Abstract

We present our successful efforts to improve intuitive programming of synchronized dual-arm operations in industrial robotic assembly tasks. To this end we extend an earlier proposed skill representation (based on our work with ontologies for industrial robots) and our respective programming interface to integrate options for the specification, adaptation and refinement of synchronization points, synchronized motions and master-slave relations during program parts. Our approach supports the user by handling motion constraints and geometrical transformations necessary for refinement and transfer of program sequences between arms implicitly. In turn, the knowledge necessary to describe and represent the respective operations for later re-use and refinement is made explicit and can hence be captured in our skill representation. We report on two experiments confirming the applicability and efficiency of our approach.

1 Introduction

Over the recent years a new type of inherently safe industrial manipulators intended for direct collaboration with human shop floor workers has entered the market, examples of which are Universal Robots’ UR5, ABB’s YuMi, Rethink Robotics’ Baxter and KUKA’s LBR iiwa. These, as well as some traditional manipulators can be set up in a dual-arm configuration (in the case of ABB’s YuMi or Rethink Robotics’ Baxter this is the default solution) to handle bi-manual tasks, e.g., in product assembly. A common property of these robot models is the lead-through programming, or kinesthetic teaching, mode which enables the user to physically guide the robot manipulator into desired positions instead of using e.g., a joystick (Rodamilans et al. 2016, for example). Still, the robot program has to be created as a sequence of motions and control logic, e.g., by adding instructions one by one using a teach pendant (shown in fig. 2), a textual program editor installed on a PC or simplified programming interfaces such as the YuMi Online Windows app.

An approach to making the programming of complex robot instructions more available to non-expert users is the use of high–level instructions to simplify robot programming for industrial tasks, e.g., using natural language (Stenmark and Nugues 2013) and semantically defined parameterized motion primitives (Felip et al. 2013), often called skills (Pedersen et al. 2016, for example). Such semantic descriptions are desirable because they enable the automatic generation of standard PDDL1–descriptions for planning and scheduling of tasks or path planning and generation from virtual models (Perzlyo et al. 2016). However, in most cases these abstractions assume a library of skills or motion primitives that can be represented in executable robot controller code sequences, which can be applied in the high-level task description. Unfortunately, this assumption does not necessarily hold for highly specialized motions and configurations that might be needed in a specific assembly task.

In particular, dual-arm operations like synchronized motions or master-slave configurations, where one arm’s movements are specified in relation to the other one’s, are quite complex to program, as often (even in originally dual-arm configurations like the ABB YuMi) the two arms are controlled as two independent robots. This means that synchronization for bi-manual manipulation must be handled manually through the necessary instructions in each of the separate robot arm programs, which encapsulates the knowledge about the synchronization implicitly in the programs, but does not make it explicit, neither to operators different from the current programmer nor for an underlying skill representation for skill re-use. As a consequence, it is in currently
available systems difficult to fine tune the positions of a single arm separately, and to debug synchronized motions in pairs (Stenmark et al. 2016). Re-use and adaptation of parts of the robot program that require essentially the same operations, but from a different arm’s or even switched masterslave perspective, are also rather complex, and suitable support for the operator that offers explicit access to the knowledge required for such operations is needed.

We have earlier reported on our work that establishes a middle ground between high-level instructing by demonstration and explicit programming by combining kinesthetic teaching and a multimodal user interface with parametrized skill representations to simplify robot programming, which allows also non-expert users to specify and semantically annotate their own skills (motion primitives and also more complex program sequences). These skills can then be re-used and adapted on a higher level of abstraction into more complex task descriptions (Stenmark, Haage, and Topp 2017; Stenmark et al. 2017b; 2017a), also allowing for certain operations to be specified relative to (adaptable) object reference frames. Although the reported experiments and user study were conducted on the dual-arm robot YuMi shown in fig. 1, they did not cover extensive handling of synchronized motions for genuine dual-arm operations.

In this paper, we can report on our successful and to our knowledge novel efforts to integrate also options for the specification of synchronization points, synchronized motions and master-slave relations during program parts, which can then, based on our parameterizable skill representation, be re-used and adapted into high-level instructions.

This adaptation includes also the transfer of a skill (motion primitive) from one arm to the other, adaptation of synchronized motions to new object reference frames, and the switch of the master-slave relationship. Our approach supports the user by handling the necessary geometrical transformations implicitly based on the high-level instruction transfers expressed by the programmer through the graphical interface. At the same time, the knowledge about the synchronization of movements or transfer of operations is made explicit in the visualization of the program and can also be captured in the description of created skills for later re-use.

The remainder of the paper is organized as follows. We will refer to related work in the area of dual-arm robot programming in section 2. In section 3 we describe our skill representation and the conceptual framework for handling the re-use and parameterization of synchronized motions. We explain our implementation and experiments in sections 4 and 5, respectively, and discuss our results and conclusions in section 6.

2 Related Work

The dual-armed industrial system used in this article is programmed as two independent arms using two programs, one for each arm. Cooperation between arms is achieved through synchronization mechanisms that coordinate program execution, suggested already by (Lozano-Perez 1983), and made available in standard multi-arm robot controllers (ABB 2017). This perspective on multi-arm motion is common at the operator shop-floor programming level. Other industrial robot brands offer similar functionality. Not only robots offer this operator perspective. For instance, PLCopen offers similar concepts of coordinated motion for the CNC community (PLCopen 2016).

At the system level, coordinated motion is achieved by allowing one controller to control several robots, or by allowing several controllers to share trajectory information at a fast system level, utilizing interfaces such as (Blomdell et al. 2005). The ability to modify trajectories at a low level forms the basis for implementation of motion primitives that need to keep relations and/or constraints intact, possibly through sensor measurements, such as the modeling, control and planning of dual-arm motion surveyed in (Smith et al. 2012), or motion formalisms such as (Ijspeert, Nakanishi, and Schaal 2002), (Smits et al. 2008). A recent motion primitive example to be used in a programming-by-demonstration system is given in (Almeida and Karayiannis 2016).

Little or no semantic meaning of motion is typically retained in shop-floor programming interfaces. With “plugged in” motion primitives it is even more difficult to adjust and re-use programs. Another concern is that so far only few programs are programmed from scratch. Source generated, such as from programming-by-demonstration systems or computer-aided manufacturing tools, may need adjustment on the shop-floor using other tools than the original and should thus be understandable for tools and/or operators. Capturing and retaining the meaning of motion in the controller is a key for good shop-floor operator interfaces. Some recent efforts in semantic modeling of tasks, motions and/or actions are reported in, e.g., (Yang et al. 2015), (Wörgötter et al. 2013), (Aksoy et al. 2017), (Perzylo et al. 2015), (Perzylo et al. 2016), and (Ramirez-Amaro, Dean-Leon, and Cheng 2015). A classification of dual-arm manipulation tasks is presented in (Zöllner, Asfour, and Dillmann 2004). Only slightly related to motion, but an interesting approach is the building of semantic product memories (Wählster 2013).

The possibility to intuitively program robots is important to shop-floor interfaces. Of concern is that different modalities may only be good at conveying some specific information. A choice of different modalities may
need to be offered to provide a good operator experience also under varying conditions. An overview of interaction modalities and operator-in-the-loop concerns is given in (Tsarouchi, Makris, and Chryssolouris 2016). The authors of the overview also report on dual arm programming experience in assembly operations (Makris et al. 2014).

Most (industrial) robot programs are expressed in a robot programming language, and it is natural to work with source code in terms of program transformations. For example, a simple transformation supported by most controllers on the shop-floor is to translate a robot program in space to allow for quick adjustment to a new placement of a work piece. However, to the authors’ knowledge there is little work to be found in this direction, in particular regarding the capture of semantics or specific task related knowledge that is needed to adhere to specific task constraints also after the application of a transformation. Currently, the mainly supported programming paradigm seems to be generative, i.e., robot programs are generated for each specific task, supported by tools for, e.g., planning, learning, and demonstration.

Our approach allows us to work towards filling this gap, as we keep all relevant information about a task in the respective description of skills, which can then be re-used. This allows a user to specify both program and task transformations for dual-arm operations in a relatively intuitive way. The following sections describe our efforts to represent skills in terms of motion primitives and work objects as well as our implementation of a graphical user interface supporting this underlying representation.

3 Objects, Primitives, and Skills
The knowledge of the system is centered around physical objects (e.g., workpieces), devices such as manipulators, grippers and sensors, and robot skills used to realize assembly processes and implemented by robot programs. Parametrized skills form a hierarchy, with primitives in its bottom.

In this experimental setup, objects are created as abstract entities with a unique name, a type, and positions (called object frames) describing places on the object that may be used later as origins of relative reference coordinate systems. For example, the gift box used in the experiments described in Section 5 has four such points defined, one for the left side, one for the right sides and two for the corners closest to the camera. Robot motions can then be specified using those reference coordinate frames and the actual positions can be updated, e.g., using cameras. Every robot manipulator has a specific reference frame attached to the wrist, the flange. The gripper is attached to the flange and may have in turn one or more reference points, e.g., the tips of the fingers. For our experiments, we use a dual-arm robot where motions of one arm may be also specified in the other arm’s flange reference system. The low-level implementation in this setup does not allow arbitrary reference points along the robot arms, thus, contact situations where the so-called elbow or lower arm are used need to be specified using joint angles.

The re-usable robot programs, i.e., robot skills, have two components. The first is a high-level step-by-step instruction of how to achieve a goal. The skills can be nested into hierarchical (compound) skills and the lowest semantically described step comprises atomic actions called primitives. The primitives must have a mapping to some executable code on the robot system, which is the second component of the skill. The current implementation generates ABB RAPID, which is the native code executing on an ABB robot controller.

3.1 Primitives
Each primitive has a set of semantically annotated parameter types with (system dependent) default values. The semantics is captured in an industrial robot ontology (Stenmark and Malec 2015), however the current experimental system does not exploit it, but rather embeds the reasoning in the native code of the GUI implementation. The following primitives are implemented in the interface described in Section 4:

- **Motion types**: Motion has subtypes Free Motion and Contact Motion. Free Motion has subtype AbsJointMove (where the target is expressed in joint angles), LinMove, Circular Move and Joint Move where the target is expressed in Cartesian space relative to a point on an object and the path will be linear, circular or obtained by moving all joints simultaneously, respectively. The Contact Motion is a simple guarded search, which moves towards a specified point until the desired (estimated) contact force is reached, or, if no contact occurs, an error is thrown.

- **Gripper Actions**: They belong to one of the subtypes Open, Close and Finger Position. The first two actions open and close the fingers fully or until the current value of the gripping force is reached. The Finger action is used to set the gripping force and finger positions to the specified parameter values.

- **Synchronization point**: There are two types of synchronization in the system, either a synchronization point or synchronized motions. By default, the two arms execute independently as two separate robots, but a synchronization point is a rendez-vous between the robot arm programs that will force one arm to wait until the other reaches the corresponding point in its program.

- **Synchronized motions**: The two arms will start and finish the motions at the same time, that is, the longer motion will determine the execution time of both motions. The motions can be specified in absolute joint (or Cartesian) coordinates, or relative to objects in the workspace. They can also be specified relative to the position of the other arm, in a so-called Master-Slave configuration. The master-slave motions are normally used to lock a specific offset between the arms, e.g., when moving an object bi-manually.

- **Native Code**: The (advanced) user can add any native robot code in plain text as a snippet that will be included inline at a given point of a program. Only object references are then analyzed semantically. This action is used by the system integrator or an expert to create skill macros for sensor communication, e.g., by creating a function that uses cameras to locate an object.
For example, if the user added an object named `screw` and the expert created a procedure that locates a screw called `screwLocatingFunction`, the position of the screw is updated with the native code line `screw := screwLocatingFunction`. Since the variable names are unique and typed both in native code and in the user interface, object references can be changed using the interface, hence, non-expert users do not have to edit the native code to update the object positions of other (similar) screws.

### 3.2 Re-use and transformation of skills

Reuse of the available skills requires careful reasoning about the task parameters and its actual realization by the robot arms. There is a number of theoretically possible parameter transformations making sense in practice. We have investigated some of them, implementing the reasoner inside the experimental interface.

Motions expressed relative to object (or robot) reference coordinate systems can be transformed either by updating (editing) the coordinate system or changing it to another one. Using only one reference position, irrespectively of whether it is the flange of the robot or a frame placed on an object, by translating and rotating it one may define rotation of the skill. To mirror the task in a plane, two pairs of frames are needed, one origin frame for each arm (the one that is used during the programming phase), and then one new frame for each arm, respectively, to allow the skills for each arm to be rotated and translated arbitrarily in the space.

By parameterizing the skills with all necessary information (e.g., both Cartesian space coordinates and joint space values are captured in the description of every motion primitive), these transformations are possible to be handled automatically during re-use and refinement of a skill.

### 3.3 Programming synchronized motions

While related work, e.g., the app provided with the YuMi robot, lets the user create a pair of motions for both arms with one click, the motions are not semantically or even syntactically connected. With our representation, where synchronization points and synchronized motions have an explicit reference to the other member in the pair, they can be created, deleted, executed, moved in the program sequence and copied, in pairs. Hence, when moving a synchronized motion from one arm to the other, the other motion in the pair has to swap places with it in order to preserve a correct program. Having this knowledge expressed explicitly in the skill description, allows to support the user in handling the re-use and refinement of a previously programmed skill, e.g., by simply moving the necessary program parts according to the synchronization specifications and expressing this explicitly in a graphical interface.

Synchronized skills can also be re-used bi-manually and moved between the arms. An example is re-using dual-arm skills containing master-slave motions, where the skills for each arm are swapped after reloading the skill into a programming interface. For example, if the original skill had the left arm as master that moved in the reference coordinate system of object `objX`, and the right arm, the slave, had an offset `offsY` to the left arm, the right arm will become the master, moving to the same positions in the reference coordinate system of object `objY`, and the position of the left arm, now the slave, will be updated with the inverse of `offsY`.

A special case is the refactoring of reference frames for synchronized motions. Since the synchronization is an alignment in time, the reference frame of one arm can be changed to the flange of the other if the original positions are known, thus synchronized motions can be refactored into master-slave configurations.

All these programming concepts are implemented and can be visualized in the GUI prototype described in the following section. An overview of the supported skill transformations is listed in Table 1.

### 4 Implementation

The prototype design was developed after several case studies where the authors developed assembly applications on the ABB dual-arm robot YuMi. The user interaction is intended to simplify agile online robot programming. Mistakes should be easily corrected, with `programming` and `debug` modes merged into a single screen to facilitate a quick program modification and execution loop. All available information about the program is retained even though it may not be used immediately. As an example, a target position is saved with data for the current joint values, the Cartesian tool position and, if a reference object is selected, the relative position, which makes it easy to switch between representations. This allows the operator to quickly create a program and later work on creating and applying abstractions such as object references to make the program easier to re-use and adapt. The current graphical user interface is shown in fig. 3.

The current graphical user interface consists of two instruction lanes covering the right half of the screen, each displaying an action sequence for one of the YuMi arms. Each action can be executed individually using the play-arrow to the left of the name, the arm pose associated with the action can be updated with a single click, the parameters for each action can be edited, and an expandable icon contains...
Transformation | Meaning | Comment
--- | --- | ---
Arm exchange | The arms switch roles in the task | In the interface, the program sequence from one swim lane is simply copied into the other one and vice versa.
Arm exchange with constraint | Switch roles and uphold extra constraints | See, e.g., elbow constraint. Specific parameters, constraints, are saved into a skill description instance for later re-use.
Move synchronization points | Change when/where the switch between the independent and the dependent move occurs | Synchronization points are indicated (handled) in pairs along the program sequences in the swim lanes of the interface.
Master-slave introduction | Lock motion to one arm (master) with the other arm following (slave) | The tool center point of the master arm is handled as reference frame for the program of the slave.
Master-slave exchange | Switch master-slave roles | Involves very often also a switch of roles in the task, i.e., an arm exchange (see above) in the interface.
Transformation of task | The task is transformed with derived arm roles | Currently roles are assumed (not derived) based on type of transformation. User has to specify transformation by pointing out new reference points.

Table 1: Transformations assisting dual-arm programming in the interface. Not all fully implemented at the time of writing.

The interface supports the writing and debugging of synchronized motions by handling their creation, reordering, deletion and execution in pairs, expressing the relations between different synchronization points explicitly with graphical synchronization markers and colour codings to show, for example, a master-slave relation between the arms.

Reference objects can be selected in absolute (world) coordinates, relative to one of the robot hands (flanges), or in any user-created object. The graphical user interface has two templates for synchronized motions, namely Sync moves that just sets the timing and Master-slave motions where one arm follows the other. The latter has a wizard to assist the user in selecting the master arm and set the references of the other arm to the correct flange. The current offset is calculated and set as a target position for the slave arm. Then the same position can be copied to create a series of motions where the slave arm follows the master.

The robot program shown in Fig. 3 is an example of multiple synchronized motions. The operator has specified an object reference system, myObject (orange) by pointing with the robot gripper. The first four motions on each arm are synchronized master-slave motions where the left robot arm moves in the myObject reference system, while the right robot arm keeps a fixed offset to the left hand, thus following the left arm. Each action has a color code matching the color of the reference object. The motions were programmed by creating four pairs of synchronized master-slave motions with the desired offset and then teaching the positions of the left arm by updating the positions. The last two motions of each arm are also synchronized motions, but this time, the left arm moves in absolute coordinates while the right moves relative to a user-specified object myObject.

5 Experiments
Two programming experiments showing the applicability and effect of our approach were conducted on the YuMi robot. The first was a bi-manual pick and place of a lid of a toy box that illustrates the master-slave introduction and master-slave exchange transformations listed in table 1. With the second experiment, which is a sub-procedure of a gift-wrapping application, a combination of task transformation, in this case a mirroring operation, and an arm exchange under constraints (see table 1 proves to be much easier to accomplish than this would have been the case with the original YuMi programming tools.

5.1 Bi-manual object manipulation
The first setup is depicted in fig. 4 and the robot was programmed to carry out a bimanual pick and place where a lid of the toy box was picked up and inserted on the box. We are aware that this operation is probably not necessary to be carried out using two arms with the very lightweight object, however, it is a good example for a two-arm configuration under constraints stemming from the shape of the manipulated object.

Programming robot (assembly) tasks is two-fold, as the first crucial step is to find a good strategy of how to use the robot’s physical capabilities to solve the problem at hand and, in the second step, develop a consistent program. We assumed thus, that measuring the time it would take an operator familiar with the robot and RAPID programming, but not familiar with the task and the interface features for the
synchronization, to solve this task, and compare them to the same person simply programming the same task using native code (RAPID) would give us a good indication of the effect of our approach.

A good strategy to solve this particular task was to place the grippers in the holes of the lid using half-opened fingers, then create a sequence of four master-slave motions to lift the lid and place it on top of the box. The respective program is shown in fig. 3. A robot programmer who had never used the tool prototype for synchronized motions and was presented with the task for the first time, needed 32 minutes to solve and debug the task fully. This involved finding a robust gripping strategy with half-opened grippers that did not subject the lid to high forces.

After this first test, the robot programmer was joined by another expert programmer. These two experts (actually two of the authors of this paper) needed 5 and 10 minutes, respectively, to program the task with our interface, but 21 and 32 minutes, respectively, using traditional tools (text editor and teach pendant). Prototyping from scratch using traditional tools was not tested, since the strategy was known after the first step.

Further experiments were used to test the representation. The lid was rotated 90 degrees as depicted in fig. 5a, which caused the robot arms to collide during the pickup. Using the respective features of the interface, the operator could swap the programs for the arms, thus applying the master-slave exchange transformation, which resulted in a collision free pickup shown in fig. 5b.

However, attempting to attach the lid on the box also caused a self-collision. The insertion position was rotated 180 degrees by adding and changing references of the original program that used absolute coordinates for the insertion. Two new object references were added by pointing at the center of the insertion point using the gripper as shown in fig. 6b, one rotated 180 degrees around the z-axis. Using these two options for recalculating the reference system and changing the object of the task, the reference system was changed from absolute coordinates to positions relative to the original position. After changing of the object reference, the resulting program carried out the same task but rotated and, again, with interchanged master-slave motions, as shown in fig. 6a.

5.2 Mirroring tasks

In the second experiment, parts of a gift-wrapping application (previously described in (Stenmark et al. 2016)) were re-implemented, see fig. 1. Folding the sides of the gift paper around the box means to carry out symmetric tasks: first the left robot arm holds the box while the right arm folds the paper around the corners to make a straight fold, which is then swapped to the box being held by the right arm and the folding being carried out with the left. The complete process involves flattening of the paper and taping, but to test the skill transformations we focused on the straightening of the corners alone.

First, one reference frame (point) was added on the right (for the robot) side of the robot. The task was first programmed using this side point as reference frame.

We then applied a task transformation based on a rotation to re-use the program for the folding operations on the left side of the box, i.e., a second reference frame was added, this
time on the left side pointing outward. The robot programs for the arms were swapped (performing an arm exchange under constraints, i.e., keeping the elbow configuration as a crucial parameter) with the new reference object for the right arm being changed to the new side point (reference frame), resulting in a rotation of the task around the object (box) center, as shown in fig. 7a, i.e., a task transformation according to table 1.

We also applied another task transformation, i.e., a mirroring of the original task from one side of the box to the other. This was done by adding a point on the corner where the right arm is pointing in fig. 1 as well as a point in the mirrored corner (fig. 7b) using the left arm. The programs for the arms were swapped but this time the folding hand used the (robot) left corner point as reference while the holding hand used the side point reference frame, which resulted in a mirrored instead of rotated task execution.

For the time being, these operations are not explicitly available in the implementation, i.e., it is not possible to just press a button for “mirroring”, but with our experimental setup we could verify that the available tools and underlying representations and geometrical transforms allow to specify such a task transformation sufficiently well by just pointing out a couple of new reference frames (corner points) after the switching of program sequences to the other side.

6 Conclusions

In this paper, we discussed our efforts to improve intuitive dual-arm programming of collaborative industrial robots. Our programming model allows for primitive dual-arm motion constructs with associated motion constraints to capture specification of synchronization points, synchronized motions and master-slave relations during program parts. We also present a taxonomy of our programming constructs as a starting point for a dual-arm ontology of programming constructs. This is combined with a programming interface that supports a hybrid instruction and evaluation (execution) process and thus quick implementation and test cycles through a graphical interface for iconic programming. The interface supports also the specification of objects and their positions in the work cell of the robot, as well as updating these at any time. It is also possible to reorder instructions graphically and sequences of instructions can be saved as a skill for later re-use and refinement.

We have identified several program transformations, including rotations and mirror operations, that are useful for dual-arm programming relating to the inherent relationship between the arms. These transformations are exemplified in this paper through two transformation constructs, the mirroring transformation which allows a program written for a left-right arm configuration to be transformed into a program to be executed on a right-left arm configuration, and the master-slave switching transformation which allows a change of arm leadership role during “follow me” operations.

Experiments confirming the applicability and efficiency of our approach were conducted on two small part assembly scenarios, collaborative placing of a lid on a box, testing dual-arm programming with motion constraints, and gift wrapping, testing dual-arm program transformations. We could observe a significant reduction (by two thirds) of programming time compared to applying traditional approaches and tools to the programming, particularly when the solution to the task at hand was known, i.e., the programmer knew how the robot arms should move and “only” had to program them to do that.

However, our experience from the experiments indicates that further work on modes for dual-arm lead-through taking into account motion constraints might be a fruitful direction of investigation, perhaps in combination with work in the force-domain to allow expression of dual-arm constraints including torque and/or force conditions, i.e., applying a certain press on the sides of a box while picking it up. Standard functionality packaged with the industrial controller today includes only independent single-arm lead-through in the position domain. We also believe that investigation of guidance of non-experts / process experts in efficient robot programming techniques during the programming process would further improve programming efficiency, perhaps with robot-induced operator teaching as part of a robotic skill set.

Acknowledgment

The research leading to these results has received funding from the European Community’s Framework Programme Horizon 2020 under grant agreement No 644938 SARAFun.

References


