, the planning problem, is as follows (Fig. 1): Where must the polishing machine be mounted on the ground plate, so that the robot polishes the parts in minimal time, considering one or even all product variants? Conventionally, the engineer tries to find a solution by manual trial-and-error methods. Present automated approaches do not consider the execution time on the basis of real robot programs, but on a generalized
internal representation that cannot consider controller-specific issues and that can therefore not provide sufficiently reliable results [5,6]. With the support of our system that generates robot program variants in controller-specific syntax, the engineer can find a more reliable and probably better solution, because he can really execute every program variant that is needed for each cell variant. Additionally, instead of trial-and-error methods, he can also use optimization strategies that are suitable for his scenario.

2 Requirements

To create numerous robot program variants, a system must fulfill the following main requirements:

1. **Geometric Extraction**: The system must provide functionality to extract geometric information from CAD models. This simplifies the retrieval of positions from complex products. Regarding our programming problem (Fig. 1), one possibility to calculate the path points from the CAD model of the metal parts is an intersection of an additional plane with the metal parts (Fig. 2), because all metal parts are planar. The result of the intersection is a curve, represented by an unsorted set of points \( E = \{ P_1, \ldots, P_N \} \) relative to the object coordinate system (Fig. 2).

![Figure 2: Extraction of Geometric Information](image)

2. **Algorithm Definition**: An engineer must be able to define algorithms that operate on the extracted geometric information. This is necessary to consider application-specific calculations. Regarding our programming problem, the engineer can, for example, change the orientation of the extracted points, or he can calculate an additional distance of the points \( P_i \) normal to the border (Fig. 1, 3).

![Figure 3: Phases of the Program Generation](image)

3. **Controller Independent Program Specification**: The system needs a controller independent specification of the desired sequence of robot program commands that must occur in the generated programs, such as movement statements, or I/O commands. From this specification, the system generates the program in the required controller specific syntax (Fig. 3). The planning engineer often has to solve a planning problem of the following kind: selecting the best robot from a set of available robots according to application-specific requirements. To transfer the programming task among robots from different vendors with minimal effort, a controller independent program specification is the only reasonable way.

4. **Program Simulation**: It is necessary to execute the controller-specific programs in a simulation. One reason is to check if the programs are error-free. Another reason is the retrieval of simulation results for a further improvement of the programs. Regarding our planning problem, we must execute each generated Mitsubishi program variant completely in a simulation to retrieve the robot execution time to optimize the position of the polishing machine.

5. **Open Architecture**: Due to complexity reasons, no robot programming system can integrate all necessary functionality right from the start [7]. Therefore, the system architecture must have an open interface to extend its functionality, e.g. specific path planning algorithms, algorithms concerning computational geometry, or optimization strategies.

3 Architecture of the Generation System

From the above requirements, we define the architecture of a generation system and present our implementation. Our system consists of the following four architecture containing two closed loops (Fig. 4):

- Application layer
- Generation layer
- Simulation layer
- Manufacturing layer

In the next sections, we describe each layer according to its responsibilities within the complete system.

3.1 Application Layer

The application layer contains the information that is necessary to generate the robot programs – except for the geometric information. The application layer is divided into two different parts.

1. The first part receives input parameters from the user to generate one specific program variant via the input assistant.

2. The second part contains the information to generate all possible program variants.
The second part consists of the information resources:
- Generation plans
- Syntax definition
- Technology packages

The engineer has to write a script that contains all necessary commands to generate the program variants for his problem. Within this script, the engineer can use the systems built-in functionality. We call such a script a generation plan. An engineer can prepare such a plan for a specific problem. After the plan preparation, a non-specialist can generate program variants by parameterization via the input assistant. The generation plan contains the commands for the geometric extraction of robot positions, further necessary algorithms to calculate complete robot paths, the controller independent program specifications, and the use of strategies to further improve the programs from simulation results. The plan language is comparable to a script language for rapid application development.

The syntax definition contains the information about the robot programming language, e.g. KUKA KRL or Mitsubishi MELFA BASIC. The advantage of isolating this syntax information is that no further intervention into our system is necessary if the robot language slightly changes. Such changes occur often if there is a version update of the robot controller software.

We can integrate technology packages, written in controller-specific syntax, into our generated robot programs. The robot vendors offer such packages to ease the programming for a robot task requiring a specific technology, e.g. KUKA offers ArcTech and ABB offers ArcWare to ease the programming of an arc-welding task.

### 3.2 Generation Layer

Within the generation layer, there are two system parts:
- Generation controller
- Program synthesis

The generation controller is mainly an interpreter and is responsible for the execution of the generation plans. The generation controller has an open interface to integrate further functionality in addition to the built-in functionality, such as specific path planning algorithms, algorithms concerning computational geometry or optimization strategies. The engineer can easily use such functionality coupled to the system via the open interface within his generation plans.

The program synthesis receives the controller independent program specification from the generation controller and reads the required syntax definition. From these two input sources, the program synthesis creates a robot program variant in controller specific syntax.

### 3.3 Simulation Layer

We use COSIMIR® to implement the simulation layer, which is responsible for various tasks.

1. The simulation layer provides the geometric information for the products located within the simulation model. Regarding our example, the simulation model contains the CAD model of the different metal parts that the robot has to polish. It also contains a complete model of the robot work cell. Fig.1 shows the geometric part for the simulation model for our polishing example.
2. The simulation layer transforms the controller-specific robot programs and executes them on a virtual robot controller (VRC) [8]. This layer sends an event to the generation controller when an execution error occurs. Regarding our example, the VRC executes the Mitsubishi program after its transformation and notifies the generation controller if the polished metal part collides with the polishing machine, or if the robot exceeds any joint limits.

3. The simulation layer provides simulation results for the generation controller (inner closed loop). Regarding our planning problem of positioning the polishing machine, the generation controller can ask for the execution time that the robot has needed to polish the metal part.

3.4 Manufacturing Layer
The manufacturing layer is the real robot work cell. It contains all necessary hardware components. The responsibility of this layer is the real execution of the robot programs. For the non-specialist, it provides information about necessary changes of the input parameters to achieve satisfying execution results (outer closed loop).

4 Generation Plans
The generation plans contain the complete information on how to generate the program variants. The plan language is kept simple, so that it is easy to learn for an engineer. The basic elements of such a plan are plan variables and plan steps.

4.1 Plan Variables
Plan variables are necessary to store data. The plan language offers built-in types for the variables (Tab. 1).

Table 1: Subset of built-in data types in the plans

<table>
<thead>
<tr>
<th>Group</th>
<th>Type</th>
<th>Plan Type Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard</td>
<td>Real</td>
<td>$NUMBER</td>
</tr>
<tr>
<td>Geometry</td>
<td>Vector</td>
<td>$VECTOR</td>
</tr>
<tr>
<td></td>
<td>Plane</td>
<td>$PLANE</td>
</tr>
<tr>
<td>Robot</td>
<td>Robot Position</td>
<td>$POSITION</td>
</tr>
<tr>
<td>Sim. Model</td>
<td>Object</td>
<td>$OBJECT</td>
</tr>
</tbody>
</table>

Additionally, the plan language provides multidimensional, dynamic arrays for every type with unknown size, because at the beginning of a plan execution, the required array size is unknown. The definition of an array within a plan looks like the following (Two minus character represent a comment):

```
-- two dimen. array of reals named matrix
$NUMBER[][] MATRIX
```

4.2 Plan Steps
Plan steps define the instructions using the plan variables. Calling a step looks like:

```
$example_step(IN1, ..., INn => OUT1, ..., OUTm);
```

A plan step call has input parameters (call-by-value) and output parameters (call-by-reference). Our system offers built-in plan steps according to the stated requirements, e.g. for the intersection of a CAD model with a plane (Tab. 2). The user can also define his own plan steps similar to a procedural programming language.

<table>
<thead>
<tr>
<th>Plan Step Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$INTERSECTION()</td>
<td>Object-Plane Intersection</td>
</tr>
<tr>
<td>$STRETCH()</td>
<td>Stretch a Closed Robot Path</td>
</tr>
<tr>
<td>$NORMAL()</td>
<td>Normal Vector on a Surface</td>
</tr>
<tr>
<td>$BOUNDING_BOX()</td>
<td>Smallest Box around an Object</td>
</tr>
<tr>
<td>$RUNTIME()</td>
<td>Runtime of Program Simulation</td>
</tr>
<tr>
<td>$SHOOKE()</td>
<td>Non-linear optimization strategy</td>
</tr>
</tbody>
</table>

5 Application of the Generation System
In the following section, we present a guide for an engineer to solve a programming or a planning problem by filling out the application-specific fields for his scenario (Fig. 5). All fields are included in a plan framework. Each field adds plan steps and plan variables to the plan framework. The result of filling out all fields is a complete generation plan. We illustrate the use of this guide with our programming problem of polishing many metal part variants.

5.1 Parameter Definition
First, the engineer has to analyze the problem and define the characteristics that are the same for all program variants, that means which are the invariants of all his programs. He further needs to know the varying characteristics of his programs. For each varying characteristic, he has to define a plan variable, including a type, a name and a default value.

Regarding our programming problem, the geometric topology of the metal parts is invariant (Fig. 1), e.g. no holes and the parts are planar. The parts vary in the shape of the border. Additionally, we allow the non-specialist to define an additional distance from the border to the contact point (border offset). Therefore, we need the following parameter and add it to the generation plan.

```
-- ADD VARIABLES FOR THE PARAMETER DEFINITION
-- Distance: Border - Contact Point CP
$NUMBER BORDER_OFFSET = 0.2; --Default: 0.2mm
```
5.2 Geometric Extraction

To solve the programming problem, the engineer has to define a procedure to extract the path points from the CAD model. If a planning problem does not need positions from a CAD model, the geometric extraction is not necessary, and the robot positions can be used from a user-defined robot position list.

Regarding our programming problem, we first need to retrieve the CAD model from the simulation model represented as a sequence of surface polygons.

`-- ADD VARIABLES FOR THE GEOMETRIC EXTRACTION
-- Metal Part Name in Simulation Model
$OBJECT PRODUCT = <Metal Part 1>
-- 2-Dim Array: [[i][j]] = Point j of Polygon i
$VECTOR [[i][j] PRODUCT_MODEL]
-- CAD model

-- REQUEST OF CAD MODEL FROM SIMULATION MODEL:
$REQUEST_POLYGONS(PRODUCT=>PRODUCT_MODEL[])`;

5.3 Calculation of Robot Positions

In this step, the engineer must do all further calculations to get the robot positions that should occur in the generated programs.

Regarding our programming problem, we get the path points through intersection between the metal part and a plane. To simplify the example, the user defines the plane within the simulation model. The engineer could also have calculated it inside a plan. Together with the calculations of the border offset, we add to the plan:

`-- ADD VARIABLES FOR THE CALCULATIONS
$OBJECT PLANE = <Plane>
-- Plane Name in Sim. Model
$POSITION PATH[]
-- Variable to store a Path
-- REQUEST OF CAD MODEL FROM SIMULATION MODEL:
$REQUEST_POLYGONS(PRODUCT=>PRODUCT_MODEL[])`

Further calculations are not explained in detail here, e.g. transforming the path points relative to the polishing machine.

5.4 Program Specification

The engineer must define the sequence of controller independent robot commands that must occur in the generated programs. Regarding our programming problem, we need a simple program. We define it by calling plan steps:

`-- ADD STEPS FOR THE PROGRAM SPECIFICATION
$PTP(PATH[1])
-- ptp to first pos.
$SMOOTH_PATH(PATH[])`

From this specification, the program synthesis (Fig. 4) generates the corresponding program together with the robot positions. The program synthesis inserts the specification steps as comments into the program for clearness reasons. For our polishing problem, the following simplified Mitsubishi program is generated:

`10 REM PLAN STEP: $PTP(PATH[1])`
`20 MOV PATH(1) 'move point to point`
`30 REM PLAN STEP: $SMOOTH_PATH(PATH[])`
`40 CNT 1 'switch on smoothing`
`50 FOR I = 1 TO 202 'path with 202 positions`
`60 MVS PATH(I) 'single linear segment`
`70 NEXT I`
`80 CNT 0 'switch off smoothing`

By choosing another syntax definition, our system can generate the program in other robot languages. Therefore, the engineer need not to know the controller specific syntax to program the robot. He only must know the plan syntax.

5.5 Observation, Evaluation, and Variation

After the generation of the robot program, the system executes the programs inside the simulation layer. If the program execution causes an error, the simulation layer sends an event to the generation controller.

Within the observation (Fig. 5), the engineer can define plan steps that react on the events generated by the simulation layer. The engineer can collect simulation results while the robot programs are executed, e.g. the current distance of the joint values to their joint limits.

Within the evaluation, the engineer defines an evaluation function using the simulation results. Regarding our planning problem of finding an optimized machine position, this evaluation function is the time to execute the polishing program. The functions input parameters are the coordinates of the machines location. Within the variation, the engineer selects a optimization strategy that minimizes the evaluation function. Regarding our planning problem, we must use a non-linear strategy that does not need a derivate, e.g. Hooke-Reeves. The strategy varies the location of the machine.
6 Results

We have built several industrial applications with this generation system, e.g. a program generator to synthesize Mitsubishi MELFA BASIC programs for a grinding and cleaning process for several hundreds of porcelain plate variants (Fig. 6a).

![Figure 6: Applications of Program Generation](image)

To illustrate the effort reduction, we compare the effort that was necessary so far using the teach-in method with the effort using the generation system (Tab. 3). The time to teach one plate corresponds to 100%.

<table>
<thead>
<tr>
<th>Number Variants</th>
<th>Teach-In [%]</th>
<th>Gen. System [%]</th>
<th>Reduction [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100.0</td>
<td>70.0</td>
<td>30.0</td>
</tr>
<tr>
<td>2</td>
<td>200.0</td>
<td>71.4</td>
<td>64.3</td>
</tr>
<tr>
<td>3</td>
<td>300.0</td>
<td>72.2</td>
<td>75.9</td>
</tr>
<tr>
<td>100</td>
<td>10000.0</td>
<td>210.0</td>
<td>97.9</td>
</tr>
</tbody>
</table>

The time to implement the generation plan for the plate variants takes about 70%. All further variants are generated in about 1.4%, including the preparation of the CAD data, and the download time of the program to the controller. For 100 variants, we save up to 97.9%. Additionally, we made the experience that the concept of the generation plans has a great practical benefit: We could easily extend the generator and rapidly implement customer request arising during the setup phase of the robot work cell, e.g. no tool frame changes. Regarding several applications, we have also made the experience that the implementation of a plan to generate programs for a complete product spectrum takes as long as programming one product using conventional programming methods. Through the execution of a plan, we achieve minimal reprogramming effort for a new product variant.

Finally, we present the solution of a planning problem: finding the closest position of a Schunk gripper depot relative to a KUKA robot (Fig. 6b), because the amount of space for the tool exchange is very limited. We have prepared a generation plan that relocates the depot position. Afterwards, the KUKA programs are created, simulated and evaluated. A program contains the robot paths for all possible tool exchange scenarios together with the digital I/O communication to the gripper depot. We have found a depot location that was about 10% closer to the robot than the solution of an experienced engineer. We could also use the optimized KRL program without any modification effort.

7 Summary

We have presented a system to solve programming and layout planning problems based on real robot programs. Using conventional systems, time consuming, repetitive manual work is necessary to create robot programs for product variants, and for finding optimized robot work cells. With our system, we reduce this effort and can even find better planning solutions by means of flexible scripts. In contrast to present offline systems, our system can also cope with application-specific and customer-specific requirements which are essential for practical solutions in industrial robot applications.

8 Outlook

We will investigate the standardization of generation plan skeletons for typical programming and planning problems to further speed up the development process for finding optimized solutions.

References