Manipulation Skill for Robotic Assembly

Roboterfertigkeiten für den Zusammenbau

Master-Thesis von Stefan Zeiß aus Heidelberg

Juni 2014

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Intelligent Autonomous Systems

Manipulation Skill for Robotic Assembly
Roboterfertigkeiten für den Zusammenbau

Vorgelegte Master-Thesis von Stefan Zeiß aus Heidelberg

1. Gutachten: Jan Peters
2. Gutachten: Hao Ding

Tag der Einreichung:

Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 01.07.2014

(Stefan Zeiß)

Preface

This Master-Thesis is the final thesis of the "Computational Engineering" Master's degree program at Technische Universität Darmstadt, which I attended from 2011 to 2014.
It was created in 2014 as a cooperation between the group "Intelligent Autonomous Systems" of the department of computer science at TU Darmstadt and the ABB Group. I wrote it at the ABB Corporate Research Center in Ladenburg, where it was part of a "Robotic Skill" project carried out at the "Robotics and Manufacturing" group of the research center's "Automation" department.
So far, one paper has been published in the context of this thesis at the IEEE International Conference on Automation Science and Engineering (IEEE CASE 2014), which describes our work in the area of "Contact Force Estimation". Another paper describing the overall concept is planned and will most likely be submitted at a different conference this year.
I would like to thank my university supervisor Professor Dr. Jan Peters for his great support and fruitful discussions. I am also greatly in debt to our project group at ABB, Dr. Hao Ding, Dr. Arne Wahrburg and Dr. Björn Matthias: Thank you for the great cooperation during the last months! Furhtermore, I would like to thank Ivan Lundberg of the ABB Corporate Research Center in Sweden for providing valuable information and support on the topic of "Contact Force Estimation". In the context of this topic I would also like to adress thanks to Andreas Stolt, Magnus Linderoth and Anders Robertsson for their inspiring work.
Furthermore, I would like to thank Allegra Pender for her great support and extensive spell-checking. Finally I would like to thank all the employees at ABB Ladenburg and especially my student colleagues there for the good time I had.
The full text of this thesis can be found on the attached CD as an PDF file.

Abstract

In this thesis, a new *Manipulation Skill* concept to fulfill robotic assembly tasks is proposed. A manipulation skill is an elementary primitive that encapsulates the capabilities to coordinate, control and supervise an elementary robot task. To gain reusability of a primitive in alike robot tasks, the primitives are represented as generic templates that are parametrized for each situation with data from an assembly specification. A skill is represented in two ways: as a 12-dimensional trajectory describing compliant motions in pose-wrench space and as a Finite State Machine to execute the motions that are derived from this trajectory. A qualitative node-based concept is proposed to store the trajectory in a generic way. Motions in the Finite State Machine are specified in the Task Frame Formalism. Mapping procedures are introduced to map parameters between the assembly specification, the Finite State Machine and the trajectory. A categorization of skills based on the proposed skill representation is presented, where the two skills *Insert* and *Snapfit* are investigated in depth. A new assembly specification is introduced that incorporates descriptive information about an assembly as well as instructions on how to perform an assembly. The applicability of the system to standard industrial hardware is shown. For control, only standard position control is employed while force/torque information is used to switch between different position-controlled motions. To circumvent the need for dedicated force/torque sensors, a novel approach to estimate forces from motor data is proposed. The concept is implemented as a demonstrator using an ABB dual-arm concept robot and standard ABB robot control software. Its applicability is shown by performing an example small part assembly featuring the skills *Insert* and *Snapfit* among others.

Zusammenfassung

In dieser Thesis wird ein neuartiges Konzept zur Durchführung von Montageaufgaben durch einen Roboter basierend auf "Manipulation Skill" vorgestellt. Ein "Manipulation Skill" ist ein primitiv, das die Fähigkeiten eines Roboters, eine elementare Aufgabe zu koordinieren, zu regeln und zu überwachen, enthält. Um Primitive für ähnliche Aufgaben wiederverwenden zu können, sind diese als generische Schablonen gespeichert, die durch Parametrisierung mit Daten einer Montagespezifikation an einzelne Situationen angepasst werden. Ein "Skill" wird auf zwei Arten gespeichert: als 12-dimensionale Trajektorie, welche Bewegungen anhand von Position, Orientierung, Kontaktkräften und Kontaktmomenten beschreibt, sowie als Zustandsautomat, welcher die Bewegungen durchführt, die von der Trajektorie abgeleitet werden. Ein qualitativer knotenbasierter Ansatz ermöglicht es, die Trajektorie auf generische Art und Weise zu speichern. Die Bewegungen im Zustandsautomat werden basierend auf dem "Task Frame Formalism" definiert. In der Thesis werden Mapping-Verfahren zum übertragen von Parametern zwischen den verschiedenen Repräsentation eingeführt. Darüber hinaus wird eine Kategorisierung von "Skills" vorgestellt, bei der die Skills "Insert" und "Snapfit" im Detail betrachtet werden. Es wird außerdem eine neue Spezifikation für Baugruppen vorgestellt, welche neben beschreibender Information auch Informationen über den Ablauf der Montage enthält. Letztlich wird die Nutzbarkeit des Systems auf industrieller Standard-Hardware gezeigt. Die Regelung von Bewegungen erfolgt anhand von Positionsregelung, während Kraft- und Momentinformation dazu verwendet wird, zwischen verschiedenen positionsbasierten Bewegungen zu wechseln. Um die Verwendung von Kraftsensoren zu umgehen, wird ein neuartiger Ansatz zum Abschätzen der Kräfte basierend auf Motordaten vorgestellt. Das Konzept wird als Demonstrator implementiert, wobei ein "ABB Dual-Arm Concept Robot" sowie Standard-ABB-Software zum Einsatz kommen. Der Demonstrator wird genutzt, um einen Zusammenbau von Kleinteilen durchzuführen, bei dem unter anderem die Skills "Insert" und "Snapfit" verwendet werden.

Contents

List of Figures

List of Tables

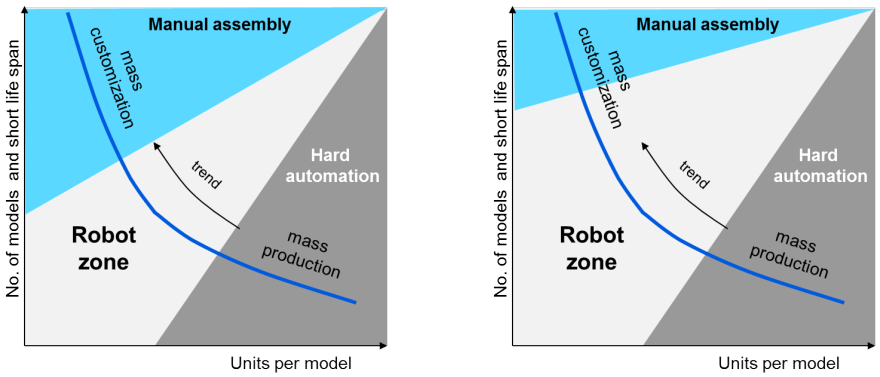
List of Symbols and Abbreviations

q	Joint angle
$\dot{q}$	Angular joint velocity
$\ddot{q}$	Angular joint acceleration
J	Robot Jacobian
p	Pose vector
p_x, p_y, p_z	Position coordinates in x,y, and z-direction
ϕ_x, ϕ_y, ϕ_z	Rotations about the x,y, and z axis described by ZYX Euler angles
f	Wrench vector
F_x, F_y, F_z	Acting forces in x,y, and z-direction
τ_x, τ_y, τ_z	Acting torques about x,y, and z-direction
μ	Friction coefficient
I	Identity matrix
P^+	Moore-Penrose-Inverse of P
$\mathrm{X} \succeq 0$	Positive semi-definiteness of matrix X
TFF	Task Frame Formalism
FSM	Finite State Machine
PCB	Printed Circuit Board
$DACR$	Dual-Arm Concept Robot
SVM	Support Vector Machine
DOF	Degree of Freedom
TCP	Tool Center Point
DH	Denavit Hartenberg

1 Introduction

1.1 Motivation

In today's industrial manufacturing robots are only economically worthwhile in a limited set of production scenarios. Their applicability mainly depends on the production scale.
The production scale differs between mass production (very large scale) and customization (very small scale). In large scale production a limited number of models is produced in very large amounts over a long period of time. Here it makes sense to use hard automation where highly specialized and inflexible manufacturing devices are employed instead of robots. The opposite case is customized manufacturing where a large variety of products is produced in smaller amounts and short life spans. Manufacturing on this scale is carried out mostly manually. The production scale region in between these extreme scenarios where robots are applicable is called the "Robot Zone", which is shown in Figure 1.1.
In today's manufacturing, there is a paradigm shift in manufacturing from mass production towards mass customization [5], which increases the demand for flexible production systems, as the production line output changes more frequently in its shape and number. Robotic systems have to adapt to these new demands. The main reason why today's industrial robots do not meet these demands is their lack of adaptivity to new situations. It is very complicated to program a robot to fulfill a new task, even if it is only slightly different from the previous task. To reprogram a robot, experts with deep technical knowledge of the domain and a lot of time are needed. Furthermore, standard industrial robots have only very limited capabilities to cope with uncertainties. To circumvent this drawback, the robot's environment has to be specifically and carefully structured for each task. It is obvious that this is difficult in a constantly changing production scenario. Another disadvantage of today's industrial robots is the lack of collaborative capabilities, as robots cannot react to humans in their environment.
A new approach in design and control of robots is necessary to push the border of the "Robot Zone" towards the domain of manual manufacturing. If robots should ever be used in more flexible production scenarios, the setup and programming efforts must be decreased while the cooperation capabilities must be increased. With the *Manipulation Skill* system we propose a system that is supposed to reduce the setup time and simplify the programming by including a set of reusable robot capability packages called "Skills". To facilitate collaborative behavior, these skills can incorporate different sensing and control possibilities.



(a) Current Robot Zone (b) Desired Robot Zone

Figure 1.1: The applicability of robots for production scenarios visualized by the "Robot Zone". The blue line represents common production scenarios as a function of the produced units per model, the model variety and the life span of a model. [57]

1.2 Problem Statement

To use robots in flexible production scenarios (see Section 1.1), like for example small part assembly, two main problems have to be solved:

- Robots need to be able to operate in partly unstructured environments under the presence of uncertainty.
- Robots need to be quickly adaptable quickly to react to frequently changing production scenarios.

A robot that can operate under the presence of uncertainty demands complex sensor-based control strategies. To adapt a robot quickly, it has to be possible to reprogram them quickly, preferably by non-experts on the shopfloor. As the former requirement increases the programming complexity, it is obvious that the solution to the two problems contradicts.
An approach to solve this contradiction is the concept of *Manipulation Skill*. The general idea is to assemble complex robot actions from elementary building blocks called *Skills*. A skill encapsulates the capabilities to coordinate, control and supervise an elementary robot action. All details of the robot control, which can be as complex as force control, are hidden from the programmer who just needs to assemble predefined skill primitives. To be useful, a skill needs to be reusable in alike situations. Otherwise, each new situation would demand the programming of a new skill primitive.
Such a concept is proposed in this thesis. To create a *Manipulation Skill* system and to use it, several requirements have to be met. Firstly, it is necessary to find a skill representation that has the flexibility to cope with uncertainties, is easy to use and can be reused in alike situations. Secondly, a framework needs to be created in which the skill representation can be embedded and used for robot control. This includes the creation of interfaces to robot control and sensing, as well as an interfaces to high-level applications, for example assembly planning. Finally, the system needs to be implemented to prove its usability.

1.3 Outline

This thesis is structured as follows: after the previous chapters provided an introduction to the topic, Part I summarizes the state of the art of *Manipulation Skill for Robotic Assembly*. The part firstly presents the individual requirements and components necessary for manipulation skill and proceeds to presenting different approaches to manipulation skill. Afterwards, these approaches are compared and the contributions of this thesis to the manipulation skill approach are presented.
Part II presents the proposed concept. In this part, first the overall concept is presented with its different elements. Then, the representation of manipulation skill that is embedded in this system is described. At the end of this part, Chapter 8 presents different example skills with respect to the proposed skill representation.
Part III presents the practical implementation of the proposed concept. It is firstly described how the demonstrator was implemented. The concept of contact force estimation, which is necessary to use the concept in practice, is presented in Chapter 8. Chapter 9 describes the robotic workstation and Chapter 12 reveals the results derived from using the implemented demonstrator to perform an example assembly on the described workstation.
Part IV summarizes and discusses the work and provides an outlook to potential extension and future development of the approach.

Part I
State of the Art

2 Robotic Assembly

This chapter presents an overview of the state of the art in robotic assembly, which is the main application area for the concept presented in this thesis.

2.1 Specification and Requirements of Robotic Assembly

Robotic Assembly can be seen as a subset of robotic manipulation tasks. Such tasks describe a process where objects are moved and rearranged in the environment by a manipulator [47]. During the motion several contact situations between the involved objects and the environment can occur. From this definition a great number of applications can be derived. Besides industrial assembly, tasks involving manipulation can also be found in everyday life. Some examples are building a card house, opening a door or pouring a glass of water [47] [72]. The focus of this thesis is on industrial small part assembly tasks like assembling a cell phone.
Historically, humans are in the role of the manipulator. With thousands of sensors and actuators as well as the intelligence to coordinate and adapt them to new situations, humans are perfectly capable of performing manipulation tasks [47]. Another important factor is that humans have the knowledge of what actions are needed to fulfill a certain task due to their experience. Furthermore, the human hand is a highly dexterous tool that is a perfect fit for nearly all manipulation tasks.
When manipulation is supposed to be automated, which is desired in industry and increasingly also in home and service environments, robotic systems can be employed. For robots manipulation tasks are very challenging, since human sensory, motor and combinatorial capabilities have to be emulated. Several requirements have to be met to make a robot suitable for manipulation and especially assembly tasks.
These requirements can be described as a mixture of planning and mechanics problems [47]. At first, suitable tools are needed to enable interaction with the objects. Then, a sequence of object motions and other actions that lead to successful completion of the manipulation task needs to be found [62, Section 26.1]. As these motions are at least partially performed in contact with the environment, contact situations need to be supervised. For this supervision sensing is necessary. Depending on the task, this can, for example, mean to measure external forces acting on the objects and adapt the motions or the strategy accordingly. Another possibility is visual feedback. Finally, it needs to be determined if the manipulation task has been performed successfully. For this purpose also a combination of sensing and prior situational knowledge is necessary. To orchestrate all of these components, a high-level coordination is necessary. The following list based on [62, Section 26.1] summarizes the requirements:

Tooling As we do not have a "perfect tool" like the human hand in robotics, we need appropriate robot end-effectors that are usually designed specifically for one task or one class of tasks. In most manipulation tasks the applied tool is a gripper to pick up and hold objects. Especially in unstructured environments where objects have uncertain orientations, this can be a difficult task, and high dexterity is needed in order to fulfill it. There is a wide variety of grippers reaching from simple open/close clamps to state-of-the-art robot hands. A detailed description of tools is out of the scope of this thesis.

Motion Planning Finding an action sequence that leads to successful completion of a task is a non-trivial problem in robotics. Its foundation are accurate models of the environment as well as the robot itself. Traditionally, the problem is divided into gross and fine motion planning. While the former is used to find a macroscopic motion strategy, the latter tries to deal with the environment uncertainties. How an assembly sequence can be planned and represented is described in Section 2.2.

Motion Specification To specify the motions involving contact with an uncertain environment, mostly kinematic and dynamic contact constraints are used. Different approaches for motion specification are described in Sections 3.1 and 3.2.

Control To satisfy these the constraints, appropriate robot motions have to be executed. Therefore, different feedback control loops can be used. For their use, various forms of sensing may be necessary. Control strategies can include different mixtures of control and sensing of robot kinematics, static and dynamic forces [47, Section 1.4]. For a more detailed decription refer to Section 3.3.

Sensing Examples for applied sensing are force/torque sensing or optical sensing. The sensing can be used directly in a feedback control strategy or in higher-level supervision of the task execution. Different sensing possibilities are presented in Section 3.4.

Coordination A way to coordinate the execution of assembly tasks is the concept of robotic skills. This is described in detail in Section 4.

2.2 Automatic Assembly Sequence Planning

The goal of assembly planning is to find a sequence of robot actions that accomplishes a certain assembly task. In general, assembly planning can be described as an extension of the basic motion planning problem.
The problem can be solved by applying two consecutive steps. Firstly, an assembly plan has to be created consisting of a sequence of steps to assemble a part. In this step, only the geometry of the part and not the robot is considered. Secondly, a sequence of robot motions to fulfill the assembly steps has to be found. For the latter, it is common to use motion planning by calculating configuration space obstacles (see Section 3.1).
To create an assembly plan, three classes of approaches exist: human interaction, geometry-based reasoning and knowledge-based reasoning [18].
The first class contains early approaches where the user is queried for information and assembly sequences are created according to the answers [21]. When the goal is to automate assembly processes, such approaches are not feasible.
The second class contains approaches to create assembly plans automatically from geometrical part representations like CAD data. Most of these geometry-based approaches employ the basic idea of "assembly by disassembly". Following this strategy, ways to partition an assembled part into sub-assemblies are searched. This way, a (reverse) order of assembly steps is obtained. Possible ways to disassemble a part can be found via geometric reasoning (e.g. collision testing) [82]. The strategy is, for example, used in [69] to create "disassembly trees" by stepwise removal of sub-assemblies via single-step translations. Representing assembly sequences in tree structures is widely used [40]. Another possible representation of assembly sequences are AND/OR graphs [28], which contain all feasible sequences. The optimal sequence is selected by analyzing the arcs of the graph with respect to a performance criterion.
The third class includes advanced planning approaches that also incorporate non-geometric information. This non-geometric information can be seen as high-level expert knowledge or experience [18]. By augmenting assembly plans with situational knowledge, it becomes possible to reuse them or sub-plans of them. An example for stored assembly knowledge is hierarchy knowledge as used in [86]. In general it is possible to reduce the computational complexity drastically by integrating knowledge and geometry-based reasoning [18]. New plans only have to be created if no predefined plans for reuse can be found based on the provided knowledge.
There are various ways to implement assembly planning systems. In [32] a system is presented that allows the generation of an assembly plan directly from CAD data. This also includes robot motion planning, as executable robot code is created.

3 Compliant Behavior

This chapter presents various approaches for specifying and controlling motions in contact with the environment. Furthermore, possibilities to incorporate sensing in such approaches are presented.

3.1 Active and Passive Compliant Motion

To perform an assembly task, a robot typically has to move parts it holds with its manipulator to a specified goal assembly state. The goal state and the path to reach it can either be defined by required spatial arrangements or contact states relative to the environment or other parts involved. To describe and execute such motions in the presence of uncertainty, the concept of *Compliant Motion* can be used. It is defined as "motion constrained by the contact between the held part and another part in the environment" [62, Section 26.4].
There are two classes of approaches to execute compliant motions, which are described in the following: *Passive Compliant Motion* and *Active Compliant Motion*.

Passive Compliant Motion

In passive compliant motion no active recognition and control of contact states are necessary. Usually this means that the compliance problem is solved by inherent mechanical solutions. A simple example is the usage of chamfering cones in assembly.
The most standard way to achieve passive compliance is the concept of the *Remote Center of Compliance* (RCC), which has been developed in the 1970s by Whitney [81]. It has first been employed for a peg-in-hole assembly task.
The RCC is a mechanical spring structure that is attached to the robot between its wrist and its tool. This structure introduces a high lateral and angular compliance to the system while maintaining a high stiffness in longitudinal direction. This shifts the center of compliance to a position near the tip of the part held by the tool and changes the way the held part reacts to contact forces induced by the environment. This enables the part to move compliantly according to the forces and to react to small position and orientation errors in the motion path of the part. [62, Section 26.4.2]
RCC has a broad field of application and is especially used in peg-in-hole tasks (Figure 3.1). Reasons for its success are its low cost and great reliability in specific cases. A drawback is that RCC devices have to be specifically designed for each application and only work for simple shapes. Furthermore, the positioning errors it can handle have to be small.

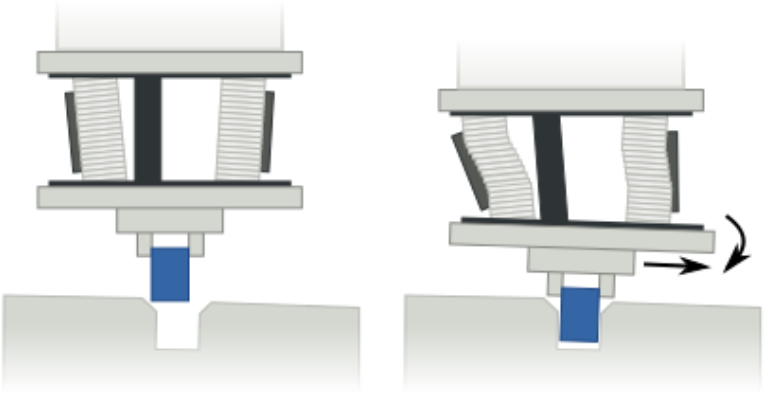


Figure 3.1: A Remote Center of Compliance (RCC) device used in peg-in-hole assembly [2]

Active Compliant Motion

While passive compliance works effectively for very specific applications, active compliant motion provides the flexibility to solve a wider variety of problems. In active compliant motion the robot adapts its control based on recognized contact states. This is achieved by using force-based feedback control. To implement such a system, the following components are necessary according to [41]:

- A **model** to describe contact states
- A **planner** to plan compliant motion commands
- An **identifier** to identify contact states
- A **controller** to execute the motion commands

Topological contact state **model** representations describe contact states as a set of topological contact primitives. These primitives can be point contacts (e.g. vertex-to-face) like in [45] or contacts between surface elements like in [16]. Models like this have the drawback that they are hard to distinguish due to uncertainties. Another kind of topological contact primitive, called *principal contacts* (PC), was introduced by Xiao [84]. They are high-level descriptions of contacts between polyhedral objects. PCs are suitable for automatic generation as proposed in [54].
Another contact state modeling approach is based on specifying controller setpoints in the *Task Frame*. This approach is, for example, described in [11] and will be discussed in more detail in Chapter 3.2.1.
The last modeling approach discussed here is based on geometric constraints that are specified among features. Features are geometrical elements like edges, vertices or faces. This approach is, for example, employed in [60] and will be described in more detail in Chapter 3.2.2.
To provide motion **planning** strategies that work in the presence of uncertainty, the concept of pre-images in configuration space (C-space) was introduced [45]. A pre-image encodes all configurations in C-space from which a desired goal state can be reached and recognized [62, Section 26.4.3]. For the computation of pre-images, the obstacles in C-space (C-obstacles) need to be given, especially when compliant motion is involved.
To avoid the expensive computation of C-obstacles, a frequently used approach is to plan compliant motions based on a predefined topological contact state graph [39]. Traditionally, the contact states in the graph are generated manually, which is very elaborate even for simple tasks [68].
Automatic contact graph generation was first done by Hirukawa et al. in [27]. Their approach is to enumerate all possible contact states and transitions between two convex polyhedra and store them in a graph. Afterwards, a path between an initial and a goal configuration is searched by solving algebraic equations. A more general divide-and-merge approach working on arbitrary polyhedra was proposed in [30].
In planning approaches that consider uncertainties, it is common to split the planning into two phases: global planning with no consideration of uncertainties and fine motion planning based on sensing and contact state analysis. This is, for example, followed in [85].
To execute predefined motion plans it is necessary to **identify** the current contact state online. In general, there are two classes of approaches: one that uses a history of previous executions of similar compliant motions and one that does not use this information.
If history data is available, it is convenient to use learning approaches to circumvent the need of explicit contact state modeling for identification. The data used for learning can, for example, be obtained by human task demonstrations. There are several ways to learn contact states. In [25] a hidden Markov model was used to learn wrench characteristics. The same characteristics can also be learned via a neural network structure [9].
When no training data is used, an explicit physical model is necessary where uncertainty is described by Probability Density Functions (PDFs). An example for such an approach is to match wrench measurements with predefined templates in a qualitative way [61].
The last necessary component of a compliant motion system is a **controller** to execute compliant motion commands. It receives desired setpoints from the planner and computes robot commands by comparing the setpoints to the measured current robot state. Appropriate control strategies for compliant motion are discussed in Chapter 3.3.

3.2 Specification of Compliant Motion Tasks

A robotic motion task can be specified by setting desired values, either on the joint level or the workspace of the robot. The latter is called *Task Level Programming*, which is easier if the task features compliant motion. To do so, motion tasks in compliant motion systems are programmed in specific frames called *Task Frames*. The basic concept to program such tasks is called the *Task Frame Formalism* (TFF). It was introduced by Bruyninckx [11] and is based on Mason's compliant motions [46]. An important extension to the TFF is *Constraint-Based Programming*, which extends the concept of TFF to multiple feature frames.

3.2.1 Task Frame Formalism

In general, task frames are described as "a set of orthogonal reference frames in which the task specification is very easy and intuitive" [62, Section 7.3.3]. Tasks are programmed by constraining the degrees of freedom of such a frame. Constraints can be imposed by specifying desired setpoints for each direction. These values can be forces, torques, linear velocities or angular velocities, and are also known as *artificial constraints* in the context of TFF. Additionally, there are *natural constraints* which are imposed by the environment [46]. There are six directions in the task frame that can be independently programmed: the X,Y, and Z axis with their respective rotational degrees of freedom [11].
To program the degrees of freedom of a task frame individually, the frame has to be described as a 6-dimensional pose vector. This way, every degree of freedom can be accessed and set individually in an easy way [37].
The commands specified in this formalism are high-level robot commands. To execute them, an appropriate hybrid low-level control system like hybrid force/motion control (see Chapter 3.3.2) is necessary [87].
TFF-based high-level commands have to be defined as atomic robot actions. Such atomic actions can conveniently be connected by a concept like the manipulation skill system presented in this thesis. An approach to use TFF in relation to manipulation primitives is, for example, presented in [37].
Research on the TFF formalism mainly focuses on related topics like hybrid control [87], assembly planning [70] or compliant motion [11]. More recent implementations featuring the TFF can be found in [38] and [72].
A drawback of the TFF is that it can only be applied to task geometries with limited complexity. The usage is only possible if it is sufficient to assign the control modes to three purely translational and three purely rotational directions of a single frame [13]. If this is not ensured, advanced approaches like the *Constraint-Based Task Specification* are available.

3.2.2 Constraint-Based Task Specification

The *Constraint-Based Task Specification* (CBS) is a more general extension of the TFF. It allows the assignment of different control modes in arbitrary directions of the workspace instead of just along and about the axes of a single task frame. This makes the approach applicable to more complex task geometries. To accomplish this, the task frame is replaced by multiple *Feature Frames*. In each of the feature frames a part of the whole task geometry is specified by axial constraints in the same way as done in the TFF [13].
In [3], geometric constraints between object features were used to derive a robot's assembly goal pose. For this purpose, a desired relative object pose is calculated from the geometric feature relations. An important foundation of the CBS is the task function approach introduced by Samson et al. [60]. This approach features the general idea that robot tasks can be described as a positioning problem. The control problem deriving from this specification is that of controlling a vector function that represents the task function. To execute the control, a general non-linear proportional control scheme has been introduced.
The approach presented in [13] uses feature and object frames to program a task. A task that is described by this concept can be specified as a desired relative interaction between two objects. To program such a task, feature and object frames are assigned to relevant features and objects by the user. In this context, an object is a real physical object in the robot workspace, while a feature is a physical entity (e.g. a vertex) or a geometric property (e.g. a

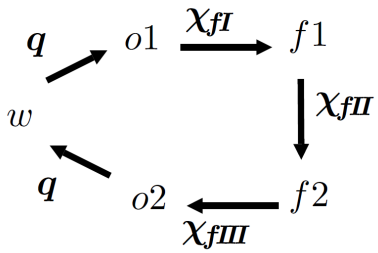


Figure 3.2: Closed Kinematic Chain that contains the feature frames f, the object frames o and the world frame w. The frames are related by using the feature coordinates χ_f and the joint values q [13].

symmetry axis) of such an object [13]. The object frames are rigidly connected to their respective objects while the feature frames are linked to the object frames. Relative interaction can now be defined by imposing constraints between two corresponding features on the objects. To define a constraint, in [13] a closed kinematic chain is specified. This kinematic chain contains four frames: two feature frames $\boldsymbol{f}_1$ and $\boldsymbol{f}_2$ as well as two object frames $\boldsymbol{o}_1$ and $\boldsymbol{o}_2$. Furthermore, the world frame $\boldsymbol{w}$ is used to close the kinematic chain, which is shown in Figure 3.3.
The advantage of specifying a kinematic chain is, that the degrees of freedom of the relative motion of the objects are split into three transformations [13]. Furthermore, each interaction task can be specified by multiple feature relationships, each expressed through a kinematic chain. This gives the approach a greater flexibility.
The total transformation between two objects is described by a 6-dimensional vector $\boldsymbol{\chi}_f$ of feature coordinates. This vector represents the closed kinematic chain as shown in Figure 3.3. Its content are position and orientation coordinates, described, for example, by Euler angles. As previously described, the total transformation is split into sub-transformations:

$$\boldsymbol{\chi}_f = (\boldsymbol{\chi}_{fI}^T \quad \boldsymbol{\chi}_{fII}^T \quad \boldsymbol{\chi}_{fIII}^T)^T \tag{3.1}$$

How these sub-transformations are distributed can be seen in Figure 3.3.
In addition to these coordinates, a second set called *uncertainty coordinates* specified by $\boldsymbol{\chi}_U$ is introduced in [13]. The uncertainty coordinates are used to represent pose uncertainties between the real and modeled version of each frame. The kinematic chain is extended by one additional frame and one transformation per existing frame, if this concept is used.
In this framework a task, is now specified by using feature relationships to constrain position-based system outputs $\boldsymbol{y}$. These system outputs represent the controlled variables of a robotic system and can, for example, be distances or contact forces. They are related to the task coordinates $\boldsymbol{\chi}_f$ by the output equation $\boldsymbol{f}(\boldsymbol{q},\boldsymbol{\chi}_f) = \boldsymbol{y}$, where $\boldsymbol{q}$ denotes joint values of the robotic system. Often, feature coordinates are chosen to simplify this relation to $\boldsymbol{\chi}_f = \boldsymbol{y}$. The system outputs are constrained by setting desired values $\boldsymbol{y}_d$.

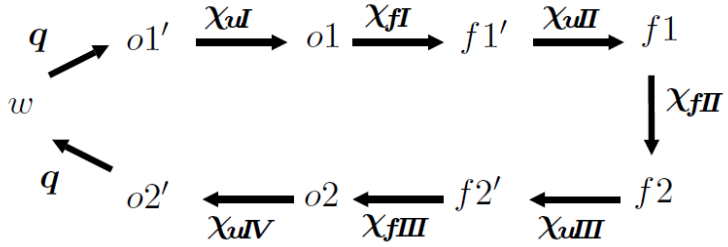


Figure 3.3: Closed Kinematic Chain that contains the feature frames f, the object frames o and the world frame w. The frames are related by using the feature coordinates χ_f, the uncertainty coordinates χ_u, and the joint values q [13].

3.3 Advanced Task-Level Robot Control Approaches

A standard industrial robot is moved with position control. As this approach is insufficient to execute compliant motion tasks where contact with the environment is relevant, more advanced robot control laws are necessary. The two most important classes of control laws for this kind of tasks are force control and vision-based control which are described in the following sections. As it is beneficial to describe these control laws in the operational space of the robot, the operational space formulation of the robotic dynamic model is presented in the first subsection.

3.3.1 Operational Space Formulation of the Robotic Dynamic Model

To control a robot in contact situations, positions, velocities or desired contact forces at contact points have to be specified. For robot control, desired values are set directly at these contact points in task space. If the values were instead set at the joint level, inverse kinematics would need to be calculated, which is computationally costly [62, Section 26.2].
To accomplish control at task level, also called operational space, it is necessary to express the robotic dynamic model in the operational space. The robotic dynamic model in joint space formulation can be written as

$$\boldsymbol{\tau} = \boldsymbol{H}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \boldsymbol{C}(\boldsymbol{q}, \dot{\boldsymbol{q}})\dot{\boldsymbol{q}} + \boldsymbol{\tau}_g(\boldsymbol{q}) + \boldsymbol{J}(\boldsymbol{q})^{\mathrm{T}} \boldsymbol{f}_{ext}, \tag{3.2}$$

where $\boldsymbol{q}$, $\dot{\boldsymbol{q}}$ and $\ddot{\boldsymbol{q}}$ are vectors of joint position, velocity and acceleration, while $\boldsymbol{\tau}$ specifies the torques acting on the joints. Each is an n-dimensional coordinate vector, where n is the number of (independent) joint variables in the mechanism. $\boldsymbol{f}_{ext}$ denotes an external force acting on the robot caused by contact with the environment. $\boldsymbol{H}$ is called the joint-space inertia matrix, and it is an $n \times n$ symmetric, positive-definite matrix. $\boldsymbol{C}$ is the matrix describing the Coriolis and centrifugal forces. Gravity terms are included in the vector $\boldsymbol{\tau}_g$ and the effect of the external force is given by $\boldsymbol{J}^{\mathrm{T}} \boldsymbol{f}_{ext}$. The matrix $\boldsymbol{J}$ is the Jacobian of the end effector.
To transform this formulation to the operational space, the following relations are necessary [62, Section 26.2.1]:

$$\dot{\boldsymbol{x}} = \boldsymbol{J}_x(\boldsymbol{q})\dot{\boldsymbol{q}} \tag{3.3}$$

$$\boldsymbol{\tau}_{\text{ext}}(\boldsymbol{q}) = \boldsymbol{J}^{\mathrm{T}}(\boldsymbol{q}) \cdot \boldsymbol{f}. \tag{3.4}$$

Equation 3.3 links the joint velocities to the velocities $\dot{\boldsymbol{x}}$ at the operation point. In equation 3.4 the instantaneous torques $\boldsymbol{\tau}_{\text{ext}}(\boldsymbol{q})$ at the joints are described by the external wrench

$$\boldsymbol{f} = \begin{bmatrix} F_x & F_y & F_z & \tau_x & \tau_y & \tau_z \end{bmatrix}^{\mathrm{T}}, \tag{3.5}$$

which is a vector containing contact forces F_x, F_y, and F_z, as well as contact torques τ_x, τ_y, and τ_z expressed in task coordinates.
Using these equations, Equation 3.2 can be mapped to the operational space formulation

$$\boldsymbol{\Lambda}(\boldsymbol{q})\ddot{\boldsymbol{x}} + \boldsymbol{\Gamma}(\boldsymbol{q}, \dot{\boldsymbol{q}})\dot{\boldsymbol{x}} + \boldsymbol{\eta}(\boldsymbol{q}) = \boldsymbol{f}, \tag{3.6}$$

where $\boldsymbol{\Lambda}$ specifies the operational space inertia matrix, $\boldsymbol{\Gamma}$ describes the Coriolis and centrifugal forces in operational space and $\boldsymbol{\eta}$ compensates gravity [62, Section 26.2.1].

3.3.2 Direct and Indirect Force Control

In force control, forces acting on the robot's end-effector are controlled. There are two kinds of force control strategies: indirect force control approaches control forces via motion control without closing the force feedback control loop, while direct approaches close the loop and allow forces to be explicitly controlled to a certain desired value. Members of the former class are, for example, *impedance control* and *admittance control*, while an example for the latter is *hybrid force/motion control*.

Impedance Control

In *impedance control*, the relation of the end-effector motion and the contact force is modeled through an equivalent mass-spring-damper system with adjustable parameters [62, Section 7.1.2]. If the robot deviates from its desired motion path, it generates forces according to this impedance to get back on the desired path. The displacement from the desired motion is the input of the path while the generated wrench is the output. A special case of this scheme is *stiffness control* where just the static relationship is considered [59]. Another variation is damping control where the relation between contact force and end-effector velocity is considered [80].
To describe the dynamic behavior of the end-effector, the acceleration-resolved motion control law [62, Section 7.2.2]

$$\boldsymbol{h}_c = \boldsymbol{\Lambda}(\boldsymbol{q})\boldsymbol{\alpha} + \boldsymbol{\Gamma}(\boldsymbol{q}, \dot{\boldsymbol{q}})\dot{\boldsymbol{q}} + \boldsymbol{h}_e \tag{3.7}$$

is employed. This equation is inserted into the operational space formulation of the robot dynamics 3.6, which results in

$$\dot{\boldsymbol{\nu}}_e = \boldsymbol{\alpha}. \tag{3.8}$$

Herein, $\boldsymbol{\alpha}$ is a control input describing a desired acceleration with respect to the end-effector frame. It is set to

$$\boldsymbol{\alpha} = \boldsymbol{K}_M^{-1}(\dot{\boldsymbol{\nu}}_d + \boldsymbol{K}_D \boldsymbol{\Delta\nu}_{de} + \boldsymbol{h}_\Delta - \boldsymbol{h}_e), \tag{3.9}$$

which results in the following equation describing the dynamic behavior of the end-effector as mechanical impedance.

$$\boldsymbol{K}_M \boldsymbol{\Delta}\dot{\boldsymbol{\nu}}_{de} + \boldsymbol{K}_D \boldsymbol{\Delta\nu}_{de} + \boldsymbol{h}_\Delta = \boldsymbol{h}_e \tag{3.10}$$

In this equation $\boldsymbol{\Delta}\dot{\boldsymbol{\nu}}_{de}$ and $\boldsymbol{\Delta\nu}_{de}$ denote the differences between end-effector velocity and acceleration and respective desired values. $\boldsymbol{h}_\Delta$ and $\boldsymbol{h}_e$ are the elastic and end-effector wrenches. $\boldsymbol{K}_M$ and $\boldsymbol{K}_D$ are 6x6 symmetric positive-definite matrices. The former describes the mechanical impedance, as it contains the mass and the inertia tensor, while the latter describes the damping.

Admittance Control

In admittance control, a measured force (input) is used to generate a motion path (output) which is related to the desired motion path. Contrary to impedance control, this scheme does not control the desired position and orientation of the end-effector but a reference position and orientation resulting from the impedance control action [62, Section 7.2.2]. This works by using the desired position and orientation as well as the measured wrench as input for the impedance control equation. By integrating this equation, a position and an orientation are generated which are used as input for an inner motion control loop. The decoupling of impedance and motion control has the advantage that force and position error, do not have a negative influence on each other.

Hybrid Force/Motion Control

The idea behind hybrid force/motion control is to split the control of a motion among the degrees of freedom of its reference frame. This way it is possible to simultaneously control contact forces and end-effector motion in independent subspaces [62, Section 7.1.2]. Usually, motion is controlled in the unconstrained directions of a task while force is controlled in the constrained directions. To allow this, it is necessary that a task frame is defined as a reference frame so that the task can be split into purely motion controlled and purely force controlled directions (see Chapter 3.2.1).

The selection of the appropriate control for each degree of freedom is done by selection matrices or more advanced projection matrices derived from explicitly constrained equations [87]. There are various implementations of hybrid control schemes available, which are, for example, based on inverse dynamics control in operational space [33], passivity-based control [75], or outer force control loops closed around inner motion loops [14].

The most common approach to design a hybrid force/motion control loop is to use an inverse dynamics control law [62, Section 7.1.2]. In this approach, the linearization and decoupling of force and motion subspaces of the nonlinear robot dynamics are handled by a model-based inner control loop. Furthermore, an outer control loop is used to control forces and motion values via separate control laws.

The foundation for the creation of a hybrid control loop is the description of an interaction task by a desired end-effector wrench $\boldsymbol{h_d}$ and a desired twist $\boldsymbol{\nu_d}$. To ensure that the desired values lie in separate subspaces, they are defined as

$$\boldsymbol{h}_d = \boldsymbol{S}_f \boldsymbol{\lambda}_d \tag{3.11}$$

$$\boldsymbol{\nu}_d = \boldsymbol{S}_\nu \boldsymbol{\nu}_d, \tag{3.12}$$

where $\boldsymbol{S_f}$ and $\boldsymbol{S}_\nu$ are task-specific selection matrices referring to the task frame (see Chapter 3.2.1). $\boldsymbol{\lambda}_d$ and $\boldsymbol{\nu}_d$ are vectors representing the wrench and twist values.

Based on these definitions and the operational space dynamics formulation 3.6 the inner control loop can be specified as

$$\boldsymbol{h}_c = \Lambda(\boldsymbol{q})\boldsymbol{S}_\nu \boldsymbol{\alpha}_\nu + \boldsymbol{S}_f \boldsymbol{f}_\lambda + \mu(\boldsymbol{q}, \dot{\boldsymbol{q}}) + \Lambda(\boldsymbol{q})\dot{\boldsymbol{S}}_\nu \boldsymbol{\nu}, \tag{3.13}$$

where $\boldsymbol{h_c}^\intercal$ is the control wrench and $\boldsymbol{\nu}$ is the current robot velocity. It can be seen that force and velocity subspaces are completely decoupled in this equation. A derivation of this equation can be found in [62, Section 7.4.1]. The values $\boldsymbol{\alpha_\nu}$ and $\boldsymbol{f}_\lambda$ can be derived from the outer control loop.

For force/velocity-based control, this outer loop can be defined as follows:

$$\boldsymbol{f}_\lambda = \boldsymbol{\lambda}_d + \boldsymbol{K}_{p\lambda}[\boldsymbol{\lambda}_d - \boldsymbol{\lambda}] \tag{3.14}$$

$$\boldsymbol{\alpha_\nu} = \dot{\boldsymbol{\nu}}_d + \boldsymbol{K}_{p\nu}[\boldsymbol{\nu}_d - \boldsymbol{\nu}] + \boldsymbol{K}_{I\nu} \int_0^t [\boldsymbol{\nu}_d - \boldsymbol{\nu}] d\boldsymbol{\tau}, \tag{3.15}$$

where $\boldsymbol{K}$ specifies suitable matrix gains. Equation 3.14 controls the force, while Equation 3.15 controls the velocity. To use force/position control instead of force/velocity, Equation 3.15 needs to be exchanged with a position control law.

3.3.3 Vision-Based Control

A different control approach is the use of computer vision data to control robot motions [62, Section 24]. Vision data can be obtained by one or more cameras mounted on the robot or in the workspace. Schemes based on vision aim at minimizing errors [62, Section 24.1]

$$\boldsymbol{e(t) = s(m(t), a) - s^*}. \tag{3.16}$$

In this equation, a set of image measurements $\boldsymbol{m(t)}$ is used to compute a vector of visual features $\boldsymbol{s(m(t), a)}$ using a set of system parameters $\boldsymbol{a}$. This feature vector is compared to the desired values $\boldsymbol{s^*}$. The measurements are, for example, image coordinates of interest points while the parameters are, for example, the intrinsic parameters of a camera. There are two general approaches to calculate the feature vector $\boldsymbol{s}$: In image-based visual servo control (IBVS), features are directly available in image data, while in position-based visual servo control (PBVSC), features have to be estimated from image data as they usually consist of 3D position data.
If the camera is mounted on the robot manipulator, the camera velocity $\boldsymbol{v_c}$ is considered as robot controller input. To relate it to the measured feature vector, the interaction matrix $\boldsymbol{L_s}$ is necessary:

$$\boldsymbol{\dot{s} = L_s v_c}. \tag{3.17}$$

In IBVS, image coordinates of interest points are used to define the feature vector $\boldsymbol{s}$, while the parameters $\boldsymbol{a}$ are the camera's intrinsic parameters. The interaction matrix is obtained from taking the time derivative of the camera projection equations which can, for example, be found in [23]. Its entries contain the 2D image coordinates of the point-of-interest as well as the depth of the point relative to the camera frame [62, Section 24.2.1].
For PBVS the camera pose in a reference frame is used to define $\boldsymbol{s}$. To derive this pose from image data, a 3D model of the observed objects has to be known in addition to the camera parameters. The literature [15] refers to this problem as 3D localization problem. For building the interaction matrix, the translation vector and rotational parametrization of the camera pose are used [62, Section 24.3].

Using Equations 3.16 and 3.17, the relation between camera velocity and error derivative can be obtained:

$$\boldsymbol{\dot{e} = L_s v_c}. \tag{3.18}$$

The control law can now be obtained by integrating Equation 3.19. For an exponential decoupled decrease [62, Section 24.1] this leads to

$$\boldsymbol{v_c = -\lambda L_s^+ e}. \tag{3.19}$$

In this equation λ is the integration constant, while $\boldsymbol{L_s^+}$ is the Moore-Penrose inverse of $\boldsymbol{L_s}$. In practical implementations, $\boldsymbol{L_s}$ and $\boldsymbol{L_s^+}$ cannot be obtained exactly and have to be approximated.

3.4 Sensing Devices and Approaches

To use robots in assembly under the presence of uncertainty, perception of the workspace is necessary. With the employment of sensors it is possible to measure the current state of a robot's kinematics and dynamics and compare this data to desired contact states. A wide variety of sensors is available for this purpose. They can be located within the robotic system (internal sensors), attached to the robotic system or located in the workspace (external sensors).
There are different ways to classify sensors. In [62, Section 4.2] Christensen and Hager differ between proprioceptive and exteroceptive sensors. The former are used to measure the internal state of the robot. This includes, for example, joint angles, motor currents, and end-effector forces. The latter can perceive information about the environment like object positions and interaction forces.
A different categorization is presented in [36]. Here, sensors are categorized into contact sensors, which include force-torque and tactile sensors, and non-contact ones, which include cameras or laser scanners.
To use sensors, an inverse sensor model has to be available in order to convert sensory data into state information [36]. As all sensory data can be described as vectors with different data types, it is possible to treat different sensors in a unified way.
For the integration of sensors into robot control open and closed loop configurations can be applied as described in [36]. In open loop configurations the desired robot state is calculated by the sensor and sent to the controller without consideration of the actual robot state. Closed loop schemes compare the desired and actual states in each cycle.
Joint position sensors acquire the angular position of the joint by reading a pattern from a disc [62, Section 4.2]. Using this technique, an accuracy of $1/1000^\circ$ can be reached. To get the end-effector position of the robot, the values calculated by the joint position sensors are used as inputs for the robot's forward kinematics model.
For the haptic measurement of interaction forces and torques, piezoelectric elements can be used [62, Section 4.2]. They generate a voltage that is proportional to their deformation. By using several carefully placed elements, it is possible to measure forces and torques in all six degrees of freedom. The force/torque sensors are usually mounted on the robot's wrist or at its fingertips.
Forces and torques can also be estimated from measured joint torques [66] (see Chapter 10).
In optical sensing an image is obtained from which local features like points and edges are extracted. From these feature, state values like positions can be extracted. The most important examples for optical sensors are laser scanners and camera sensors. With laser scanners, a 3-dimensional representation can be generated on which different geometric measurements can be carried out. Camera sensors acquire digital images as array data. This data can be used to extract all kinds of useful information. A common example is to extract significant points from an image and calculate their position in the real world by applying the inverse model of the camera.
Another possibility to measure position, velocity and forces is the usage of accelerometers and gyrometers. With these devices, translational and rotational accelerations can be measured and dynamic forces can be calculated directly if the mass is known. A drawback is that methods like this cannot measure static forces.
To combine sensory information from different sources, a variety of sensor fusion approaches exists. These are most commonly based on probabilistic methods [74]. The foundation of most approaches is *Bayes' Rule*. This general rule can, for example, be implemented using *Probabilistic Grids* as done in [67]. Another widely used approach is the *Kalman Filter* [4], which is a recursive linear estimator for a continuous valued state [62, Section 25.1.3]. As an alternative to probabilistic approaches fuzzy logic can be employed [19].

4 Assembly Skill

In Chapter 2 robotic assembly and its requirements were presented. Chapter 3 described the low-level requirements to fulfill assembly tasks. Here, the gap between these two aspects is filled by presenting the concept of *Manipulation Skill for Robotic Assembly* (or *Assembly Skill*).

4.1 General Idea and Requirements

The paradigm shift in manufacturing from mass production towards mass customization [5] demands a new kind of flexible industrial robot. These robots have to be able to cope with uncertainties in order to be able to operate in at least partly unstructured environments [5]. It also has to be possible for non-experts to reprogram them quickly on the shopfloor. These two goals contradict each other: uncertainty handling demands advanced sensor-based control strategies, which results in even more complicated programming than standard position control.
A possible solution to this conflict is the concept of *Assembly Skill*. The general idea is to store the ability to perform elementary robot actions in reusable primitives. A skill is such a primitive that allows the coordination, control and supervision of a specific task. The primitives can incorporate advanced task specifications (see Chapter 3.2), necessary control (see Chapter 3.3), and sensing capabilities (see Chapter 3.4), which allows a skill to handle uncertainties during execution. As all of this information is encapsulated, the programmer does not need to worry about the details and can program the robot by assembling predefined skill primitives. An even more convenient approach is to eliminate the need for manual programming entirely and derive the robot programs from an assembly planning algorithm that selects the sequence of skill packages to be executed automatically. Consequently, skill primitives form a link between high-level planning and low-level control of task execution.

State of the Art

The first approach to use skill primitives as representation of atomic robot motions was employed by Hasegawa [26]. Another skill-like system was used in [49] for robotic assembly. They used a rule-based logic combined with hidden Markov models to let the robot autonomously define how a task is executed.
Skill primitives have been combined to discrete motion networks which represent tasks by Kröger et al. [37]. They used the TFF to define manipulation primitives, which were used as building blocks for skill primitives.
Smits [63] used a constraint-based programming approach [13] in his PhD Thesis to create a skill-controlled robot motion system. He used the iTaSC [64] framework, which is a skill-based low-level motion control framework, to interpret the motion commands defined by constraint-based programming. A skill is then defined as a set of constraints that form a functional motion. Linderoth [43] followed a very similar approach. In his thesis, skills are represented by state machines implemented in JGrafchart[1], while each state contains iTaSC-based motion commands. Additionally, the "Knowledge Integration Framework" (KIF) for storing, sharing, and reusing skills and other assembly knowledge was introduced in [65].
Discrete motion networks are also used in [51], where the networks represent force-over-position trajectories.
Bøgh et al. [5] proposed a different approach. They see skills as traditional, "unintelligent" robot motion programs (macros) that are augmented with pre and post-conditions to add situational knowledge. Macros are supposed to work on objects in the workspace that are recognized via some kind of sensing device (e.g. optical).
Another way to represent skills is as dynamic system. In [29] Ijspeert uses spatially and temporally invariant dynamical systems. Here, a simple canonical system is transformed to adapt to new environmental constraints. This was, for example, employed in [55] to acquire new skills by learning techniques.
A novel approach of employing dynamic systems is presented in [48]. Here, skills are firstly recorded as a spline trajectory. Afterwards, 2-dimensional parts of this spline are instantiated as dynamic systems known from fluid dynamics: the Navier Stokes Equation for incompressible fluids. For every skill one of these fluid simulations

[1] http://www.control.lth.se/grafchart/

is created. Robot motions can be described by this approach as flows in a simulated current. This makes the approach very robust towards environment disturbances because the simulated current bypasses obstacles.
A very recent implementation of a skill-based system by the German Aerospace Center (DLR) can be found in [72]. In this publication a new programming language based on UML/P statecharts is introduced to describe skills. The TFF is used for motion description.

To create a robotic system based on manipulation skill, three important aspects have to be considered: Acquisition, representation and reusability.

Acquisition

The idea of robots acquiring skills is based on human behavior. To fulfill complex tasks like assembly, humans rely on a limited set of motor primitives. These are acquired in childhood by imitating the surroundings and are improved by experience.
This idea is used for the acquisition of skills that are usually acquired by human demonstration. Some approaches introduce additional learning steps to improve the acquired skills.
This was, for example, done in [34], where skills are acquired by learning techniques. The presented approach combines mimicking a human demonstration via imitation learning with subsequent self-improvement by reinforcement learning.
Another popular way to record a skill from human performance is a teaching process as it is, for example, used in [12] with additional (e.g. learning) steps to generalize it. This way is also used in [48]. To extract skills from a recorded 3-dimensional trajectory, it is split into 2D parts via a 3D plane fitting algorithm and matched to previously recorded trajectories.
To identify human actions that are fit for representation as skill, it is possible to analyze industrial tasks performed by humans [5].

Representation

A skill representation must have the capability to represent a task as well as necessary control and sensing to fulfill it. Task specifications suited for skills are, for example, the Task Frame Formalism or a constraint-based specification (see Section 3.2). Appropriate control strategies are, for example, force or vision-based strategies (see Chapter 3.3). Suited sensors are mainly position, haptic, tactile and optical sensors. This information should be encapsulated in a way that the primitive can easily be refined, parametrized and reused in different situations. Various examples for skill representations can be found in Section 4.2.

Reusability

The aspect of reusability is very important when designing skill primitives. It is gained by using a generic template of an elementary action for a skill primitive that can be parametrized and refined to fit to new situations. Templates are stored in a library. To select skills from a library, they can, for example, be matched to an assembly plan and chosen according to identifier attributes [73].
To meet the requirement of reusability, a balance has to be found between specific details and generality in the skill representation. If it is not possible to adapt a skill to appropriate new situations because it is too specific, a new set of skills has to be developed for each new task, which contradicts the advantage of this concept. If, on the other hand, the representation is too generic, the parametrization is too difficult and does not depict a simplification compared to robot programming.
This balance also has to be considered for the size of task a skill can fulfill. If a primitive is designed to fulfill a very basic action, it can be reused in many applications. Contrary, larger "action packages" are easier to use as they reduce the effort to choose and parametrize appropriate skills for a certain task.
It is not only desired to reuse a skill in different situations, but also on different robotic platforms. To achieve this, the knowledge incorporated in a skill has to be stored and shared in a unifying way. An example is the knowledge integration framework presented in [65], where semantic web technologies are used for a standardized representation, storage, and distribution of skill knowledge.

4.2 Skill Representation Approaches

In this section, various approaches that are briefly presented in the previous section are described in more detail.

Task Frame Hybrid Commands

Kröger et al. [37] use the Task Frame Formalism (see Chapter 3.2.1) to define skill primitives. According to [22] skill primitives are defined as a tuple containing a tool command, a stop condition and a hybrid move. A tool command is a simple action like "open gripper". The stop condition is a Boolean function and it determines the end of the execution of a primitive. In a hybrid move, a task frame and related setpoints are defined. The setpoints contain desired control values (e.g. velocities and forces), depending on the applied control strategy. Hybrid control strategies as described in Chapter 3.3.2 are used to carry out commands defined like this. The skill primitives can be connected to a skill primitive net to describe complex tasks. This net is implemented as a Finite State Machine. In the net, each primitive is represented as a node connected to possible successor nodes via directed arcs. Arcs can be augmented with transition conditions. The disjunction of all transition conditions of the outgoing arcs of a node form the stop condition of the skill primitive related to the node.
With the advanced concept of the "Adaptive Selection Matrix" [22] it is possible to define several alternative setpoints in each primitive and switch between them during execution. Each setpoint is connected to a fitting control strategy.

UML/P Statecharts

A practical approach to robotic skill was followed by the German Aerospace Center DLR [72]. They proposed the new robot programming language *LightRocks* (Light Weight Robot Coding for Skills) based on UML/P state-charts. This approach enables intuitive skill-based robot programming. In their example application they used this new language for programming compliant motions on a DLR light weight robot arm. The general theoretical concept behind this implementation is derived from [71]: TFF-based motion primitives are connected to nets to fulfill complex tasks. In the *LightRocks* implementation, nets are formed from basic elements called *Elemental Actions*. These elements are specified as tuples {Device, DeviceCommand, StopCondition, ReturnValue}. In this tuple, a device is a tool, robot, or perception unit. If the device is a robot, the device command is specified as a Cartesian TFF-based motion command containing information about the task frame, the controller setpoint, and additional controller parameters. As an impedance controller (see Section 3.3.2) was employed in the example implementation, the additional parameters contain stiffness and damping values. The device commands are sent to the control unit via a TCP/IP interface. The *Skills* in the DLR concept are specified as nets where the nodes are elemental actions connected by transitions. Transitions have pre and post-conditions by which they are triggered. Another level in the implementation are *Tasks* built from nets of skills in the same way skills are defined. The implementation of this concept using UML/P state-charts is convenient as many of the languages elements like states and transitions can be easily mapped to the skill definition framework.

Constraint-Based Knowledge Framework

As in most of the presented approaches, FSMs are also used for the skill representation presented in [63]. Contrary to, for example, [71], each skill is described by an individual FSM. This denotes that skill primitives are higher-level elements and describe a larger part of an overall task. In each FSM, states, actions to be executed within states, transitions, and events to trigger transitions are specified. A state contains a specific configuration of a controller interface. This configuration is specified by a constraint-based task description [13] (see Chapter 3.2.2). A configuration consists of a set of weighted, parametrized constraints. The action that is connected to a state can produce continuous changes of the constraints defined within it. The continuous changes are limited to the discrete configuration defined by the state. To change the state of the FSM, transitions are activated based on events. These events can be triggered from within a state (when a state parameter reaches a certain value) or from the outside (from a higher-level supervision module). To make transitions smoother, the concept features the possibility to use a transition parameter. This parameter is simply a weighting coefficient to create blends between states during transition.

For the execution of skills the iTaSC [64] control framework is used. This software is designed to connect constraint-based task specifications as employed in this approach to robot control. The controller is embedded in a threefold control structure consisting of a task controller, a skill controller and a motion controller.
While the skill control triggers transitions in the above described FSMs, the motion controller executes the actions from the FSM states. The task controller represents the high-level knowledge in the system. It accesses a repository in which skills are stored and connects them to sequences. In such a system, the skill level forms a link between continuous robot control and symbolic high-level knowledge representation.
In [65], a framework based on iTaSC was used as well. There, the constraint-based skills were employed for a small part assembly application. A robot workstation like the one used in this thesis is employed with additional external force/torque sensors to perform the assembly. On the high-level side, a platform to store and share skills was presented. This "Knowledge Integration Framework" stores the skills in ontology and provides graphical task specification tools.

Force-Position Trajectories

A different approach to use state machines in skill representations is presented in [51]. The state machines are used to describe assembly tasks and control them based on the occurring contact situations in the task. To accomplish this, position-based control actions are selected based on contact states. These contact states are detected by investigating a trajectory of sensor measurements. Measurements can, for example, be force or position values. Critical points in this trajectory represent transitions between actions. A critical point can be described as a discontinuity in the trajectory, for example a rapid decrease of a measured force. For the detection of critical points in practice, the derivative of the force is used. To make the primitives reusable, the transitions triggered by the critical points are specified in a qualitative way.

Dynamic Motion Primitives

Instead of a highly engineered approach, where a lot of predefined information is included in skill primitives, Peters et al. [55] use a more general representation. They represent skills as dynamic systems that are acquired by human demonstration and refined by trial and error learning approaches. The framework they developed contains three main components: Primitives represent elementary motions as dynamic systems, a supervisor module selects and parametrizes the primitives, and an execution module is used to execute the primitives. Furthermore, a superior learning module can adapt all of these three components.
The **motor primitives** are based on a formulation by Ijspeert et al. [29], where motions are described as dynamic systems:

$$\dot{\boldsymbol{x}}^d = \pi_i(\boldsymbol{x}^d, \boldsymbol{x}, t, \rho_i)\,. \tag{4.1}$$

Herein, $\boldsymbol{x}^d$ denotes the system state and t denotes time. The shape of such a primitive is determined by task parameters $\rho_i = [\theta_i, d, g, A, ...]$. This array can contain various elements like duration, amplitude and goal of the motion. The reusability of the primitive is enabled, as its shape is solely determined by the parameter θ_i, while it is invariant under changes of the other parameters.
The main task of the **supervisor level** is to allow usage of a primitive by selecting it and modifying its task parameters. Advanced functions are, for example, to create new primitives by creating convex combinations of existing primitives. Furthermore, different sequencing functions like blending and superposition of primitives are available.
The **execution level** has to create motor commands from the primitives. Therefore, control laws employing precise analytical models are used. To make the models more accurate, they can be adapted online by learning techniques.
Learning is applied to all three components. For the motor primitives, the parameter θ_i is adapted. In general, the learning is performed in two steps: imitation learning is used to initialize the skill while reinforcement learning is used to improve it [55].

4.3 Comparison of Skill Representation Approaches

	Task Frame Skills	Constraint-Based Skills	Force-Position Trajectories	Dynamic Motion Primitives
Task Specification	++	+	–	0
Uncertainty Handling	+	+	+	++
Reusability	++	+	+	++
Semantic Meaning	–	++	–	–
Engineering Effort	–	–	–	++
Implementation	++	+	0	–
Industrial Usability	++	++	–	–

Table 4.1: Comparison of Skill Representation Approaches presented in Section 4.2. The comparison is based on the symbols "++" (very good), "+" (good), "0" (neutral) and "–" (not good).

The presented concepts can be compared according to different criteria described in the following:

- *Task Specification*: How easy is the specification of a task in the concept?
- *Uncertainty Handling*: How well can uncertainties be resolved?
- *Reusability*: How well can skill primitives be reused?
- *Semantic Meaning*: How meaningful are the tasks a skill primitive refers to?
- *Engineering Effort*: How elaborate is the design of skill primitives?
- *Implementation*: How easy is the implementation of the concept?
- *Industrial Usability*: How suited is the concept for industrial use?

Task frame skills, which include the approaches presented in [37] and [72], are very well suited for industrial use, as demonstrations have shown. Although the Task Frame Formalism limits the applicability of the concept, it is able to fulfill most industrial tasks. Specifying tasks is intuitive in this approach, but the effort to do so is considerable.
Constraint-based skills are more flexible, because constraint-based programming is used. A drawback of this formalism is that the specification of tasks becomes less intuitive. Each skill primitive is responsible for a semantically meaningful task consisting of several basic actions. This has advantages but also limits reusability of the primitives.
Force position trajectories are well suited to deal with uncertainties. The qualitatively described trajectories that are used enable the reuse of primitives. Drawbacks are the high effort of designing trajectories and the well structured environment (regarding sensing and geometry) that is currently necessary to use the concept.
Dynamic motion primitives are a highly flexible and self-adaptive way to fulfill all kinds of motion tasks. The employed learning techniques enable a reuse of primitives in many situations and a robustness towards uncertainties. The drawback is that it is hard to include the system in today's industrial setups. In general, the concept aims more at working in completely unstructured environments like the service sector and it is overpowered for industrial shopfloors.

5 Contribution

The approaches presented in the previous section are either not suited to industrial use or do not feature a generic skill representation. This thesis contributes to the development of skill-based robot systems to fulfill assembly tasks by presenting an approach that aims at finding a tradeoff between a generic representation and industrial usability. The contributions with respect to different levels of the execution of an assembly task are shown in Figure 5.1.

The main contribution of this thesis is the introduction of a new manipulation skill representation that is used as an interface between high-level assembly planning and low-level assembly execution. Skill primitives are stored as reusable templates that can be applied to new situations by parametrization. A skill is represented in two ways: as a 12-dimensional trajectory describing compliant motions in pose-wrench space and as a Finite State Machine to execute the motions that are derived from this trajectory. The trajectory can represent a robot motion independently of the robot's speed as it is not explicitly dependent on time. A qualitative node-based concept is proposed to store the trajectory in a generic way. Mapping procedures are introduced to transfer the data from the trajectory to the Finite State Machine representation. The Finite State Machine uses the Task Frame Formalism to specify robot control actions. For the parametrization of skill primitives, mappings are introduced that allow the acquisition of parameters from an assembly tree representation. Furthermore, an interface for fast manual parametrization is introduced. A categorization of skills based on the proposed skill representation was started. The two skills *Insert* and *Snapfit* were investigated in depth.

On the high-level side of the assembly system, a new assembly specification was introduced. This specification is an assembly tree structure which incorporates descriptive information about an assembly as well as instructions on how to perform an assembly. The specification was implemented as an XML data structure. A systematic way to acquire the assembly tree was out of the scope of this thesis.

On the low-level side, the applicability of such a system to standard industrial hardware was shown. For control, only standard position control was employed while force/torque information was used to switch between different position-controlled motions. To circumvent the need for dedicated force/torque sensors to gather the relevant information, a novel approach to estimate forces from motor data was proposed. The results in this section were published in [79].

The concept was implemented as a demonstrator using an ABB dual-arm concept robot and standard ABB robot control software. Its applicability was shown by performing an exemplary small part assembly featuring the skills *Insert* and *Snapfit* among others.

Another contribution is the summary of the state of the art in manipulation skill for robotic assembly.

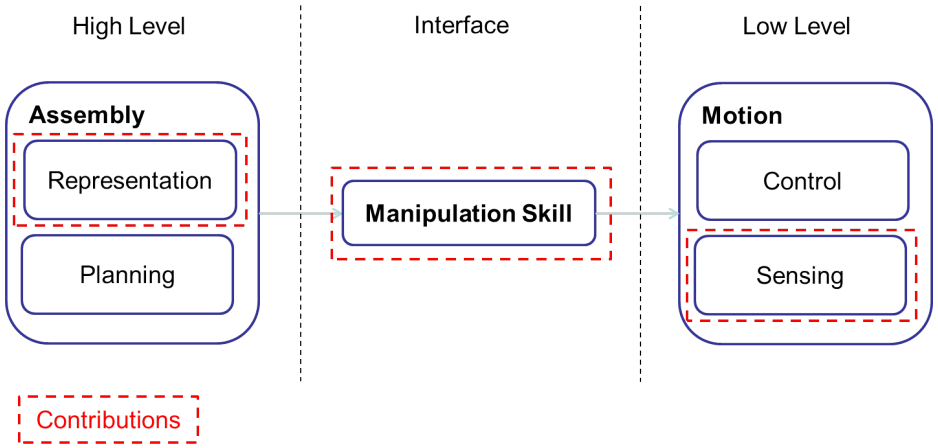


Figure 5.1: Schematic overview of the different levels of an skill-based assembly system and visualization of the contributions of this thesis made on the different levels.

Part II
Assembly Skill Representation

6 Assembly Skill System

As presented in the previous chapters, a manipulation skill-based concept is well suited to conduct assembly tasks. This chapter presents the understanding of assembly employed in this thesis and the concept derived from it.

6.1 Assembly Definition

Assembly is the process of putting together parts in order to create a complete product [62, Section 26.4]. Several assembly tasks as described in Chapter 2 are derived from this general topic.
When two parts are assembled, they move at least partially in contact to each other. Subsequently, assembly is a compliant motion problem where contact forces and torques between parts play an important role (see Chapter 3). To describe this kind of motion, the relative pose and wrench of the parts can be used. The desired relative motion of two parts during assembly can be seen as following a desired trajectory embedded in a 12-dimensional space spanned by the components of the relative pose and wrench (a more detailed description of this concept can be found in Chapter 7.2).
In relation to this assembly concept, each skill is specified by a pose-wrench trajectory representing a certain part of an assembly task. For the actual execution this pose-wrench trajectory is transformed into a Finite State Machine representation. By sequencing skills, the trajectory of an overall assembly task can be generated. A crucial property of a skill is that it is reusable for tasks of the same type. To accomplish this, the skill is stored in a generalized way as a template and needs to be parametrized before it can be used. The skill representation is described in more detail in Chapter 7 while this chapter puts skill into the concept of an overall execution system.

6.2 Concept Structure and Information Layers

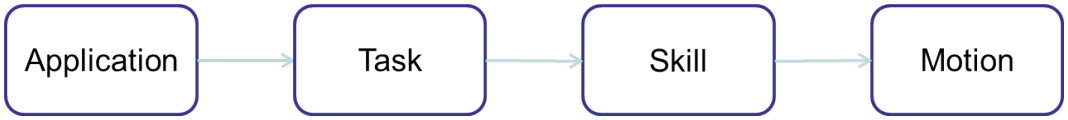


Figure 6.1: Information Layers of the Proposed Assembly Skill Concept.

The overall goal of the proposed system is to create and oversee a series of robot actions, which leads to successful completion of a specified assembly application. To accomplish this, a series of skills, represented as reusable templates, is selected from a library and parametrized according to the tasks the application consists of. Each of the skills includes the capability to coordinate, control and supervise an elementary robot action. The concatenation of all the selected elementary actions results in the accomplishment of the assembly application.
To follow this basic course of action, four layers of information are distinguished in the proposed system as shown in Figure 6.1.
The *Application* and *Task* levels are centered around the geometric description of an assembly and are independent of the system the assembly is performed on. Contrary, the *Skill* and *Motion* levels contain information on how the assembly can be performed on a robotic platform and are centered on motion descriptions. In more detail, the levels contain the following information:

Application The *Application* specifies a complete product that is to be assembled. It is represented by a binary tree structure called *Assembly Tree*. This data structure contains the whole part described by a hierarchy of geometrical relations between sub-assemblies and individual components. Furthermore, annotations of the tree nodes contain information on how the parts have to be assembled.

Task A *Task* is a traversal step in the assembly tree. In this context, a task is limited to a scenario of two parts being put together. Each task is instantiated by one or many skills which form a net of skills.

Skill A *Skill* is an elemental manipulation constituting a step towards achieving a task. It contains all the necessary data to coordinate, control and supervise the actions necessary to fulfill this step. In the system a skill is represented by a generic template selected from a skill library and a specific instantiation created from this template and data from the task. The skill is build from a net of motions that is represented twofold in the skill: as a Finite State Automaton and as a trajectory in pose-wrench-space.

Motion A *Motion* is a constrained or unconstrained action that describes which degrees of freedom of a reference frame are controlled in which way. The frame is associated with the parts to be assembled. This definition of a motion refers to the Task Frame Formalism described in Chapter 3.2. Specific frame and specific motion values are set using the parameters from the instantiated skill. Instead of a motion the data on this level can also represent an elementary tool action.

The application and task levels are described in more detail in the following section, while the skill and motion levels are presented in Chapter 7.

6.3 Assembly Tree Specification

6.3.1 Assembly Tree Structure

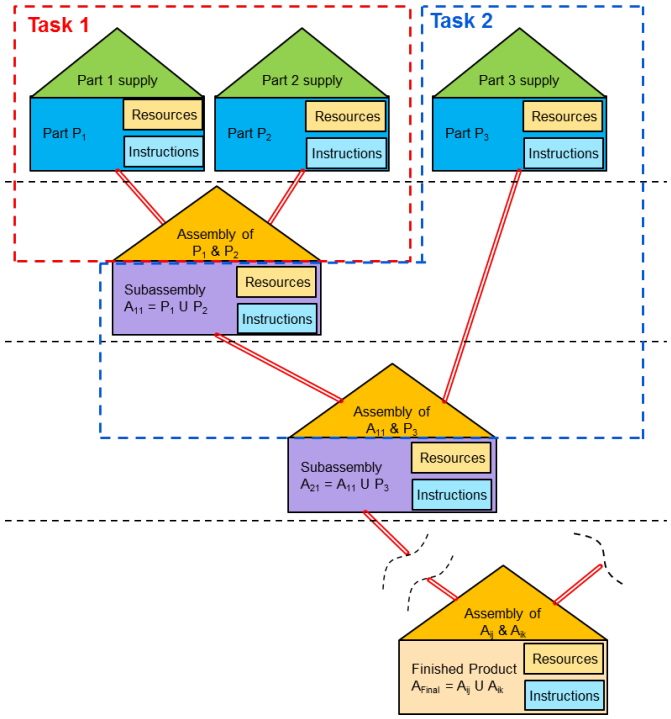


Figure 6.2: Assembly tree representation containing individual parts and sub-assemblies as tree nodes that together form a finished product. Each node contains assembly instructions and a description of resources necessary for assembly. [17]

The application is specified by a tree structure called *Assembly Tree*. This Assembly Tree is an annotated graph with an explicit root node representing the completed product and a branch hierarchy representing constitutive sub-assemblies. Besides descriptive information about the parts and sub-assemblies, annotations contain assembly-specific instructions and requirements.
In this concept the sub-assemblies are limited to two parts being put together. This is not a hard constraint in assembly processes as scenarios are imaginable where more than two parts are put together at the same time. But as two part situations are the most common case in assembly, this limitation was a fair simplification for the present project. More complex situations can mostly be described by a hierarchy of binary assemblies if necessary. The tree structure resulting from these specifications consequently is a binary tree.
The hierarchical structure of a tree structure like this implicitly contains the order of the assembly to be performed. Sub-assemblies on a lower hierarchy level have to be completed before a higher-level assembly can be performed. By default, assemblies on the same level can be performed independently of their relative order. If a specific order is desired, this can be accomplished by specifying a sequence indicator valid for one hierarchy level.
The following node types can be found in the assembly tree:

- A *Root Node* representing the completely assembled product
- *Branch Nodes* representing sub-assemblies
- *Leaf Nodes* representing individual parts

6.3.2 Assembly Tree Elements

This section describes the elements present in an assembly tree structure in more detail. These elements refer to the types of nodes in the previously described tree structure.

Part

A part $\boldsymbol{P_i}$ is an individual component used in an assembly and it is represented by a leaf node in the tree structure. It contains the following:

- A *geometric representation* of the part (e.g. CAD)
- Additional *descriptive information* (e.g. mass properties, material)
- A transformation matrix $\boldsymbol{T_i} \in \mathbb{R}^{4x4}$ describing the part's assembly goal position with respect to its parent node
- Assembly instructions $\boldsymbol{H_i}$ specifying how this part has to be assembled with another part or sub-assembly
- Resources $\boldsymbol{R_i}$ specifying what resources are needed to assemble this part with another part or sub-assembly

The *geometric representation* implicitly defines a local coordinate frame of the part. This local frame is related to the local frame of the part's parent node by $\boldsymbol{T_i}$. If no assembly instructions are contained in a node, this means that the part or sub-assembly represented by the node is not manipulated during the task and remains at its position.

Assembly

An assembly $\boldsymbol{A_i}$ can be a sub-assembly in a product structure represented by a branch node or a completely assembled product represented by the root node. This element contains the following:

- A transformation matrix $\boldsymbol{T_i} \in \mathbb{R}^{4x4}$ describing the assembled position of the assembly with respect to its parent node (equals identity if $\boldsymbol{A_i}$ is the root node)
- Assembly instructions $\boldsymbol{H_i}$ specifying how this sub-assembly has to be assembled with another part or sub-assembly
- Resources $\boldsymbol{R_i}$ specifying what resources are needed to assemble this sub-assembly with another part or sub-assembly
- Up to two child nodes of type assembly $\boldsymbol{A_j}$ or part $\boldsymbol{P_j}$

Descriptive properties like CAD data do not need to be stored explicitly in assembly nodes, as this information is derived from a combination of the respective properties of its child nodes. It has to be noted that $\boldsymbol{H_i}$ and $\boldsymbol{R_i}$ refer to an assembly task specified by this node's parent assembly node - the task where the sub-assembly specified by **this** node is assembled with another sub-assembly or part.

Assembly Instructions

The assembly instructions $\boldsymbol{H_i}$ specify the necessary steps to fulfill an assembly. They consist of a sequence of assembly actions $\boldsymbol{a_{ik}}$ that need to be performed.

Each assembly action $\boldsymbol{a_{ik}}$ contains the following:

- An *activity indicator* specifying the type of action
- A sequence of targets $\boldsymbol{\tau_{ikl}}$ that define a path the associated part needs to follow on the way to its assembly goal

- A task frame $\boldsymbol{TF_i} \in \mathbb{R}^{4x4}$ that is used as a reference frame to specify the targets $\boldsymbol{\tau_{ikl}}$
- Additional *process parameters* (e.g. speed)

The activity indicator is used to select suitable skills from the skill library to execute the action. Each target $\boldsymbol{\tau_{ikl}}$ can be explicitly defined as a point in pose-wrench space or as an offset to a previous target.

Resources

The recources node $\boldsymbol{R_i}$ contains information about the hardware requirements of an assembly task. These are specified by the following:

- A list of *necessary sensors*
- A list of *necessary tools*
- Additional requirements of the executing unit (e.g. number of robotic arms, degrees of freedom per arm)

The resources can be used to determine if a task can be fulfilled by the available hardware.

6.3.3 Assembly Tree Traversal

Two kinds of information can be extracted from the assembly tree: the geometrical representation of an assembled product and instructions how the product has to be assembled. By extracting the latter, an executable assembly plan is created. To do so, the assembly tree is first split into tasks by traversing it. The hierarchical structure of the tree ensures that a sequence of tasks with a meaningful execution order is obtained. By default, each traversal step in the tree represents a task as visualized in Figure 6.2.
Let this traversal step consist of an assembly node $\boldsymbol{A_{12}}$ and its two child nodes $\boldsymbol{P_1}$ and $\boldsymbol{P_2}$. To define a task, the actions $\boldsymbol{a_{1k}}$ and $\boldsymbol{a_{2k}}$ are derived from the respective instruction nodes $\boldsymbol{H_1}$ and $\boldsymbol{H_2}$.
As the instruction nodes contain actions in a sequential order, an action sequence for the overall assembly can easily be generated from the task sequence. These actions are now mapped to skill templates from the library. The mapping is not necessarily a one-to-one relationship as a skill might be suitable for a sequence of several actions. After the selected skills have been parametrized with data from the actions, an executable sequence of skills is generated.
If a task is defined as in Figure 6.3, the skill sequences that are derived from the two involved assembly instructions are executed simultaneously. If this is not desired, it is more convenient to divide the task into two sub-tasks. This way, two strictly sequential skill sequences with a defined order are derived.

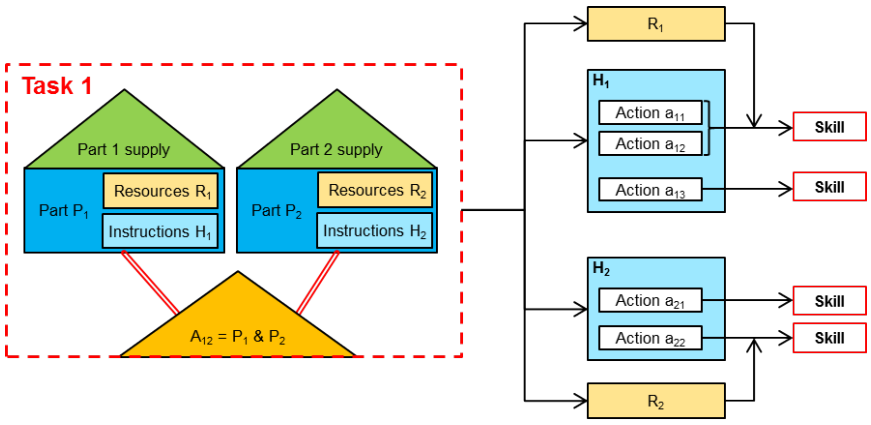


Figure 6.3: Schematic overview of how a task derived from the assembly tree is split into actions and mapped onto skill primitives

6.4 Skill Selection and Parametrization

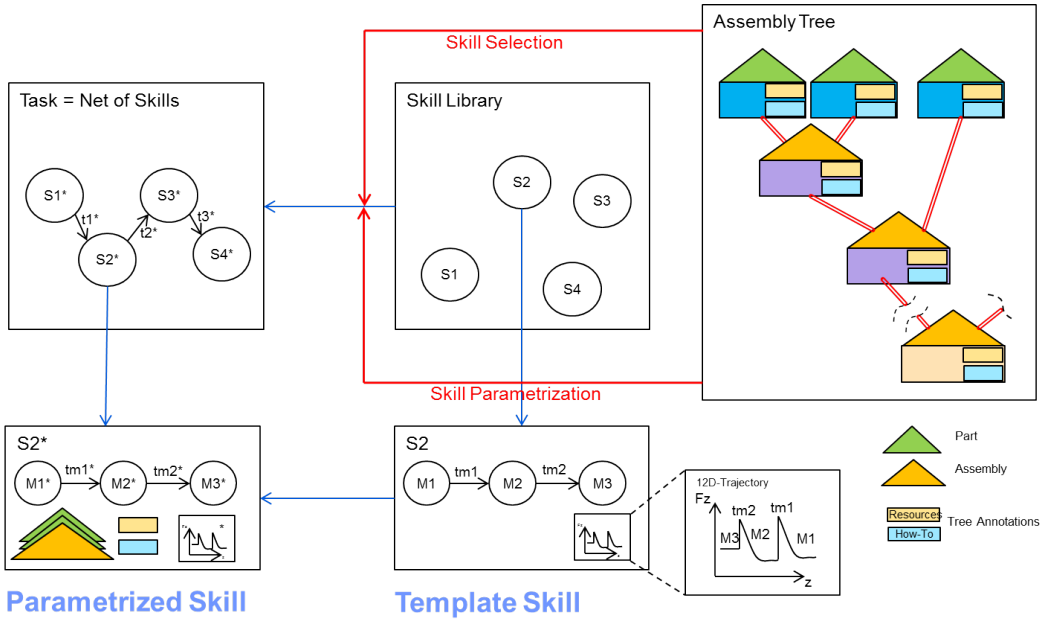


Figure 6.4: Schematic overview of the selection and parametrization of skill primitives from the skill library based on data from the assembly tree [17]

To gain a series of executable robot actions from an assembly specification, a set of skill primitives has to be selected from a skill library and parametrized with data from the assembly tree. Essentially, the problem is to map data from the application level onto the skill level. This can be done automatically according to the assembly tree. If the mapping is ambiguous, additional manual selections can be necessary in the process. The skill selection and parametrization workflow is shown in Figure 6.4.

Skill Selection

To enable the selection of skill templates from the skill library, a sequence of tasks has to be available. After the tasks have been obtained according to Section 6.3.3 and a sequence of actions was generated from it, the skill selection can start. The skill library is queried for the activity indicator and resource requirements of each action. If the activity indicator of an action is featured in a skill and the skill's resource capabilities match the action's resource requirements, the skill is marked as suited for the action. Some skills are suited for a sequence of actions instead of a single action. Therefore, they are not described by a single activity indicator but a sequence of activity indicators. If such a skill is selected, the succeeding actions in the action sequence have to be compared according to the succeeding activity indicators. When more than one skill is found, the desired one has to be selected either manually or automatically according to certain criteria. The result of this process is a set of skill templates each linked to one or many actions. In the current system, this set contains skills in a strictly sequential order. This arises from the limitation to a binary assembly tree. More complex assembly representations result in a net structure of skills instead of a sequence.

Skill Parametrization

After a sequence of skills has been selected, it has to be parametrized according to the data of the actions linked to each skill. This is done by a mapping process described in detail in Chapter 7.

6.5 Example Assembly Application

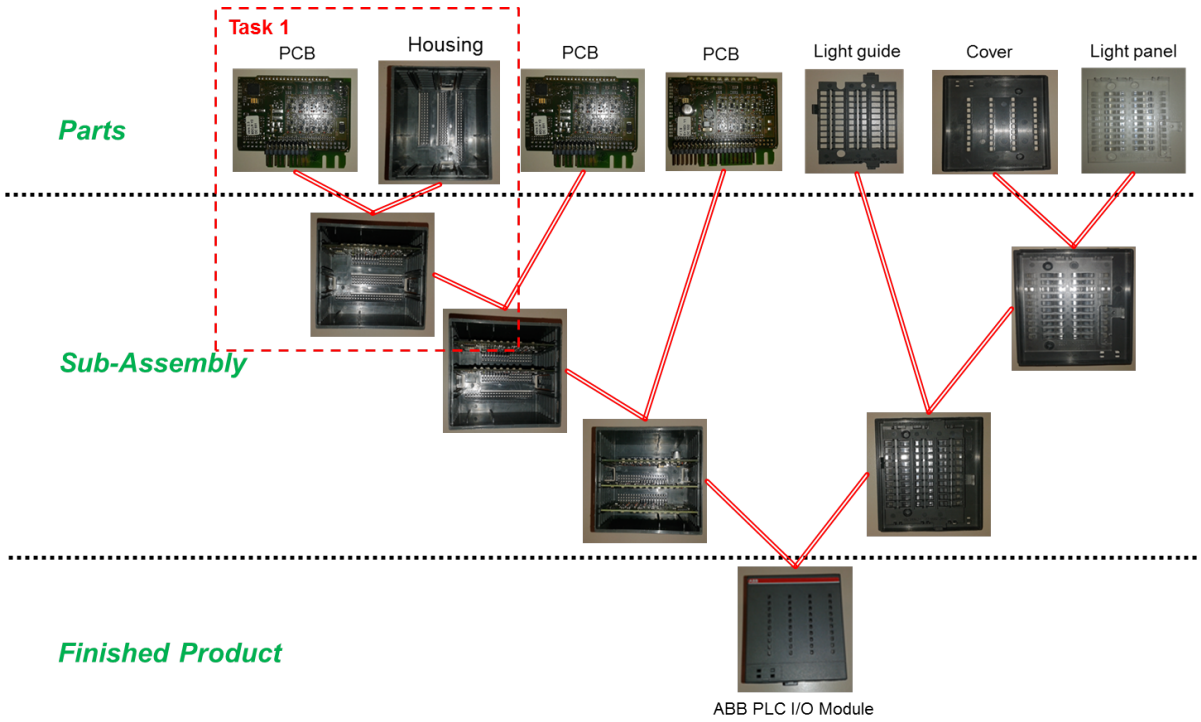


Figure 6.5: Assembly tree for an example assembly application, the assembly of an *ABB PLC I/O Module* [17]

The different parts of the presented concept can be shown by the following example. It has to be noted that this example was also used for the implementation of the demonstrator (see Section 9.3). In the example an *ABB PLC I/O Module*, which is a small electronic component, is assembled. This is done as follows: At first, three Printed Circuit Boards (PCB) are inserted into a housing. Then, a cover is assembled by attaching different components to a plastic plate via snap-fits. Finally the cover is attached to the housing via another snap-fit. The assembly process can be represented by the following assembly tree:
The overall assembly of the module, as shown in the root node, is represented by the **Application**, which in this case is called "Assembly of an I/O Module".
A possible **Task** in this application is "Insert PCB1 into housing", which is marked as "Task 1" in Figure 6.5. This task consists of:

- The assembly node $\boldsymbol{A_{12}}$: "Housing with PCB"
- Its child node $\boldsymbol{P_1}$: "PCB"
- Its child node $\boldsymbol{P_2}$: "Housing"

The actions that need to be performed to assemble the PCB and the housing are stored in $\boldsymbol{P_1}$ and $\boldsymbol{P_2}$. It is assumed that the housing is at its assembly position and does not need to be moved. This implies that the only node in this task containing assembly instructions is $\boldsymbol{P_1}$. Its instructions $\boldsymbol{H_1}$ contain the following **Actions**:

- $\boldsymbol{a_{11}}$: Pick up PCB1 (activity: "PickUp")
- $\boldsymbol{a_{12}}$: Transfer PCB1 to insertion position (activity: "Transfer")
- $\boldsymbol{a_{13}}$: Insert PCB1 into housing (activity: "Insert")

To fulfill these actions, two different sets of **Skills** are selectable:

- Skill Set 1
  - PickUp (activities: "PickUp")
  - Transfer (activities: "Transfer")
  - Insert (activities: "Insert")
- Skill Set 2
  - PickUpAndInsert (activities: "PickUp","Transfer","Insert")

It has to be noted that the skill "PickUpAndInsert" implicitly has a transfer motion defined in its net of motions. Applied to Figure 6.4, this example looks as follows:

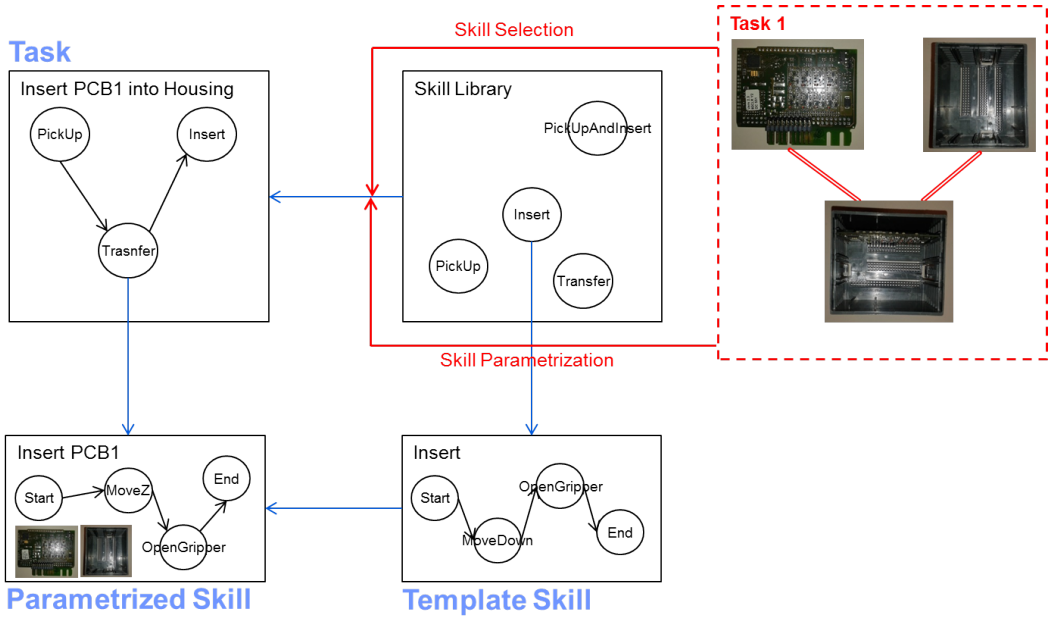


Figure 6.6: Schematic overview of the selection and parametrization of skill primitives from the skill library based on data from the assembly tree in the example assembly application [17]

7 Skill Representation

After skills have been described in the context of the overall system in the previous chapter, this chapter describes their content in more detail.

7.1 Properties of a Skill Primitive

The main purpose of a skill is to provide a simple interface to advanced robot functions. It is an elementary package encapsulating all necessary information to *coordinate*, *control* and *supervise* a logical part of an overall assembly task.
The proposed representation of a skill is twofold:

- A *motion net representation*, which represents the skill as a Finite State Machine, is used for the discrete coordination and control of robot motions. Specifically, it is used to switch between different continuous control actions.
- A *trajectory representation*, which represents the skill in pose-wrench space, supervises the continuous robot actions and triggers transitions in the Finite State Machine.

To fulfill more complex tasks, skill primitives are connected to nets that can also be described by a Finite State Machine.

Reusability is a key property of such a package, as the idea is to have a limited set of skill primitives that are capable of fulfilling a wide variety of complex tasks by connecting them. To achieve this, skill primitives are stored as generic templates that can be reused in alike situations. They can be adapted to each situation by setting a limited set of parameters. Therefore, a specific skill parametrization is created for each situation. To create parametrized skill primitives, mappings are used to merge the skill templates with external parameters. Usually, the external parameters are acquired from tasks defined in the assembly tree (see Section 6.3).

A **skill template** has the following content:

- A *Trajectory Template* to describe a skill qualitatively in pose-wrench space
- A *Motion Net Template* that specifies the type of the contained robot actions and connections
- A set of *Mapping Rules* $\boldsymbol{R1}$ to determine how external parameters are mapped onto the trajectory template to produce the trajectory
- A set of *Mapping Rules* $\boldsymbol{R2}$ to determine how the trajectory is mapped onto the motion net template to produce the motion net
- *Metadata* used for identification and selection of skill templates from the library

The metadata of a skill template contains specifications of the resources necessary to execute a skill and a sequence of activities describing the actions this skill can carry out.

The **parametrized skill** created from the template contains the following:

- A *Trajectory* to describe a skill quantitatively in pose-wrench space
- A *Motion Net* that specifies the robot actions and transition conditions in between

Beside the content described above, a skill has input and output signals used for high-level supervision purposes. Based on these signals the skill's successor can be selected, for example. The signals are implicitly defined in a skill by states and transitions in its motion net representation, which are activated under certain conditions.

Precondition If the *Precondition* is met, the execution of the related robot actions begins. A typical precondition would be that all involved parts are at a certain position.

Completion Signal The *Completion Signal* indicates that the last robot action relevant for the successful completion of a skill execution has been executed. After the completion is indicated, the quality of the skill is checked.

Quality Criteria The *Quality Signal* indicates the success of the skill execution. Several degrees of success can be represented, depending on how many different quality conditions are defined in a skill. Most commonly, it is only distinguished between "success" and "failure".

Interruption Signal The execution of a skill is interrupted, if an *Interruption Signal* occurs. Based on this signal, predefined error recovery actions can be triggered.

The following scheme visualizes the input and output signals of a skill:

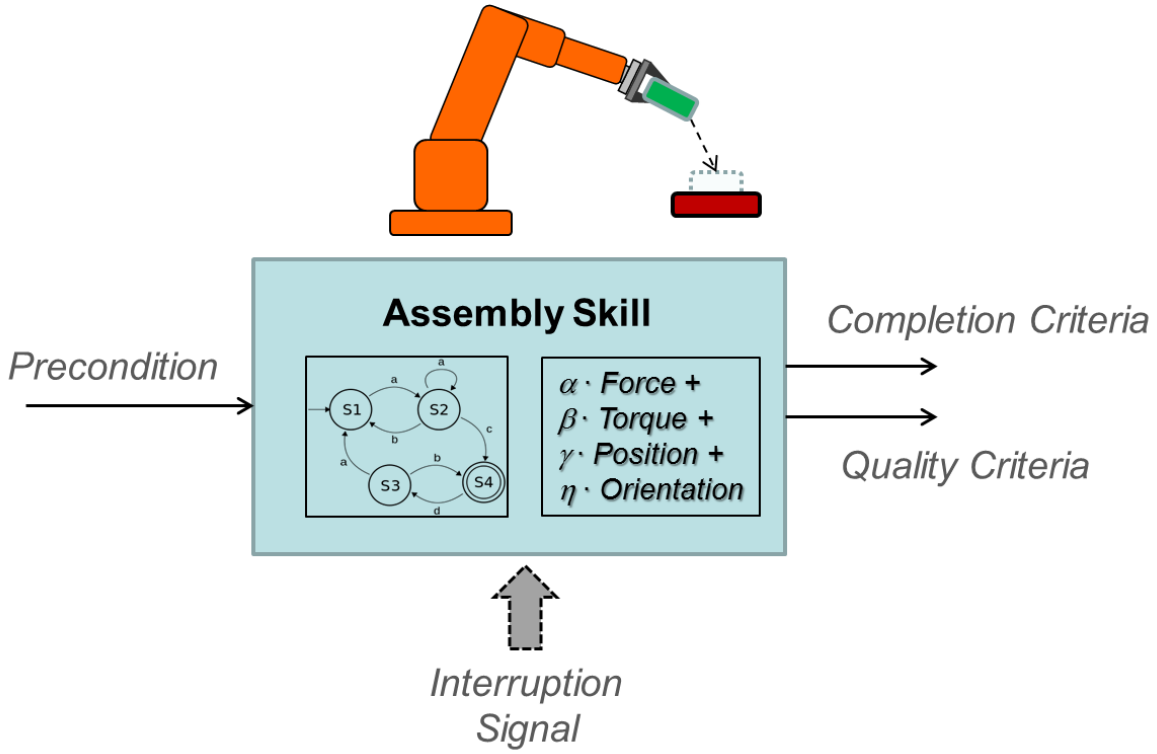


Figure 7.1: Assembly skill scheme showing the trajectory and the motion net representation of a skill as well as its input and output signals [17]

7.2 Trajectory Representation

The trajectory representation describes a skill in pose-wrench space. It has the purpose of an abstract high-level description of a skill as well as the supervision of a skill.

7.2.1 The Pose-Wrench Space Concept

In rigid body kinematics, the pose describes the spatial position and orientation of a body in the 3-dimensional Euclidean space. To specify a unique pose in space, six independent coordinates are necessary. These coordinates refer to a Cartesian reference coordinate frame, which allows the pose to be expressed as a relative rigid body displacement between two frames. The position is described by a 3-dimensional vector containing the coordinates of the origin of a frame in relation to the reference frame. To describe an orientation, a variety of representations exists. Some of these representations contain superabundant coordinates like 3x3 rotation matrices or quaternions. A minimal 3D orientation representation are Z-Y-X Euler angles, which will be used in the following (e.g. [62, Section 1.2.2]). They consist of a vector of three angles, where each of the angles describes a rotation about an axis of a moving frame.
With these definitions the pose can be described by a 6-dimensional vector:

$$\boldsymbol{p} = \begin{bmatrix} p_x & p_y & p_z & \phi_x & \phi_y & \phi_z \end{bmatrix}^\mathsf{T} \tag{7.1}$$

To describe the dynamics of a rigid body, a spatial equivalent of the pose, called wrench, is used. A wrench describes forces and torques acting on the center of gravity of a body. They derive from external forces and torques applied to the body. The wrench can be written as a 6-dimensional vector, where each of its entries describes a force f or torque τ acting in or about an axis of a specified reference frame:

$$\boldsymbol{f} = \begin{bmatrix} F_x & F_y & F_z & \tau_x & \tau_y & \tau_z \end{bmatrix}^\mathsf{T} \tag{7.2}$$

The desired relative motion of two parts during assembly can be seen as following a desired trajectory embedded in a 12-dimensional space spanned by the components of the relative pose and wrench. It consists of a sequence of 12D state vectors describing the relative position, orientation, contact forces, and contact torques between two frames, each attached to one of the parts to be assembled. Using the above definitions the vectors can be rewritten as follows:

$$\boldsymbol{v} = \begin{bmatrix} p_x & p_y & p_z & \phi_x & \phi_y & \phi_z & F_x & F_y & F_z & \tau_x & \tau_y & \tau_z \end{bmatrix}^\mathsf{T} \tag{7.3}$$

A trajectory expressed by such a representation is not explicitly dependent on time. This is a useful property as it allows the representation of a motion independent from time-related properties like motion speed.

It has to be noted, that not every motion needs all 12 dimensions to be specified for its control and supervision. In many cases it is sufficient to consider a subset of these dimensions for supervision. To represent allowed regions instead of strict values, the trajectory can be extended to a hypertube in 12D space by adding tolerance values for each dimension.

In robotics, pose and wrench usually refer to the motion of the end effector frame of the manipulator with respect to a reference frame in the workspace. Here, both frames are usually attached to parts either held by the end effector or located in a fixture in the workspace. One of the frames is called the *task frame*. In this frame, the pose-wrench configuration of the other part is expressed.

7.2.2 Elements of a Trajectory

A trajectory consists of a sequence of trajectory nodes. Each trajectory node consists of a sequence of trajectory states. These definitions refer to a fully parametrized trajectory. A trajectory template, which is described in Section 7.2.5, contains a subset of this information.

Trajectory Node

A trajectory node $\boldsymbol{N_i}$ is defined by :

- A set of 12 node types $\boldsymbol{\eta_{ik}} \in \{\text{unsupervised}, \text{constant}, \text{finiteChange}, \text{infiniteChange}, \text{zero}\}$
- An initial trajectory state $\boldsymbol{S_I}$
- A final trajectory state $\boldsymbol{S_F}$
- An interpolation rule $\boldsymbol{H} \in \{\text{linear}, \text{polynomial}, \text{spline}, \dots\}$ to acquire inner trajectory states

While the initial and final trajectory states $\boldsymbol{S_I}$ and $\boldsymbol{S_F}$ are explicitly set during parametrization, the inner states of the node are acquired by employing an interpolation rule $\boldsymbol{H}$.

Trajectory State

A trajectory state $\boldsymbol{S_i}$ is defined by :

- A state vector $\boldsymbol{v_i} \in \mathbb{R}^{12}$
- A tolerance vector $\boldsymbol{r_i} \in \mathbb{R}^{12}$

The state vector $\boldsymbol{v_i}$ and the tolerance vector $\boldsymbol{r_i}$ specify the represented region in the 12D pose-wrench space. In most cases this region has to be understood as desired or allowed region. The values in a trajectory state refer to a specific task frame defined on the skill level. Consequently it is used for the whole trajectory related to a skill.

Node types

The node types $\boldsymbol{\eta_{ik}}$ specify the qualitative shape of a trajectory without referring to explicit numeric values. A node type assigns one of the values $\boldsymbol{\eta_{ik}} \in \{\text{unsupervised}, \text{constant}, \text{finiteChange}, \text{infiniteChange}, \text{zero}\}$ to each of the 12 dimensions of the trajectory states. Node types are set once for a trajectory node and refer to all trajectory states contained in the node.
They are predefined in the trajectory template. If a qualitative shape descriptor like this is defined in the template, its usage (and thereby the usage of the skill templates) for a class of motions instead of one specific motion is enabled.
The advantage of using node types is a significant reduction of the amount of parameters that need to be set during the creation of a parametrized trajectory. Without their definition, 24 parameters would have to be set for each trajectory node (two trajectory states $\boldsymbol{S_I}$ and $\boldsymbol{S_F}$ with 12 dimensions each). Most importantly, the *unsupervised* node type excludes dimensions from the respective node. Therefore, this dimension can be ignored during parametrization and evaluation of the trajectory. Furthermore, the *constant* type allows to derive trajectory state values from the previous trajectory state. If a type is set to *zero*, the value can be set to zero without further efforts.
Another task of the node types is to define the subset of the node's dimensions that is used for the evaluation of the trajectory. All dimensions with node type *unsupervised* are excluded from the evaluation.
A further advantage of this kind of representation is the great simplification of the mapping rules used to map the trajectory to the states and transitions of a motion net. To define a rule, it is sufficient to define a desired type and use all dimensions of a trajectory state which are of this type for the parametrization. Furthermore, the node types are an indicator of how a node is mapped to the motion net: infinitly changing values indicate a mapping of the node onto a motion net transition, while constant or unsupervised values refer to a motion net state.

7.2.3 Example Trajectory

Node types are visualized using the following example trajectory (Figure 7.2). It has to be noted that the shown trajectory does not refer to any specific assembly task, but is a fictional example. The example describes a process where the only relevant dimensions are the z-component of the position $\boldsymbol{z}$ and the z-component of the contact force $\boldsymbol{F_z}$. These two dimensions form the considered subset for the example. Therefore, dimensions other than these two have the node type $\boldsymbol{\eta_{ik}} =$ unsupervised. Five nodes N_i are featured in the trajectory.

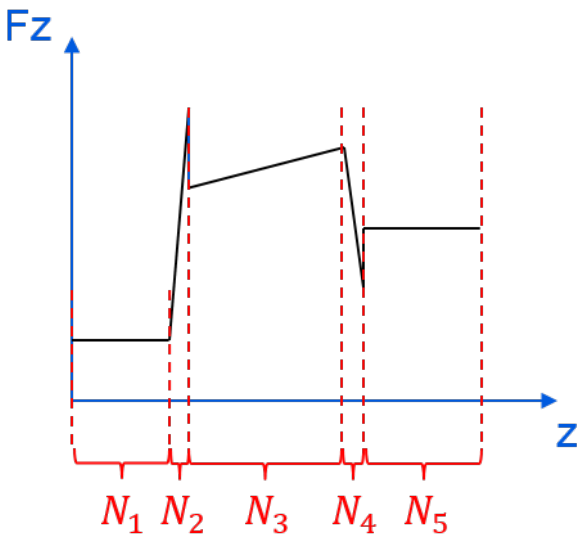


Figure 7.2: Example Trajectory

$\boldsymbol{N_5}$ features a constant, non-zero force in z-direction. For example this could be caused by a constant friction force during a motion in contact with the environment (e.g. sliding a block on a plane).

At nodes $\boldsymbol{N_2}$ and $\boldsymbol{N_4}$ a rapid increase in $\boldsymbol{F_z}$ followed by a sudden decrease can be observed. A scenario that would cause such a force profile is, for example, the occurrence of an obstacle on a movement in z-direction. When contact with the obstacle is made, the force increases rapidly. If the contact force gets too large, the obstacle might be destroyed (e.g. a membrane is ripped apart), which would cause a sudden force decrease.
Node $\boldsymbol{N_3}$ has a steadily increasing $\boldsymbol{F_z}$. This could, for example, be caused by the insertion of a part into a canal with flexible, narrowing walls. The further the part moves in z-direction, the narrower the wall gets, which causes an increase in the contact force.

Node	Node Type	Example Motion
$\boldsymbol{N_1}$	$z =$ finiteChange $F_z =$ constant	Free motion in z-direction, no contact with the environment
$\boldsymbol{N_2}$	$z =$ constant $F_z =$ infiniteChange	Obstacle in z-direction
$\boldsymbol{N_3}$	$z =$ finiteChange $F_z =$ finiteChange	Motion in z-direction with increasing contact force
$\boldsymbol{N_4}$	$z =$ constant $F_z =$ infiniteChange	Obstacle in z-direction
$\boldsymbol{N_5}$	$z =$ finiteChange $F_z =$ constant	Motion in contact with constant contact force

Table 7.1: Nodes in the Example Trajectory

7.2.4 Motion State Evaluation

While the motion net representation of a skill is used as discrete switch between continuous robot motions, the main task of the trajectory representations is to supervise these continuous motions in pose-wrench space. Consequently, the trajectory is continuously evaluated during the execution of a motion by comparing the current state of the assembly with the desired state from the trajectory.

Desired State

The desired state of an assembly task is defined by a specific point in pose-wrench space, which can be expressed by trajectory state $\boldsymbol{S_i}$. This state is acquired from the trajectory node $\boldsymbol{N_k}$, which is linked to the currently active motion net state $\boldsymbol{M_k}$ of the currently executed skill (see Section 7.3.1). For the acquisition of the desired state, the interpolation rule $\boldsymbol{H}$ stored with $\boldsymbol{N_k}$ has to be applied. To apply the interpolation rule, one of the 12 dimensions has to be used to acquire the interpolation point. Ideally, the dimension $\boldsymbol{d}$, in which the difference between the values of the initial and final states $\boldsymbol{S_I}$ and $\boldsymbol{S_F}$ of $\boldsymbol{N_k}$ is at its maximum, is used. This results in the best resolution when interpolating between $\boldsymbol{S_I}$ and $\boldsymbol{S_F}$. When applying the interpolation, the value specified by dimension $\boldsymbol{d}$ of the current assembly state $\boldsymbol{S_C}$ is used as an interpolation point. The desired values in the other 11 dimensions are interpolated at this point.

Current State

The current state of an assembly task can also be expressed in pose-wrench space as a trajectory state $\boldsymbol{S_C}$. For its acquisition the executing system has to be equipped with appropriate sensors. The pose of $\boldsymbol{S_C}$ can usually be acquired by applying the forward kinematics model of the robot manipulator performing the assembly task to sensed angles of its joints. Joint angle sensors are available in all current robotic system. The acquisition of the wrench dimensions is more challenging, since additional sensors (see Chapter 3.4) or an advanced force estimation scheme (see Chapter 10) based on available robot properties are necessary.

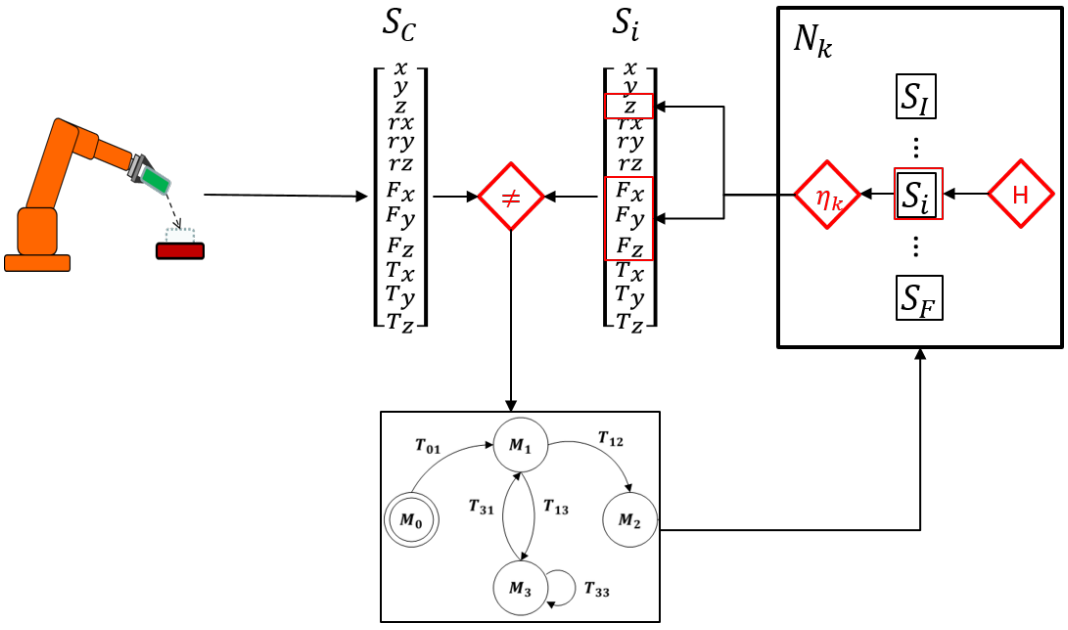


Figure 7.3: Schematic overview of how the current state of the robot S_C is compared to the current desired state S_i, which is derived from the trajectory node N_k

Comparison

To detect errors during execution, the deviation of S_i and S_C is calculated. As it is not always necessary to supervise this deviation in all 12 dimensions, the supervised subspace is defined by the Node types η_{kl} of the current trajectory node N_k. If η_{kd} has the value *unsupervised*, dimension d is excluded from the supervision.
As small deviations might not be problematic, it is necessary to define confidence ranges for each dimension instead of a specific value. These ranges are specified by the tolerance vector r_i of trajectory state S_i. Including these tolerance values, the trajectory becomes a hypertube in pose-wrench space instead of a 12-dimensional "line".
If a deviation between S_i and S_C is detected, S_C is evaluated against the transition conditions of the currently active transitions in the motion net. The transition conditions are also expressed as trajectory states S_{lj} (see Section 7.3.1). If a transition conditon is evaluated as *true* with respect to a certain comparison operator $\square \in \{\leq, \geq, =, <, >\}$, the respective transition fires. This can, for example, indicate an interruption in the motion (the motion net enters an error recovery state) or its completion (the motion net enters its "complete" state). If the execution is already completed, the deviation from the final desired state can be used as a quality indicator (and consequently triggers a "quality check" transition in the motion net).

7.2.5 Trajectory Template and Parametrization

To be reusable in alike situations, the skill template contains the trajectory as a *Trajectory Template*. This template consists of a sequence of *Template Trajectory Nodes*.

Trajectory Template Node

A trajectory template node $N_i^{'}$ contains:

- A set of 12 node types $\eta_{ik} \in \{\text{unsupervised}, \text{constant}, \text{finiteChange}, \text{infiniteChange}, \text{zero}\}$
- An interpolation rule $H \in \{\text{linear}, \text{polynomial}, \text{spline}, \dots\}$ to acquire inner trajectory states

The initial and final trajectory states S_I and S_F are specified during a mapping process by employing the skill template's mapping rule set $R1$.
The trajectory template can be seen as a qualitative shape descriptor of a trajectory. It defines what kinds of nodes are contained in which order and specify the principle characteristics of a trajectory. No specific values of the trajectory are set at this point, thus the trajectory sections represented by the template nodes still have variable lengths and magnitudes. The values are set in the mapping process described in the following.

Mapping Process

A prerequisite to perform the mapping process is that a sequence of skills has been selected, and each skill has been linked to an action or a sequence of actions from an assembly tree task (see Section 6.4).
Each rule contained in mapping rule set $R1$ that is stored in the skill template relates a template trajectory node $N_k^{'}$ to an assembly action a_i. It may be noted that this is not a one-to-one relation and an action a_i can be employed in several mapping rules. Contrary, only one mapping rule is allowed for each node template $N_k^{'}$.
To create a trajectory node N_k, the template node $N_k^{'}$ has to be augmented with initial and final trajectory states S_I and S_F. The trajectory states can be acquired from the targets τ_{il} defined in an action a_i. In the mapping rule it is specified which target should be used to define S_I and S_F. There is one exception to that scheme: if the trajectory does not have a discontinuity between two trajectory nodes, the initial trajectory state of a node can be set according to the final trajectory state of the previous node.

7.3 Motion Net Representation

The *Motion Net* contains the execution logic of the robot actions contained in a skill and is represented as a Finite State Machine (FSM) (e.g. [24]). An FSM is a useful mathematical model employed to design and supervise logical connections. It consists of a finite number of states connected by state transitions, while it can only be in one state at a given time called the active state. Its tasks are the coordination and control of a robot's motions.
In particular, an FSM consists of the following elements:

- States
- Actions
- Transitions

Actions are performed when entering a state (some implementations allow entry and exit actions), while transitions are used to switch between states. In the motion net each state contains a robot action that needs to be performed. To switch between robot actions, transitions are triggered. Accordingly, the motion net elements are referred to as *Motion Net States*, *Motion Net Transitions* and *Motion Net Actions*.

7.3.1 Elements of a Motion Net

The definitions presented here refer to the data present in a motion net of a fully parametrized skill. A skill template contains a subset of the data presented here. For an overview of the data in a skill template, refer to Section 7.3.3.

Motion Net State

A motion net state $\boldsymbol{M_i}$ is defined by:

- A state type $\boldsymbol{\Gamma} \in \{\text{motion}, \text{tool}, \text{initial}, \text{complete}, \text{success}, \text{failure}, \text{goal}\}$
- A robot command type $\boldsymbol{K}$, e.g. $K \in \{\text{linearMotion}, \text{openGripper}, \dots\}$

If the state type Γ is *motion*, a motion net state aditionally contains:

- A set of control values $\boldsymbol{C_i}$
- A trajectory node $\boldsymbol{N_i}$

Furthermore, an action executed upon entering a state can be linked to the state. As it is not contained in the state itself, it is not listed here. In a system with multiple execution units, it has to be specified in the state to whose execution unit it refers.
While the control values $\boldsymbol{C_i}$ are used to control the execution of the action, the trajectory node $\boldsymbol{N_i}$ is used for its supervision.

Motion Net Transition

A motion net transition $\boldsymbol{T_{ij}}$ is defined by:

- An initial motion net state $\boldsymbol{M_i}$
- A final motion net state $\boldsymbol{M_j}$
- A set of n transition conditions $\boldsymbol{t_{ik}}$ (optional)

If all n transition conditions t_{ik} are evaluated as true, transition T_{ij} fires and the final state M_j is the new active state of the motion net. There are *primitive transitions* which do not contain any transition conditions. These transitions fire as soon as their initial state is active.

Motion Net Action

A motion net action linked to a state $\boldsymbol{M_i}$ is executed when the FSM enters this state. The action always consists of the execution of the robot command specified by the robot command type $\boldsymbol{K}$ of state $\boldsymbol{M_i}$. This command can, for example, be a motion or a tool action. For the former, the control values $\boldsymbol{C_i}$ are employed for the execution. It is highly dependent on the executing platform, how the command is executed in detail.

Control Values

A set of control values $\boldsymbol{C_i}$ is defined by:

- A controller setpoint $\boldsymbol{c_i}$
- A task frame $\boldsymbol{TF_i} \in \mathbb{R}^{4x4}$

This definition follows the Task Frame Formalism explained in Section 3.2.1. The task frame is defined in relation to an arbitrary reference frame. Usually this is the world frame in the robot's workspace.
Depending on the employed controller, additional controller parameters have to be stored with the control values. This can, for example, be damping and stiffness values if an impedance controller is employed.

Controller Setpoint

A controller setpoint $\boldsymbol{c_i}$ is defined by:

- A set of six control values $\boldsymbol{\nu_i}$
- A set of six control mode indicators $\boldsymbol{\mu_i} \in \{\text{force}, \text{position}, \dots\}$

The setpoint $\boldsymbol{c_i}$ defines the desired values and control types along and about each of the task frame's axis. It has six entries in total, one for each of the degrees of freedom of $\boldsymbol{TF_i}$. If a degree of freedom is not controlled, it has the control mode *unused*.

Transition Condition

A transition condition $\boldsymbol{t_{ik}}$ is defined by:

- A trajectory state $\boldsymbol{S_{ik}}$ representing the condition
- A comparison function

$$\boldsymbol{h}(S_c \square S_{ik}) \rightarrow \{true, false\} \tag{7.4}$$

  where $\square \in \{\leq, \geq, =, <, >\}$ is a comparison operator and S_c is the current state of the robot.

When a transition condition is evaluated, only the entries l of S_c and S_{ik}, for which the state entry types $\eta_{cl}, \eta_{ikl} \neq$ *unsupervised*, are used for the evaluation.

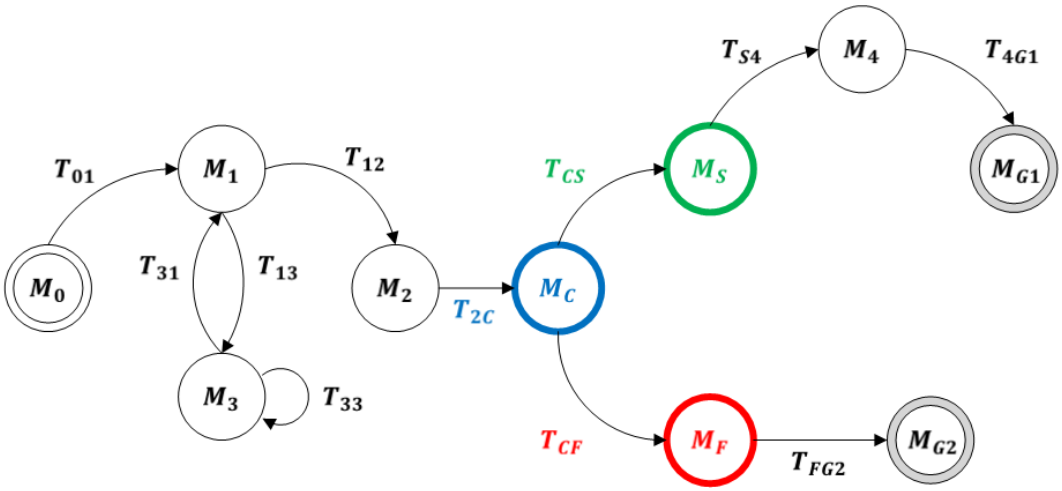


Figure 7.4: States and Transitions in an Example Motion Net

Figure 7.4 shows the typical structure of a motion net in the proposed system. Depending on the current state of the motion net, the behavior of the system changes. There are several different types of states and transitions present in the illustrated motion net:

- M_0 is called the *Initial State* (mandatory: 1)
- M_1 is called a *Motion State* (optional)
- M_2 is called a *Tool State* (optional)
- M_3 is called an *Interrupt State* (optional)
- M_4 is called an *Unproductive State* (optional)
- M_C is called the *Complete State* (mandatory: 1)
- M_S is called a *Success State* (optional)
- M_F is called a *Failure State* (optional)
- M_{G1} and M_{G2} are called *Goal States* (mandatory: 1:N)
- T_{01} is called the *Precondition* (mandatory: 1)
- T_{13} is called an *Interruption Criterion* (optional)
- T_{33} is called a *Retry Transition* (optional)
- T_{2C} is called the *Completion Criterion* (mandatory: 1)
- T_{CS} and T_{CF} are called *Quality Criteria* (optional)

Some of the states are marked as "mandatory", which means they have to be included in every motion net. The following explains, how the system behaves when a state of a certain type is active or a transition of a certain type is fired.

Initial State When a motion net is in the initial state, the precondition is evaluated. The precondition is fired if the current state of the executing unit meets the transition condition contained in it. Consequently, the motion net enters the first regular state of the motion net which is usually a motion state. If the precondition is not met, the execution of the skill's motion net cannot start.

Motion State Motion states represent all kinds of controlled actions a robot performs during the execution of a skill. These actions are usually desired motions on a path towards a goal pose in the workspace. On entering a motion state $\boldsymbol{M_i}$, the related action is executed according to the robot command type $\boldsymbol{K}$ with respect to the control values $\boldsymbol{C_i}$. During the execution, the desired robot state defined by the trajectory node $\boldsymbol{N_i}$ and the current robot state acquired by sensor input are continuously compared (see 7.2.4). If a deviation between the desired and current state is recognized, the outgoing transitions of the motion state are evaluated. A deviation can be undesired - when an error occurs during execution - or desired - to indicate the completion of the execution. In the first case an interruption transition has to be triggered (if one is defined), while in the latter case a transition to the next regular state has to be fired.

Interrupt State An interrupt state is a motion state that was entered through an interruption transition. Its action represents a error recovery motion to return to the desired motion path. As the interrupt state does not represent a distinct state type, it has the same properties as a motion state. A feature mostly used in the context of interruption is the *Retry Transition* (although it is not limited to these states). When this transition is triggered, the state is reentered and its entry action is performed once more. This can, for example, be useful when an error recovery motion did not yield the desired results (that is, the recovery from an error). To prevent a potential deadlock situation in the FSM, the allowed number of retries needs to be limited.

Unproductive State Unproductive states are motion states which are executed after the skill related to the motion net has yielded its result. They occur after the completion transition was triggered and the quality check was performed. The reason for this concept is that there are actions which are not relevant for the successful execution of a task. An example is the retraction of a robot arm.

Tool State A tool state represents a primitive action of a tool attached to a robot. In the proposed concept, tool actions are supposed to be activated by binary information (e.g. "on/off"). Therefore, the execution of a tool state does not need dedicated supervision. It only contains one transition that is fired when the tool action is finished. In general, tool states can also be defined as more complex elements that allow supervision of the current execution state via a trajectory. The simplification was made because tools with appropriate sensing to enable this kind of supervision are still rare and were not available for the work on this thesis.

Complete State The complete state marks the completion of all productive and therefore quality-relevant actions in the motion net. However, this does not mean that it is the final state of the motion net as unproductive states may be present. When the FSM enters the complete state, the quality criteria are evaluated, if defined. According to the result of this quality check, the FSM enters a success or failure state. If no quality criteria are defined, the FSM enters an unproductive state or a final state.

Success State If the quality check is positive, the FSM enters a success state. It is possible to define several success states, if an adequate number of positive quality criteria are defined. Following the success state, an unproductive state or a goal state is entered.

Failure State If the quality check is negative, the FSM enters a failure state. It is possible to define several failure states, if an adequate number of negative quality criteria are defined. Following the failure state, an unproductive state or a goal state is entered.

Goal State A goal state indicates the complete execution of a skills motion net. Upon entering, a transition in the skill net is fired and therefore the execution of the next skill is triggered. If the skill has no successor, the overall execution is finished. It is possible to define multiple goal states and choose the succeeding skill accordingly. For this concept to be useful, the different goal states should be related to the different success and failure states.

7.3.3 Motion Net Template and Parametrization

To be reusable in similar situations, the skill template contains the motion net as a *Motion Net Template*. The motion net template contains states and transitions with limited amount of information compared to the above definitions. They are called *Template States* and *Template Transitions* in the following.

Template State

A template state $\boldsymbol{M_i^{'}}$ is defined by:

- A state type $\boldsymbol{\Gamma} \in \{\text{motion}, \text{tool}, \text{initial}, \text{complete}, \text{success}, \text{failure}, \text{final}\}$
- A robot command type $\boldsymbol{K}$, e.g. $K \in \{\text{linearMotion}, \text{openGripper}, \dots\}$

The control values $\boldsymbol{C_i}$ and the connected trajectory node $\boldsymbol{N_i}$ of the motion net state $\boldsymbol{M_i}$ are specified during a mapping process employing rules from the skill template's mapping rule set $\boldsymbol{R2}$.

Template Transition

A template transition $\boldsymbol{T_{ij}^{'}}$ is defined by:

- An initial template state $\boldsymbol{M_i^{'}}$
- A final template state $\boldsymbol{M_j^{'}}$
- A set of n transition conditions $\boldsymbol{t_{ik}}$

The equivalent motion states $\boldsymbol{M_i}$ and $\boldsymbol{M_j}$ as well as the transition conditions $\boldsymbol{t_{ik}}$ of the motion transition $\boldsymbol{T_{ij}}$ are specified during a mapping process by employing the skill template's mapping rule set $\boldsymbol{R2}$.

The motion net template can be seen as an empty skeleton of a motion net. It defines what kinds of actions are connected in which way, but does not provide concrete values to perform these actions. The values are set in a mapping process described in the following.

Mapping Process

The mapping rule set $\boldsymbol{R2}$ stored in the skill template consits of two subsets $\boldsymbol{R2.1}$ and $\boldsymbol{R2.2}$. The set contains one mapping rule for each state and each transition in the template motion net that requires a mapping. The only states that require mapping are motion states. All other states do not contain actions that employ sensing or control and therefore do not need control values or a trajectory node. All transitions require mapping except primitive transitions, as these do not contain any transition conditions.
Each rule relates a template state $\boldsymbol{M_i^{'}}$ or a template transition $\boldsymbol{T_{ij}^{'}}$ to a trajectory node $\boldsymbol{N_k}$. It may be noted that this is not a one-to-one relation and a node $\boldsymbol{N_k}$ can be employed in several mapping rules. Contrary, only one mapping rule is allowed for each template state or transition.

Subset $\boldsymbol{R2.1}$ contains all rules that map nodes to template states to produce motion net states. To create a motion net state $\boldsymbol{M_i}$, the template state $\boldsymbol{M_i^{'}}$ has to be augmented with control values $\boldsymbol{C_i}$ and a related trajectory node $\boldsymbol{N_i}$. $\boldsymbol{N_k}$ can be acquired directly from the connection between $\boldsymbol{M_i^{'}}$ and $\boldsymbol{N_k}$. To create $\boldsymbol{C_i}$, two decisions have to be made by the mapping rule:

- Which trajectory state is used in the mapping, initial state $\boldsymbol{S_I}$ or final state $\boldsymbol{S_F}$ (D1)
- Which of the 12 dimensions of the used trajectory state are mapped to $\boldsymbol{C_i}$ (D2)

Control mode and value of $\boldsymbol{C_i}$s setpoint are defined by the used dimensions of the trajectory state. The task frame $\boldsymbol{TF_i}$ can be acquired directly from the trajectory state.

Subset **R2.2** contains all rules that map nodes to template transitions to produce motion net transitions. To create a motion net transition $\boldsymbol{T_{ij}}$, the template transition $\boldsymbol{T'_{ij}}$ has to be augmented with transition conditions $\boldsymbol{t_{ik}}$. To create $\boldsymbol{t_{ik}}$, three decisions have to be made by the mapping rule:

- Which trajectory state is used in the mapping, initial state $\boldsymbol{S_I}$ or final state $\boldsymbol{S_F}$ (D1)
- Which of the 12 dimensions of the used trajectory state are mapped to which condition $\boldsymbol{t_{ik}}$ (D2)
- Which operator $\square \in \{\leq, \geq, =, <, >\}$ should be used in the transition condition's comparison function $\boldsymbol{h}(m_c \square m_{ik}) \rightarrow \{true, false\}$ (D3)

The condition values represented by the condition's trajectory state $\boldsymbol{S_{ik}}$ can be acquired directly by the used trajectory. The operator has to be preassigned in the mapping rule.
In both cases decision D2 can be made according to the node types $\boldsymbol{\eta_{ik}}$ of a trajectory state $\boldsymbol{S_{ik}}$. By specifying a desired (or undesired) node type, all state dimensions that are (or are not) of this type can be selected.

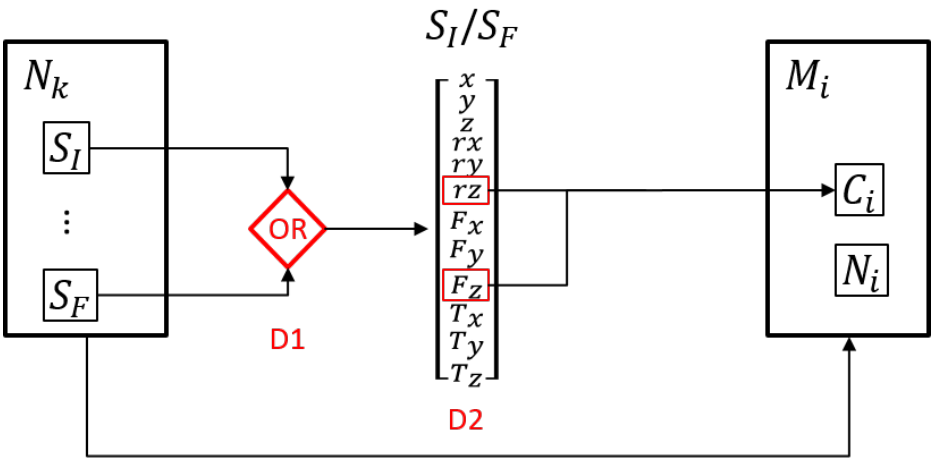


Figure 7.5: Schematic visualization of mapping R2.1, which describes how a motion net state M_i is generated from a trajectory node N_k

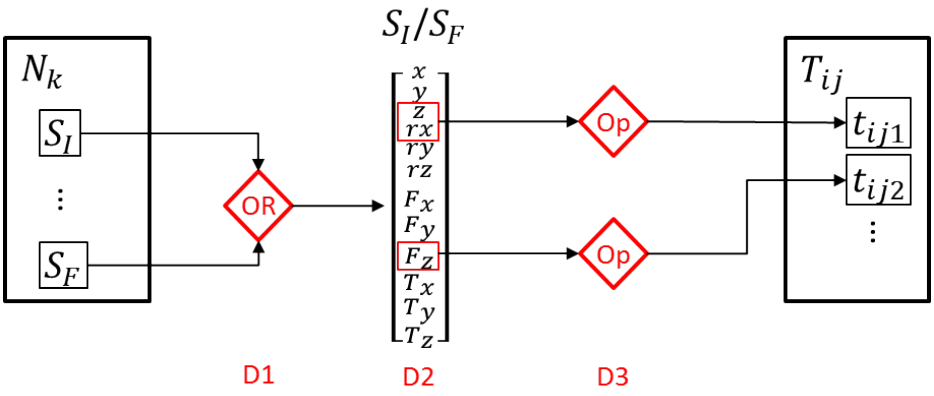


Figure 7.6: Schematic visualization of mapping R2.2, which describes how a motion net transition T_{ij} is generated from a trajectory node N_k

8 Skill Categorization

The skill representation presented in the previous chapter can be used to describe a wide selection of robotic tasks relevant for assembly and beyond that. This chapter gives examples for skills using the previously described representation.
Every skill is presented as a data sheet in this chapter. The data sheet features the most important properties of a skill based on the representation and definition presented in the previous chapters. In the following, the template used for the data sheet is presented with explanations of its categories (Table 8).
In this thesis the skills *Insert* and *Snapfit*, which are discussed in more detail in the following, were mainly investigated. Other skills are presented in brief as data sheets. The skills presented here are used for the example assembly implemented in the demonstrator (see Chapter 11). Further skills necessary for the example assembly application are presented in short in Appendix C. Section 8.3 contains short descriptions of further skills not used in the demonstrator.

<table>
<tr><td>Name</td><td>The name of the skill; should be unique and intuitive</td><td rowspan="3">Symbolic visual representation</td></tr>
<tr><td>Category</td><td>The category of the skill depending on the fulfilled task</td></tr>
<tr><td>Description</td><td>A short description of the skill</td></tr>
<tr><td>Trajectory</td><td colspan="2">A symbolic visualization of the trajectory representation according to Section 7.2</td></tr>
<tr><td>Motion Net</td><td colspan="2">The motion net representation according to Section 7.3</td></tr>
<tr><td>DOF</td><td>The supervised dimensions of the trajectory</td><td>The controlled dimensions of the skill (which are a subset of the supervised dimensions)</td></tr>
<tr><td>Precondition</td><td colspan="2">The precondition according to Chapter 7</td></tr>
<tr><td>Interruptions</td><td colspan="2">Possible interruption conditions according to Chapter 7</td></tr>
<tr><td>Completion</td><td colspan="2">The completion criteria according to Chapter 7</td></tr>
<tr><td>Quality</td><td colspan="2">The quality criteria according to Chapter 7</td></tr>
<tr><td>Resources</td><td>The necessary tools for the execution of the skill</td><td>The necessary sensors for the execution of the skill</td></tr>
<tr><td>Skill Dependencies</td><td>If the skill is a concatenation of other skills, its "subskills" are listed here.</td><td>If the skill has dependencies from other skills that have to be executed in advance, these skills are listed here.</td></tr>
</table>

Table 8.1: Template for data sheets used to describe the properties of skill templates

8.1 Insert Skill

Insertion tasks in robotics refer to the classic peg-in-hole problem which has been widely discussed in literature (e.g. [83]). In insertion, a part is inserted into a fitting hole. Part and hole can have rectangular or round geometries, while the focus here is on the former. Different sub-problems derive from this problem like searching the hole and appropriate alignment of the part. Also, different insertion strategies like tilted insertion in combination with different compliant motion strategies exist. This is omitted here, as these problems are already extensively covered in literature. Here the focus is on a simple insertion task with a rectangular geometry where the part is already correctly aligned above the hole and is inserted straight into it. From the mathematical model of this task, the "Insert" skill is derived.

8.1.1 Mathematical Model of an Insertion Task

In straight insertion, a part is moved in z-direction while resulting forces $\boldsymbol{F}_x$, $\boldsymbol{F}_y$, and $\boldsymbol{F}_z$ are generated by the contact between the part and the hole. The resulting torques are not employed in this analysis and therefore are omitted.
Ideally, if a part is inserted straight into a hole, no contact with the walls occurs except a final contact at the bottom of the hole. In practice, this is not realistic, as positioning uncertainties prevent a perfectly straight insertion. Depending on the geometry, a tilted orientation of the part can cause several contacts with the environment, which can be described by the following contact states. It has to be noted that this analysis was done in two dimension, but all assumptions about $\boldsymbol{F}_x$ can easily be transferred to $\boldsymbol{F}_y$.

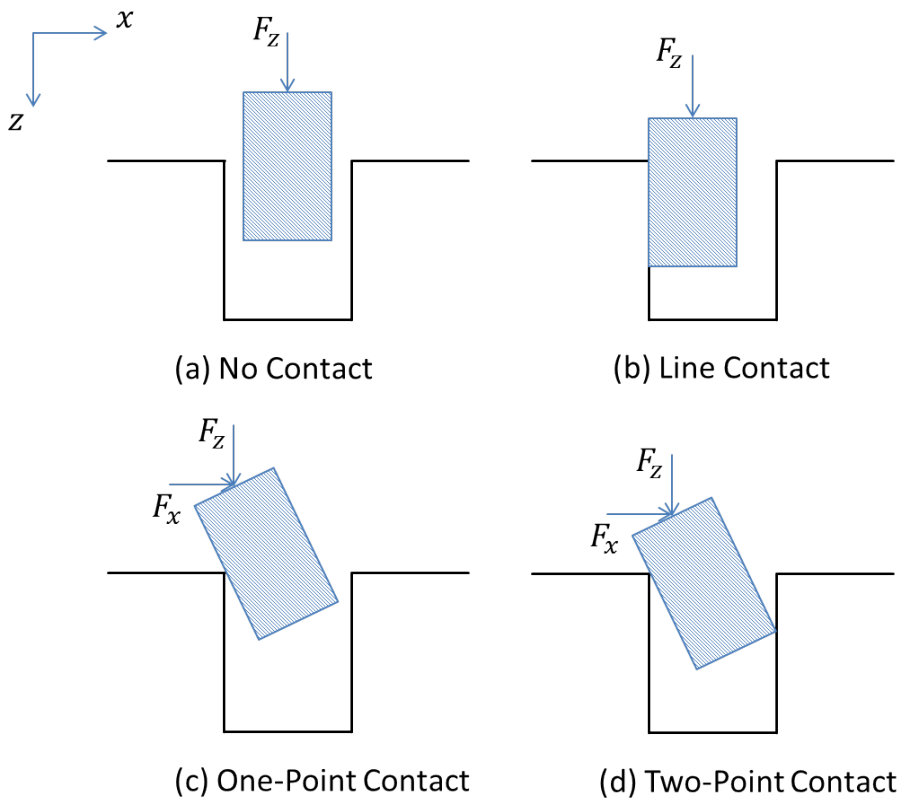


Figure 8.1: Contact states that can occur during the insertion of a rectangular peg into a rectangular hole

For each contact situation the resulting forces $\boldsymbol{F}_x$ and $\boldsymbol{F}_z$ can be calculated. In the following equations [83] the geometric relations shown in Figure 8.2 are employed.

- (a) No contact: no contact forces, $F_x = F_z = 0$
- (b) Line contact: constant wall friction, $F_x = f_a, \quad F_z = \mu f_a$
- (c) One-point contact: like two-point contact with $f_a = 0$ or $f_b = 0$
- (d) Two-point contact: contact forces dependent on tilt angle θ

$$\begin{aligned} F_x &= f_1 \sin\theta + f_2 \cos\theta + k_1 f_a - f_b \\ F_z &= f_1 \cos\theta + f_2 \sin\theta - k_2 f_a - \mu f_b \\ k_1 &= \cos\theta - \mu \sin\theta \\ k_2 &= \sin\theta + \mu \cos\theta \end{aligned} \tag{8.1}$$

In these equations, μ denotes the friction coefficient between the part and the hole. θ is the tilt angle of the part, while f_a and f_b denote the contact forces at the contact points. f_1 and f_2 are the reaction forces in longitudinal and transverse directions of the part.
In contact state (b) an interruption situation called *jamming* can occur, which will prevent the part from further insertion. Jamming is a situation where forces and moments applied to the peg through the support are in the wrong proportions [83]. To detect jamming, the ratios $\frac{T_x}{F_z}, \frac{T_y}{F_z}, \frac{F_x}{F_z}, \frac{F_y}{F_z}$ are analyzed with the help of the jamming diagram (e.g. [83]). A simplified analysis as used in this case is possible: as jamming will always prevent a part from a motion in z-direction, it is sufficient to use a contact force threshold in the F_z dimension for its detection.

To acquire a trajectory from these contact states, contact state (a) or (b) is chosen as desired state for the process, depending on the geometry. This means that the desired trajectory can be expressed by a constant or zero forces F_x,F_y, and F_z throughout the motion in z-direction. If the bottom of the hole is reached, a peak in F_z is detected. When additional force peaks in F_x or F_y are detected, this indicates a two-point contact according to Equation 8.1 instead of a ground contact and therefore a failed insertion. F_x and F_y can consequently be used as a quality indicator. Interruptions can be detected by a peak in F_z according to the jamming analysis described above.

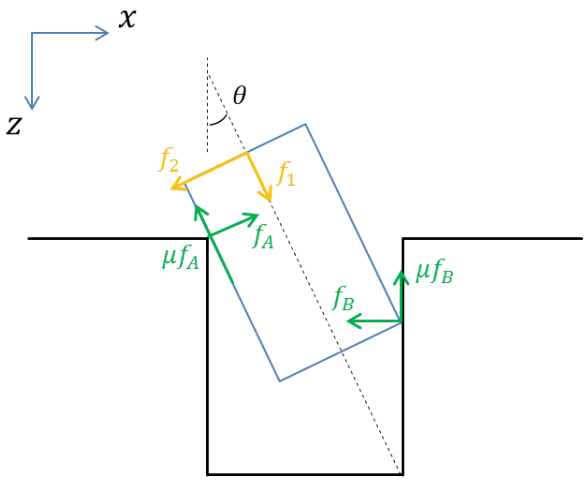


Figure 8.2: Visualization of the contact forces f_a and f_b, and the resulting forces f_1 and f_2 at a two-point contact situation

8.1.2 12D Pose-Wrench Trajectory of the Insert Skill

As described in Section 8.1.1, the trajectory shown in the following is derived from the previously described contact states. For the definition of this trajectory it is assumed that the task frame of the motion is set in such a way that the predominant direction of the motion is $\boldsymbol{z}$ and the geometry allows insertion without wall contact.
It has to be noted that the change in z at node N_2 is infinitely small, since the maximum z position has already been reached.

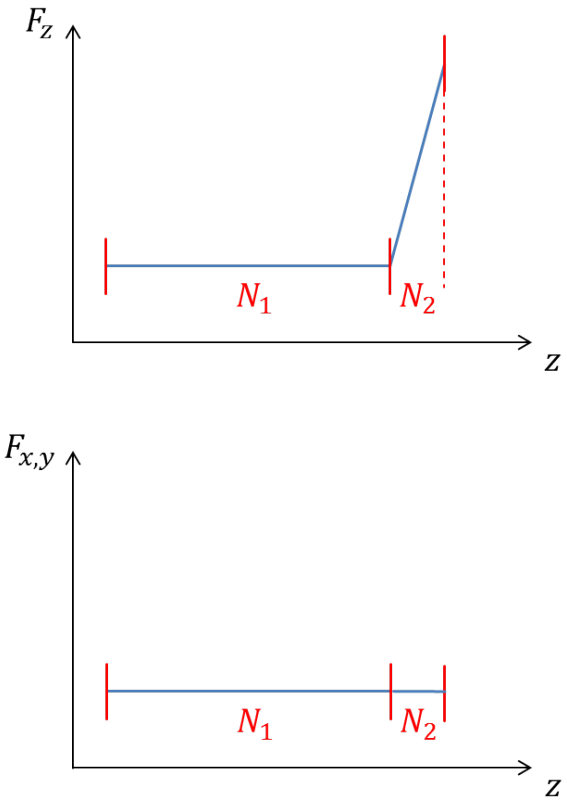


Figure 8.3: Trajectory of the *Insert* skill shown for the dimensions F_x, F_y and F_z

Node	Node Type	Example Motion
$\boldsymbol{N_1}$	$z = \text{finiteChange}$ $F_x = \text{zero}$ $F_y = \text{zero}$ $F_z = \text{zero}$	Free motion in z-direction; no contact with the environment
$\boldsymbol{N_2}$	$z = \text{constant}$ $F_x = \text{zero}$ $F_y = \text{zero}$ $F_z = \text{infiniteChange}$	Bottom of the hole reached; rapidly increasing force

Table 8.2: Explanation of the nodes of the *Insert* trajectory shown in Figure 8.3

8.1.3 Motion Net Finite State Machine of the Insert Skill

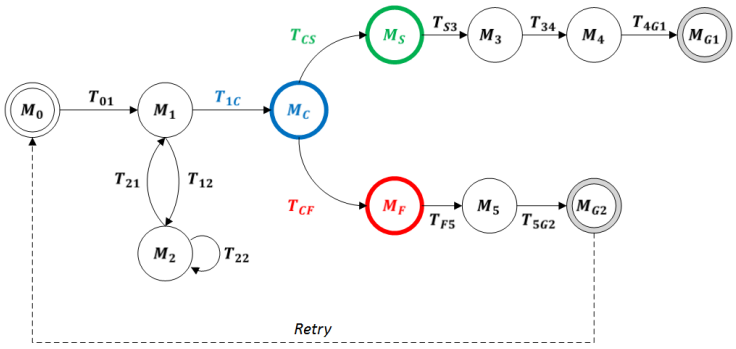


Figure 8.4: Motion Net of the *Insert* Skill

State/Transition	Type	Control Value/Condition	Referring Node
M_0	Initial	-	-
T_{01}	Precondition	$t_{01} : S_{C,pos} = S_{I,pos}$	N_1
M_1	Motion	$C_1 = [\text{-}, \text{-}, z_d, \text{-}, \text{-}, \text{-}]$	N_1
T_{12}	Interruption Criteria	$t_{12} : S_{C,F_{x,y,z}} > S_{i,F_{x,y,z}}$	N_1
M_2	Motion	$C_2 = [\text{-}, \text{-}, z_d, \text{-}, \text{-}, \text{-}]$	N_1
T_{21}	-	$t_{21} : S_{C,F_{x,y,z}} = S_{i,F_{x,y,z}}$	N_1
T_{1C}	Completion Criteria	$t_{1C} : S_{C,F_z} >= S_{F,F_z}$	N_2
M_C	Complete	-	-
T_{CS}	Quality Criteria Success	$t_{CS} : S_{C,F_{x,y}} <= S_{F,F_{x,y}}$	N_2
M_S	Success	-	-
T_{CF}	Quality Criteria Failure	$t_{CF} : S_{C,F_{x,y}} > S_{F,F_{x,y}}$	N_2
M_F	Failure	-	-
T_{S3} / T_{F5}	-	Always True	-
M_3	Tool (unproductive)	Open Gripper	-
T_{34}	-	Gripper Open	-
M_4	Motion (unproductive)	$C_4 = [\text{-}, \text{-}, z_d, \text{-}, \text{-}, \text{-}]$	N_1
T_{4G1}	-	$S_{C,pos} = S_{I,pos}$	N_1
M_5	Motion (unproductive)	$C_4 = [\text{-}, \text{-}, z_d, \text{-}, \text{-}, \text{-}]$	N_1
T_{5G2}	-	$S_{C,pos} = S_{I,pos}$	N_1
M_{G1} / M_{G2}	Goal	-	-

Table 8.3: Description of the States and Transitions in the *Insert* Motion Net

8.1.4 Insert Data Sheet

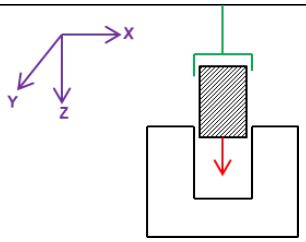
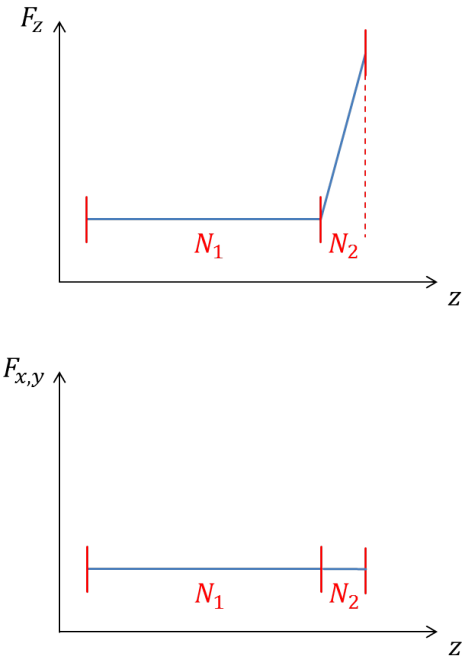
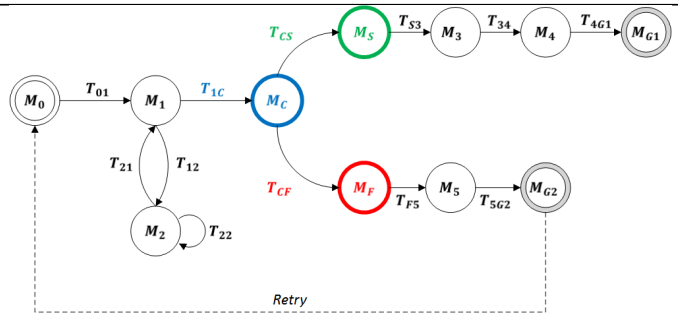
Name	Insert	
Category	Assembly	
Description	Skill to fulfill insertion tasks where a part is inserted straight into a hole (no alignment)	
Trajectory		
Motion Net		
DOF	Supervised : z, F_x, F_y, F_z	Controlled : z
Precondition	Start position reached	
Interruptions	$F_{x,y,z} > F_{x,y,z,desired}$	
Completion	$F_z = F_{z,desired}$	
Quality	Success: $F_{x,y} < F_{x,y,threshold}$, failure: $F_{x,y} > F_{x,y,threshold}$	
Resources	Tools: gripper	Sensors: force, position
Skill Dependencies	Subskills: none	Preceeding skills: none

Table 8.4: *Insert* Data Sheet

8.2 Snapfit Skill

The *Snapfit* skill is designed to fulfill tasks that involve the handling of snap-fit mechanisms. Snap-fits are mechanical joint systems where part-to-part attachment is accomplished by a deflecting latch that is snapped into a hole [8] under the influence of an applied force. Mostly this is realized using plastic parts. Mechanisms like this can be found in assembly tasks as well as in every day live (e.g. the battery cover of a remote control). There is a wide variety of snap-fit types such as cantilever hooks (see Figure 8.5), compression hooks, or torsional snaps. A detailed discussion about the different types of snap-fits can be found in [8], for example.
The mechanical design of the snap-fit is crucial, especially when the snap-fit is performed by a robot. An example of how to ease a robotic snap-fit operation is the mechanical guidance of the latch. Such design choices are necessary in order to reduce the risk for unforeseen situations during assembly. While manipulation skill for robotic assembly is able to handle uncertainty to some extent, it is still desired to keep the assembly as simple as possible.
To create a skill for snap-fit tasks, it is necessary to create an abstracted representation of the tasks according to the skill definition. As there is a great variety of snap-fit types, the abstraction can be very challenging.
Snap-fits may consist of multiple deflective latches instead of a single one, which introduces an additional level of complexity to the abstraction. Multiple snaps can occur at the same time or consecutive, depending on how the force is applied. If the force is applied in the middle of the part, the latches snap in an undefined order or even simultaneously, which makes an abstraction of the process nearly impossible. The solution employed in this case is to divide a multi-latch snap into multiple single-latch snaps. This is accomplished by applying consecutive forces near the latches to gain a defined order of snaps.
In the following sections a mathematical model to describe snap-fit processes is presented, from which a trajectory and motion net representation necessary to create a skill are derived.

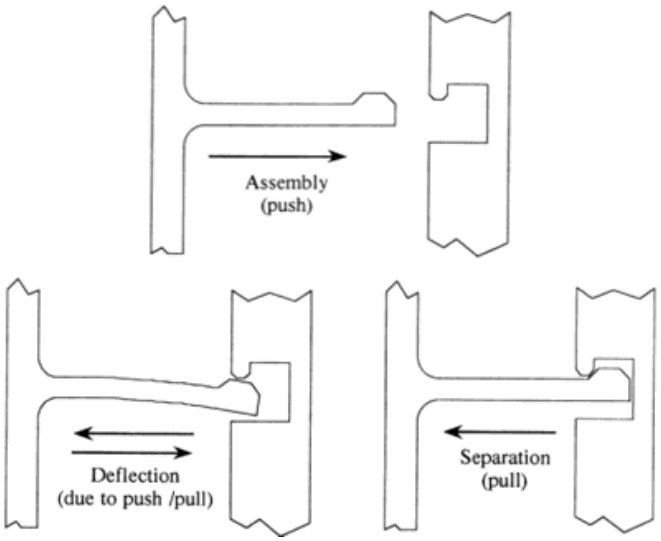


Figure 8.5: Snap-fit working principle shown on a cantilever hook [1]

8.2.1 Mathematical Model of a Single Latch Snap-Fit Operation

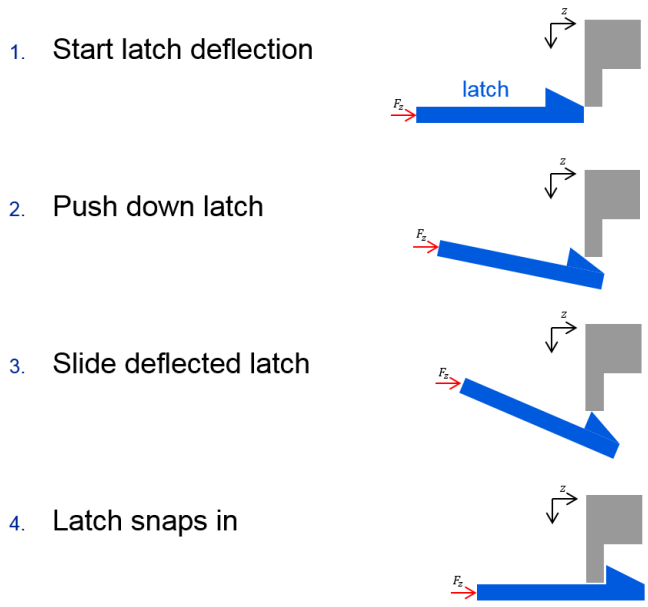


Figure 8.6: Motion Scheme of a Single-Latch Snap-Fit [78]

For the modeling of a snap-fit operation, a single latch that is snapped into the respective opening of a part is considered. The latch is moved in z-direction, while a resulting force $\boldsymbol{F_z}$ is generated by the contact between the latch and the part. This motion can be visualized by the scheme shown in Figure 8.6.
To estimate the resulting force $\boldsymbol{F_z}$, the deflection of the latch is described as a beam bending problem. The latch is therefore described as a rectangular beam for which the following bending formula is valid:

$$x = \frac{l^3}{3EJ} F_x, \quad J = \frac{1}{12} a b^3 \tag{8.2}$$

In this equation $\boldsymbol{x}$ describes the deflection of the beam under the influence of the force $\boldsymbol{F_x}$. $\boldsymbol{J}$ denotes the inertia tensor of the beam while $\boldsymbol{a}$,$\boldsymbol{b}$ and $\boldsymbol{l}$ denote its geometric dimensions width, height and length. $\boldsymbol{E}$ is the modulus of elasticity. Figure 8.7 shows the geometric composition of the bending problem in detail.

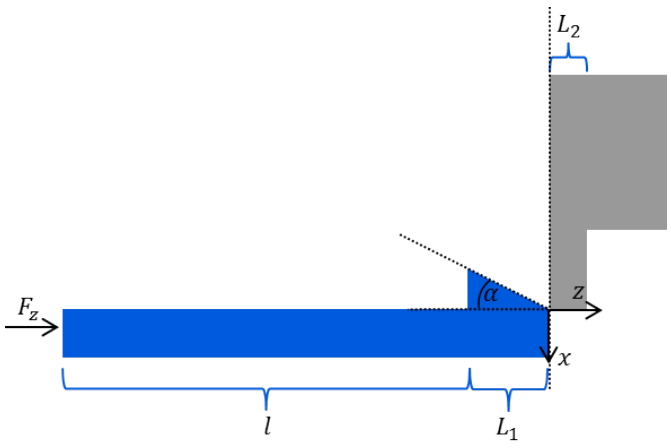


Figure 8.7: Geometric Composition of the Snap-Fit [78]

The force $\boldsymbol{F}_z$ can be calculated from the deflection force $\boldsymbol{F}_x$ by using the relationship of the friction between the latch and the guidance described by

$$F_R = \mu F_N, \quad \mu = \tan\beta\,, \tag{8.3}$$

where μ is the coefficient of friction, $\boldsymbol{F}_R$ the resulting force caused by friction and $\boldsymbol{F}_N$ the normal force. In order to get $\boldsymbol{F}_z$, $\boldsymbol{F}_R$ is partitioned into its x and z components, which leads to

$$F_Z = F_X \frac{\mu + \tan\alpha_{eff}}{1 - \mu\tan\alpha_{eff}} = F_x \mu_{eff}\,. \tag{8.4}$$

In this equation μ_{eff} denotes the effective coefficient of friction in z-direction. It has to be noted that the effective angle α_{eff}, and therefore also μ_{eff}, increases due to the bending of the beam according to the position in z:

$$\alpha_{eff}(z) = \alpha + \arctan\frac{z\tan\alpha}{l + L_1 - z}\,. \tag{8.5}$$

When the latch is fully deflected (see 3. in Figure 8.6), $\boldsymbol{F}_N$ coincides with $\boldsymbol{F}_x$, and $\boldsymbol{F}_R$ coincides with $\boldsymbol{F}_z$. Therefore, μ can be used instead of μ_{eff}.
By joining Equations 8.4 and 8.2, the resulting model for $\boldsymbol{F}_z$ can be written as follows:

$$F_z(z) = \begin{cases} 3\mu_{eff}(z)EJ\cdot\tan(\alpha)\frac{z}{(l+L_1-z)^3} & : 0 < z \le L_1 \\ 3\mu EJ\cdot\tan(\alpha)\frac{L_1}{l^3} & : L_1 < z \le L_1 + L_2 \\ 0 & : \text{otherwise} \end{cases} \tag{8.6}$$

This model can be employed to plot $\boldsymbol{F}_z$ against $\boldsymbol{z}$ using example values as shown in Figure 8.9.

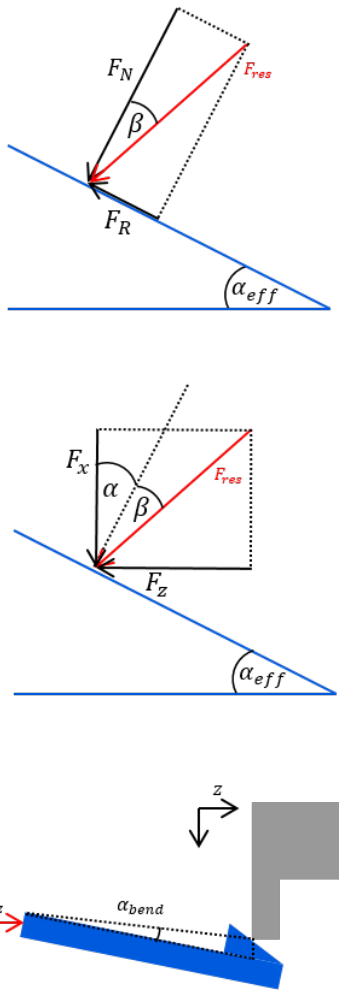


Figure 8.8: Friction Geometry [78]

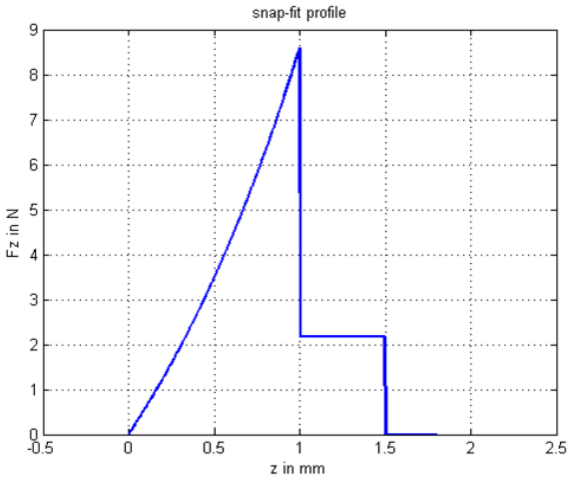


Figure 8.9: MATLAB plot of Equation 8.6 using the example values $a = 5mm$, $b = 1mm$, $l = 10mm$, $L_1 = 1mm$, $L_2 = 0.5mm$, $alpha = 30$, $E = 10GPa$, $mu = 0.3$ [78]

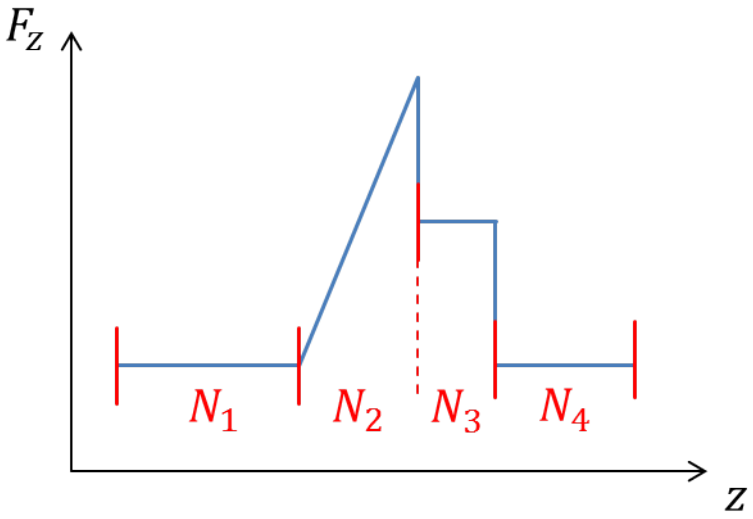


Figure 8.10: Trajectory of the *Snapfit* Skill

From this model the trajectory can easily be acquired. For the definition of this trajectory it is assumed that the task frame of the motion is set in such a way that the predominant direction of the motion is z. This assumption leads to the conclusion that all dimensions except z and $\boldsymbol{F}_z$ are of the type *unsupervised*.
It has to be noted, that for most geometries N_2 and N_3 can be merged into one trajectory node. The reason for this is that the motion phase referring to N_3 (3. in Figure 8.6) is usually very short and can be omitted.

Node	Node Type	Example Motion
$\boldsymbol{N_1}$	z = finiteChange F_z = zero	Free motion in z-direction; no contact with the environment
$\boldsymbol{N_2}$	z = finiteChange F_z = finiteChange	Bending of the latch; increasing force
$\boldsymbol{N_3}$	z = finiteChange F_z = constant	Latch fully bent; constant wall friction
$\boldsymbol{N_4}$	z = constant F_z = zero	Latch snapped in; contact force released

Table 8.5: Description of the nodes in the *Snapfit* trajectory shown in Figure 8.10

8.2.3 Motion Net Finite State Machine of the Snapfit Skill

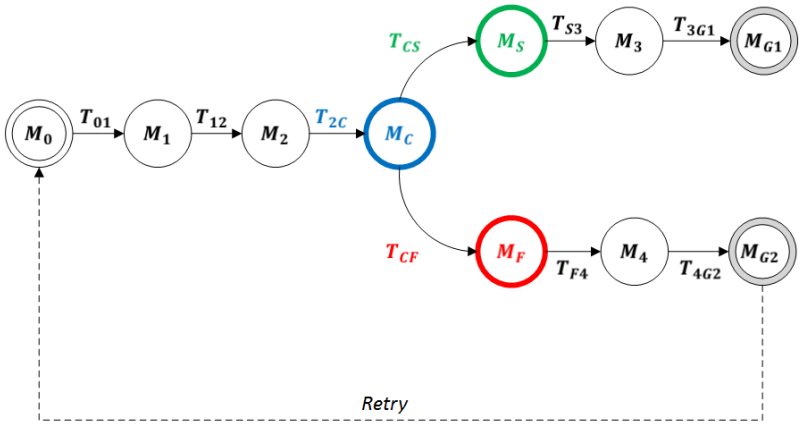


Figure 8.11: Motion Net of the *Snapfit* Skill

This trajectory is mapped to the motion net shown in Figure 8.11 to enable the execution of the *Snapfit* skill.

State/Transition	Type	Condition/Control Value	Referring Node
M_0	Initial	-	-
T_{01}	Precondition	$t_{01} : S_{C,pos} = S_{I,pos}$	N_1
M_1	Motion	$C_1 = [\text{-}, \text{-}, z_d, \text{-}, \text{-}, \text{-}]$	N_2
T_{12}	-	$t_{12} : S_{C,F_z} = S_{F,F_z}$	N_2
M_2	Motion	$C_2 = [\text{-}, \text{-}, F_{zd}, \text{-}, \text{-}, \text{-}]$	N_3
T_{2C}	Completion Criteria	$t_{2C} : S_{C,F_z} < S_{F,F_z}$	N_3
M_C	Complete	-	-
T_{CS}	Quality Criteria Success	$t_{CS} : S_{C,F_z} <= S_{I,F_z}$	N_4
M_S	Success	-	-
T_{CF}	Quality Criteria Failure	$t_{CF} : S_{C,F_z} > S_{I,F_z}$	N_4
M_F	Failure	-	-
T_{S3}	-	Always True	-
T_{F4}	-	Always True	-
M_3	Motion (unproductive)	$C_3 = [\text{-}, \text{-}, z_d, \text{-}, \text{-}, \text{-}]$	N_4
M_4	Motion (unproductive)	$C_4 = [\text{-}, \text{-}, z_d, \text{-}, \text{-}, \text{-}]$	N_4
T_{3G1}	-	$t_{3G1} : S_{C,pos} = S_{F,pos}$	N_4
T_{4G2}	-	$t_{4G2} : S_{C,pos} = S_{F,pos}$	N_4
M_{G1} / M_{G2}	Goal	-	-

Table 8.6: Description of the States and Transitions in the *Snapfit* Motion Net

8.2.4 Snapfit Data Sheet

The most important data can be summarized in the following data sheet according to the template described in 8.

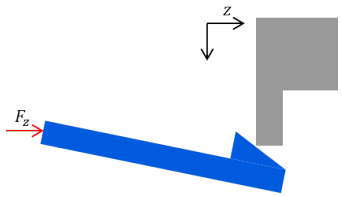
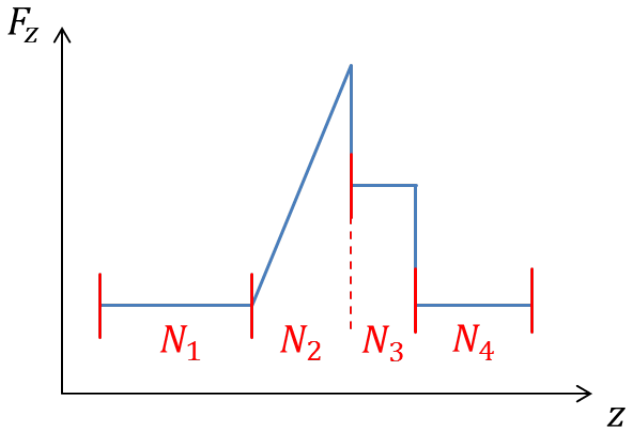
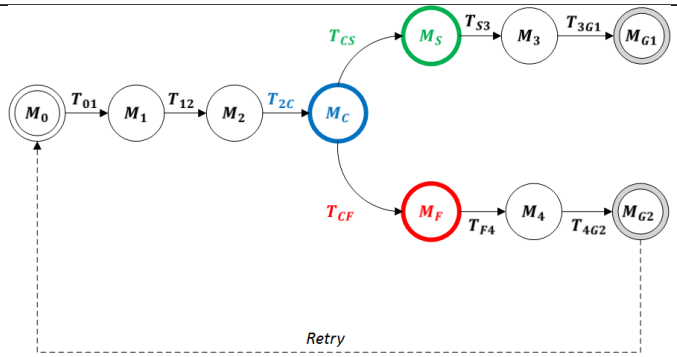
Name	Snapfit	
Category	Assembly	
Description	Skill to fulfill snap-fit tasks involving a single bending latch	
Trajectory		
Motion Net		
DOF	Supervised : z, F_z	Controlled : z, F_z
Precondition	Start position reached	
Interruptions	None	
Completion	Rapid decrease in F_z	
Quality	Success: $F_z <= F_{z,desired}$, failure: $F_z > F_{z,desired}$	
Resources	Tools: none	Sensors: force, position
Skill Dependencies	Subskills: none	Preceeding skills: none

Table 8.7: *Snapfit* Data Sheet [77]

8.3 Other Skills

The skills presented in this chapter are further skills that are defined using the previously presented manipulation skill concept. They are presented in short by means of the data sheet template described in the beginning of this chapter. The skills presented here were not implemented in the demonstrator or used in the example assembly application, both described in Part III.

Bayonet Mount

The *Bayonet Mount* skill is used to handle bayonet like mounts, which are, for example, used to install lightbulbs. This skill is a combination of the subskills *Insert* and *Search Rotational Contact*. It has to be noted that the latter skill is not described in this thesis.

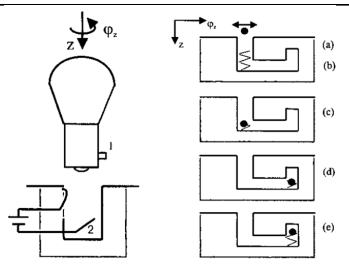
Name	Bayonet Mount	
Category	Assembly	
Description	Skill to fulfill tasks where a bayonet mount is closed using an "Insert" and a "Search rotational contact" skill	
Trajectory	Concatenation of the "Insert" and the "Search Rotational Contact" trajectories	
Motion Net	Concatenation of the "Insert" and the "Search Rotational Contact" motion nets	
DOF	Supervised : z, ϕ_z, F_z, τ_z	Controlled : z, ϕ_z
Precondition	Start position reached	
Interruptions	Interruption conditions from "Insert" and "Search Rotational Contact"	
Completion	$\tau_z > \tau_{z,desired}$	
Quality	Success: $z \geq z_{desired}$, failure: $z < z_{desired}$	
Resources	Tools: gripper	Sensors: position, force, torque
Skill Dependencies	Subskills: Insert, Search Rotational Contact	Preceeding Skills: none

Table 8.8: *Bayonet Mount* Data Sheet [71]

Screw

The *Screw* skill is used to tighten a screw that has already been positioned in a screwhole. It is mainly presented as control and supervision in the rotational dimenions is used. A variation of this skill is the tightening of a nut on a screw.

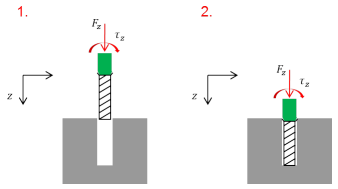
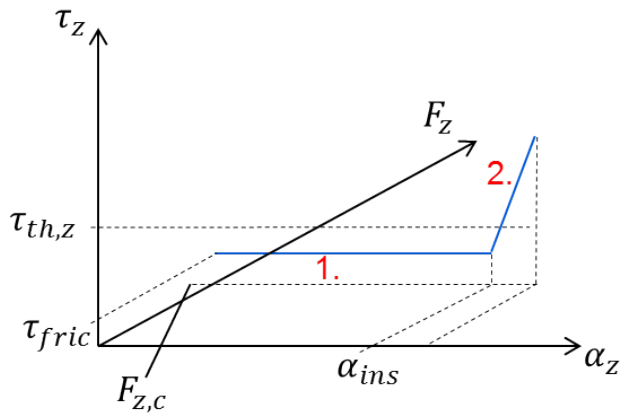
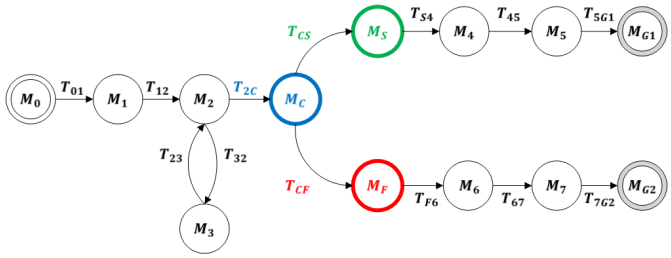
Name	Screw	
Category	Assembly	
Description	Skill to fulfill tasks where a screw is screwed into a hole	
Trajectory		
Motion Net		
DOF	Supervised : z, ϕ_z, F_z, τ_z	Controlled : z, ϕ_z
Precondition	Start position reached	
Interruptions	$\tau_z > \tau_{z,interruption}$	
Completion	$F_z > F_{z,desired}$	
Quality	Success: $\tau_z \geq \tau_{z,desired}$, failure: $\tau_z < \tau_{z,desired}$	
Resources	Tools: screw bit	Sensors: position, force, torque
Skill Dependencies	Subskills: none	Preceeding Skills: Screw positioning

Table 8.9: *Screw* Data Sheet [77]

Part III
Application Results

9 Experimental Setup

In this chapter, the laboratory setup used for the implementation of the assembly skill system is introduced. For the implementation of the demonstrator, an ABB dual-arm concept robot (DACR), which is controlled by an ABB IRC5 robot controller, is used. The robot can be accessed by a standard PC via different communication interfaces.

9.1 ABB Dual-Arm Concept Robot

The dual-arm concept robot is a prototypical robot design by ABB [35]. Its application area is small part assembly, for example, the assembly of consumer electronics. To be suited for assembly tasks like this, it is necessary that the robot can work in mixed human-robot environments.
As can be seen in Figure 9.1, the robot design is highly inspired by a human torso in its dimensions as well as in its kinematic abilities. The reason for this is to maintain maximal compatibility, safety and acceptance when the robot is embedded into a human-centered workspace.
The most crucial analogy to the human body are the two manipulator arms. They have seven degrees of freedom (DOF) each and can operate completely independent of each other. With one additional DOF compared to the six spatial DOF of Cartesian space, kinematic redundancy is introduced. This redundancy can be used in order to choose between different joint configurations to reach a motion goal. For human robot collaboration, this can, for example, mean that the robot reorients its arm while maintaining a certain motion to avoid collision with a human entering the workspace. In general, a much higher flexibility can be gained by kinematic redundancy, which allows the robot to work in a more constrained environment.
An important aspect for guaranteeing the safety of the robot and its surroundings is the limitation of the maximum forces it can exert. This is mainly accomplished by limiting the allowed payload of each manipulator to 0.5 kg and the resulting low power of the drives. Due to this requirement, the moving parts of the mechanism all have low inertia. The tool center point (TCP) speed is limited to 1.5 m/s during normal operation.
Another factor of the safety concept is the inherently safe design of the parts themselves. By leaving enough clearance in the geometric design of the manipulator links and applying soft foam padding to them, the risk of injury when working with the robot is minimized.

Figure 9.1: ABB Dual-Arm Concept Robot with Operator [35]

Within each arm there are electric, pneumatic, and communication (LAN) connections to each of the joints and to the tool. While there is a specifically designed multi-purpose gripper for the system, all kinds of pneumatic and/or electrically powered tools can be attached to the wrist of the robot. In the current setup a simple "open/close" clamp is employed.

The fact that all the wires are on the inside of the arm is a great benefactor to safety. Furthermore, the robot controller as well as the motor power electronics, which are normally in an external device, are inside the robot body. The low weight of this unit and the handles on its backside make sure that a single person can carry the robotic system and deploy it on a workstation.
The robotic system has already been used in various projects and experiments, mostly in the context of human-robot collaboration and assembly.

9.2 ABB IRC5 Robot Controller

The IRC5 controller is used to operate the ABB dual-arm concept robot as well as ABB's conventional industrial robots. It has a modular design to make it applicable to a wide range of different robot models. For each model different power electronic modules, called drive packs, can be included to provide power to the robots motors.
To connect the controller to the robot and additional devices like sensors, serial and ethernet connections are available by default. Additional field buses like PROFIBUS can be added if necessary. The DACR is connected to its controller by ethernet.
The controller runs a real-time operating system that executes different robot control applications. To program the robot movements, the RAPID programming language is used. With this high-level language developed by ABB, positions and other properties of the robot motion can be defined. It can be interpreted by the controller and translated to movement commands that can be executed by the motion planning kernel. This motion kernel computes trajectories needed to reach the programmed motion paths in real-time and sends control signals to the drive packs. To compute the trajectories, the kernel uses optimization based on advanced dynamic modeling. As optimization goals either the shortest cycle time (*QuickMove* option) or the most precise path accuracy (*TrueMove* option) is used.
There are two options for the user interface to the controller: the *Flex Pendant* terminal or a PC with the control and simulation software ABB RobotStudio. While the former is mostly used for manual robot control (jogging) and small modifications on the shopfloor, the latter is an extensive software package containing a RAPID development environment and a graphical design interface with simulated robotic systems. The *Flex Pendant* is directly connected to the controller, while the RobotStudio connection is established via ethernet.
With the *Multi Move* feature up to four robots can be controlled by a single controller. This is very important for the usage in the DACR, as its two arms are each represented as separate robots.
The real-time architecture of the controller allows an interruption-free execution of the robot motions. If motions are influenced from outside of the controller, as necessary in the proposed manipulation skill system, a communication delay in the range of 10-100 ms has to be considered.

9.3 Setup of the Example Assembly Application

This section describes the example assembly process performed by the previously described robot to demonstrate the capabilities of the manipulation skill system. The example assembly refers to the example assembly tree presented in Chapter 6.5. It consists of small plastic parts that are assembled to an *ABB PLC I/O Module*.
The following parts are involved in the assembly:

- A: Three *Printed Circuit Boards* (PCB)
- B: *Housing*
- C: *Cover*
- D: *Light Guide*
- E: *Light Guide Cover*

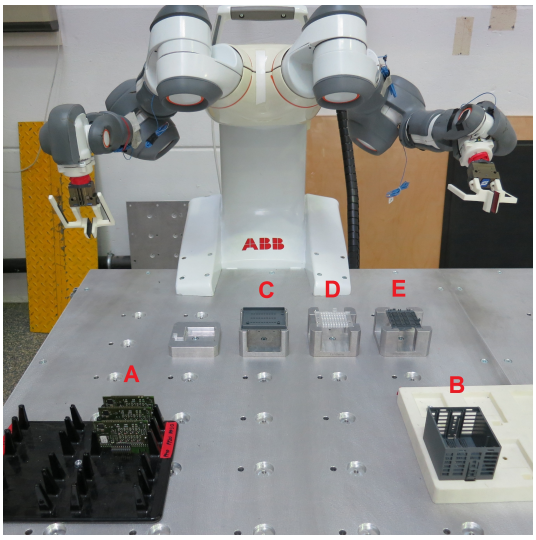


Figure 9.2: Workstation with Assembly Parts

In figure 9.2, these parts are shown in their initial position at the workstation before the assembly is performed.
All parts are placed within fixtures that are in the reachable area of the robot's arms. Both of these arms can be used in the assembly. Each arm has a gripper suited to grasp and hold the involved parts.
The steps the robot needs to perform to create a complete *ABB PLC I/O Module* from these components are described in the following.

Step 1 : Provide Housing
At first, the housing has to be transferred to the empty fixture in front of the robot. This motion is purely position-controlled and has no special requirements. The following operations have to be performed in this step:

- Pick up housing
- Transfer housing to fixture
- Place housing in fixture

Step 2 : Insert PCBs
To insert the PCBs into the housing, three identical steps have to be carried out. They consist of the following operations:

- Pick up PCB
- Transfer PCB to housing
- Insert PCB in housing

During the insertion of the PCBs, contact between the PCB and the housing is established (see Figure 9.3 (a)). As the geometry only allows a small clearance between the two components during insertion, the contact situation can become critical ("jamming", see Chapter 8.1). To prevent this, the contact force has to be controlled.

Step 3 : Install Light Guide

The light guide is installed by placing it on the cover plate, where it is kept in place by two plastic pegs (see Figure 9.3 (b)). For an easier placement, these pegs are chamfered. The following position-controlled operations have to be performed in this step:

- Pick up light guide
- Transfer light guide to cover
- Place light guide on cover

In order to enable the gripper to hold the light guide during the assembly, the part was designed with a handle at its side. As this handle is very small, the gripper has to be positioned with high precision.

Step 4 : Install Light Guide Cover

To install the light guide cover, it is placed on the cover plate and tightened by two snap-fits (see Figure 9.3 (c)). For an easier placement, the previously attached light guide contains a guidance for the light guide cover. The following operations have to be performed in this step:

- Pick up light guide cover
- Transfer light guide cover to cover plate
- Place light guide cover on cover plate
- Close snap-fits of light guide cover

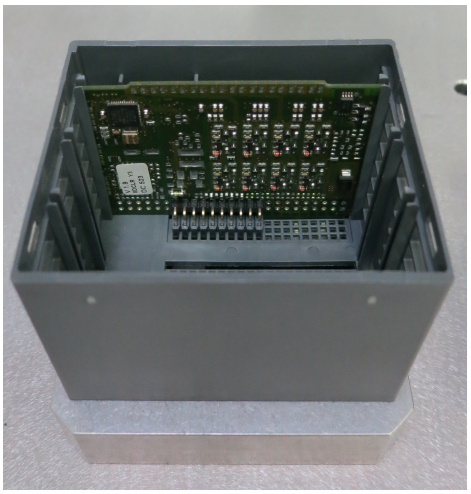
To close the snap-fit, the gripper pushes the left and the right latch down with its tip. This is done successively for one latch at a time to produce clearly defined contact states (see Chapter 8.2). During the snap-fit operations, the contact force is supervised.

Step 5 : Install Cover

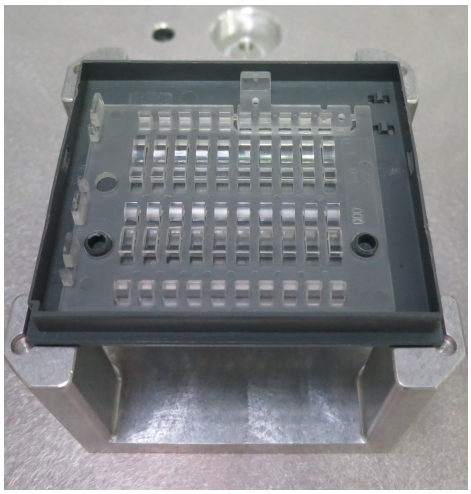
The last step is to install the cover. To do so, it is placed on the housing and tightened by a snap-fit consisting of four latches that are closed simultaneously (see Figure 9.3 (d)). The following operations have to be carried out in this step:

- Pick up cover
- Transfer cover to housing
- Place cover on housing
- Tighten cover by snap-fit

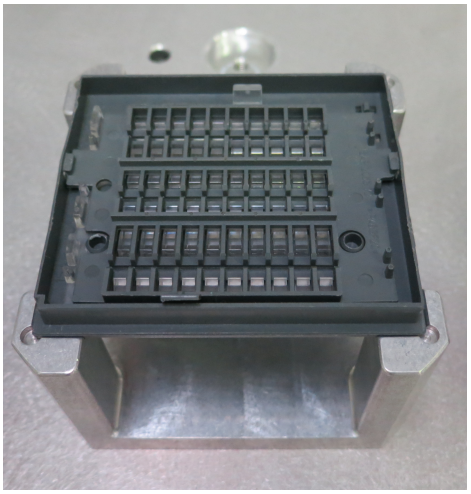
The snap-fit is very powerful as it is performed with the robot's arm padding. This reduces the precision requirements to the placement of the cover, as it is pushed into place by high operational forces combined with mechanical guidance. The high forces also eliminate the need for successive performance of the snap-fits as done in step 4. For the picking up of the cover, higher precision is required, as it is held at its border, which is very narrow.



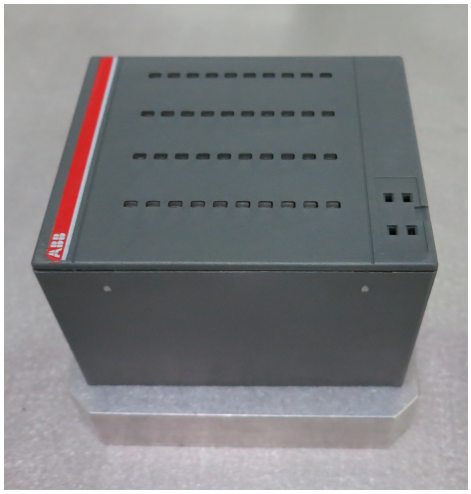
(a) Housing with Inserted PCB



(b) Light Guide on Cover Plate



(c) Light Guide Cover on Cover Plate



(d) Assembled *ABB PLC I/O Module*

Figure 9.3: Goal state of the parts involved in the *ABB PLC I/O Module* assembly after the assembly steps 2-4

10 Contact Force Estimation

This chapter introduces a method to estimate the external wrench without force/torque sensors. The results presented here were published in [79].

10.1 Schematic Overview and Previous Work

For tasks in robotic assembly a basic requirement is the capability of the robotic system to detect contacts with the environment. While this information can be obtained from dedicated force-torque sensors, it might either be technically difficult to mount such sensors or too expensive to do so. An alternative approach is presented here. It reconstructs the external forces and torques based on signals that can be obtained without a force sensor, namely motor currents/torques and joint angles. The approach is visualized in Figure 10.1.
To calculate external contact forces, firstly the motor torques caused by the contact forces have to be extracted from the measured motor torques. To do so, the motor torques caused by friction have to be estimated. A simple friction model, which has to be identified offline, is employed. From the external motor torques the contact forces can be calculated by employing the robot's Jacobian. Prior knowledge of the contact forces is included in the calculation by employing weighting matrices which are calibrated offline via a defined calibration procedure.
The method proposed here can be regarded as an extension of [44, 66]. Here, methods for torque-based contact force estimation are proposed. Only joint angles and motor torques that directly depend on motor currents are used as input data, while prior knowledge of the variance of contact forces and measured motor torques is incorporated. The main novelty of the approach presented here is that the unknown covariance matrices are obtained systematically from a calibration measurement instead of going through a trial-and-error procedure.

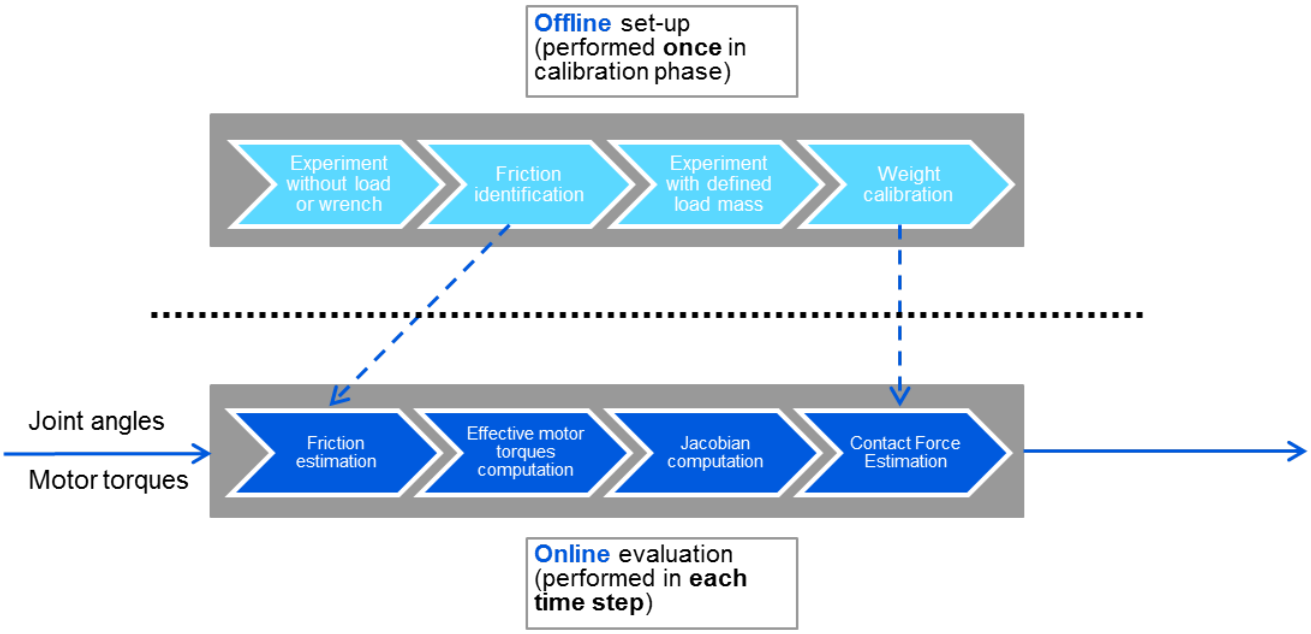


Figure 10.1: Schematic overview of the steps performed during the contact force estimation [76]

10.2 Problem Statement

We consider robotic systems that can be described by

$$\boldsymbol{H}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \boldsymbol{C}(\dot{\boldsymbol{q}},\boldsymbol{q})\dot{\boldsymbol{q}} + \boldsymbol{\tau}_{\text{grav}}(\boldsymbol{q}) + \boldsymbol{\tau}_{\text{fric}}(\dot{\boldsymbol{q}}) + \boldsymbol{\tau}_{\text{ext}} = \boldsymbol{\tau}_{\text{mot}}. \tag{10.1}$$

With the inertia matrix $\boldsymbol{H}(\boldsymbol{q})$ and the Coriolis matrix $\boldsymbol{C}(\dot{\boldsymbol{q}},\boldsymbol{q})$ the term $\boldsymbol{H}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \boldsymbol{C}(\dot{\boldsymbol{q}},\boldsymbol{q})\dot{\boldsymbol{q}}$ in Equation 10.1 captures dynamic effects, $\boldsymbol{\tau}_{\text{grav}}(\boldsymbol{q})$ are torques resulting from gravity, and $\boldsymbol{\tau}_{\text{fric}}(\dot{\boldsymbol{q}})$ describes friction torques. In addition, external forces and torques lead to reaction torques $\boldsymbol{\tau}_{\text{ext}}$ in the robot joints. The robot movement is driven by the motor torques $\boldsymbol{\tau}_{\text{mot}}$, which are assumed to be available from motor current measurements. Apart from the motor torques of each joint and the corresponding joint angles, no other measurements are necessary for the proposed contact force estimation scheme.
An approximation of $\boldsymbol{H}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \boldsymbol{C}(\dot{\boldsymbol{q}},\boldsymbol{q})\dot{\boldsymbol{q}}$ as well as $\boldsymbol{\tau}_{\text{grav}}$ is available from low-level controls.
To estimate the friction torques $\boldsymbol{\tau}_{\text{fric}}$, a method is proposed in Section 10.3.2.
The torques $\boldsymbol{\tau}_{\text{ext}}$ resulting from the external wrench are given by [62, Section 1.10]

$$\boldsymbol{\tau}_{\text{ext}}(\boldsymbol{q}) = \boldsymbol{J}^{\mathsf{T}}(\boldsymbol{q}) \cdot \boldsymbol{f}, \tag{10.2}$$

where $\boldsymbol{J}(\boldsymbol{q})$ is the robot's Jacobian and

$$\boldsymbol{f} = \begin{bmatrix} f_x & f_y & f_z & \tau_x & \tau_y & \tau_z \end{bmatrix}^{\mathsf{T}} \tag{10.3}$$

is the external wrench vector containing contact forces f_x, f_y, and f_z, as well as contact torques τ_x, τ_y, and τ_z. $\boldsymbol{f}$ is expressed in base-frame coordinates.
The objective is to estimate $\boldsymbol{f}$, i.e. the external wrench based on the measured motor torques and joint angles.

10.3 Contact Force Estimation Scheme

10.3.1 Basic Idea

The motor torques $\boldsymbol{\tau}_{\text{mot}}$ can be split into a dynamic feed-forward term $\boldsymbol{\tau}_{\text{ff}}$ to cancel $\boldsymbol{H}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \boldsymbol{C}(\dot{\boldsymbol{q}},\boldsymbol{q})\dot{\boldsymbol{q}}$, a gravity compensation term $\boldsymbol{\tau}_{\text{grav,comp}}$, a friction compensation term $\boldsymbol{\tau}_{\text{fric,comp}}$, and remaining effective torques $\overline{\boldsymbol{\tau}}_{\text{mot}}$, i.e.

$$\boldsymbol{\tau}_{\text{mot}} = \boldsymbol{\tau}_{\text{ff}} + \boldsymbol{\tau}_{\text{grav,comp}} + \boldsymbol{\tau}_{\text{fric,comp}} + \overline{\boldsymbol{\tau}}_{\text{mot}}. \tag{10.4}$$

Obviously, due to parametric uncertainties and unmodeled effects, the compensation of dynamic effects, gravity, and friction are never exact. This is summarized in the error

$$\begin{aligned} \boldsymbol{e} = & \underbrace{\boldsymbol{H}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \boldsymbol{C}(\dot{\boldsymbol{q}},\boldsymbol{q})\dot{\boldsymbol{q}} - \boldsymbol{\tau}_{\text{ff}}}_{\boldsymbol{e}_{\text{dyn}}} + \dots \\ & + \underbrace{\boldsymbol{\tau}_{\text{grav}}(\boldsymbol{q}) - \boldsymbol{\tau}_{\text{grav,comp}}}_{\boldsymbol{e}_{\text{grav}}} + \dots \\ & + \underbrace{\boldsymbol{\tau}_{\text{fric}}(\dot{\boldsymbol{q}}) - \boldsymbol{\tau}_{\text{fric,comp}}}_{\boldsymbol{e}_{\text{fric}}}. \end{aligned} \tag{10.5}$$

Notice that, while $\boldsymbol{e}_{\text{dyn}}$ and $\boldsymbol{e}_{\text{grav}}$ are small if an accurate model is available, $\boldsymbol{e}_{\text{fric}}$ comprises the major part of the error vector $\boldsymbol{e}$.

By combining Equation 10.1 with Equations 10.4 and 10.5,

$$\boldsymbol{\tau}_{\text{ext}}(\boldsymbol{q}) + \boldsymbol{e} = \boldsymbol{J}^{\mathsf{T}}(\boldsymbol{q}) \cdot \boldsymbol{f} + \boldsymbol{e} = \overline{\boldsymbol{\tau}}_{\text{mot}} \tag{10.6}$$

is obtained. The first step in estimating the external wrench $\boldsymbol{f}$ is thus to compute $\overline{\boldsymbol{\tau}}_{\text{mot}}$. Following Equation 10.4, this can readily be achieved by

$$\overline{\boldsymbol{\tau}}_{\text{mot}} = \boldsymbol{\tau}_{\text{mot}} - \boldsymbol{\tau}_{\text{ff}} - \boldsymbol{\tau}_{\text{grav,comp}} - \boldsymbol{\tau}_{\text{fric,comp}}. \tag{10.7}$$

While the dynamic feed-forward and gravity compensation torques are available from low-level robot controls and the motor torques $\boldsymbol{\tau}_{\text{mot}}$ can be measured, the friction torques $\boldsymbol{\tau}_{\text{fric}}(\dot{\boldsymbol{q}})$ have to be estimated. This is described in detail in Section 10.3.2.
The next step is to obtain an estimation of $\boldsymbol{f}$ based on Equation 10.6. This problem is non-trivial due to the fact that the disturbance torques $\boldsymbol{e}$ are unknown. Furthermore, since each arm of the ABB DACR has seven DOF, $\boldsymbol{J}(\boldsymbol{q}) \in \mathbb{R}^{6\times 7}$ is a non-square matrix. Consequently, the six components of the external wrench are mapped to the seven robot joints by Equation 10.2.
Neglecting the disturbances $\boldsymbol{e}$, a simple approach to obtain an estimate $\hat{\boldsymbol{f}}$ would be to employ the Moore-Penrose-Inverse of $\boldsymbol{J}^{\mathsf{T}}(\boldsymbol{q})$, i.e.

$$\hat{\boldsymbol{f}} = \left(\boldsymbol{J}^{\mathsf{T}}(\boldsymbol{q})\right)^{+} \cdot \overline{\boldsymbol{\tau}}_{\text{mot}}. \tag{10.8}$$

However, similar as proposed in [44, Section 10.2], [66], it is possible to include prior knowledge or assumptions on $\boldsymbol{e}$ and $\boldsymbol{f}$ in the estimation by employing a Bayesian approach. Here, both the disturbances $\boldsymbol{e}$ and the external wrench $\boldsymbol{f}$ are modeled as random variables, where $\mathbb{E}\left[\boldsymbol{e}\right] = \boldsymbol{0}$, $\mathbb{E}\left[\boldsymbol{f}\right] = \boldsymbol{0}$ is assumed. The covariance matrices are denoted by $\text{Var}\left[\boldsymbol{e}\right] = \mathbb{E}\left[\boldsymbol{e} \cdot \boldsymbol{e}^{\mathsf{T}}\right] = \boldsymbol{R}_e$ and $\text{Var}\left[\boldsymbol{f}\right] = \mathbb{E}\left[\boldsymbol{f} \cdot \boldsymbol{f}^{\mathsf{T}}\right] = \boldsymbol{R}_F$, and the variables $\boldsymbol{f}$ and $\boldsymbol{e}$ are assumed to be uncorrelated, i.e. $\mathbb{E}\left[\boldsymbol{f} \cdot \boldsymbol{e}^{\mathsf{T}}\right] = \boldsymbol{0}$, $\mathbb{E}\left[\boldsymbol{e} \cdot \boldsymbol{f}^{\mathsf{T}}\right] = \boldsymbol{0}$. How the contact force estimation can be derived is shown in detail in Appendix A.
The optimal solution for the contact force estimation based on $\overline{\boldsymbol{\tau}}_{\text{mot}}$ and the prior knowledge on $\boldsymbol{f}$ and $\boldsymbol{e}$ is given by

$$\hat{\boldsymbol{f}} = \boldsymbol{R}_F \boldsymbol{J}(\boldsymbol{q}) \cdot \left(\boldsymbol{J}^{\mathsf{T}}(\boldsymbol{q}) \cdot \boldsymbol{R}_F \cdot \boldsymbol{J}(\boldsymbol{q}) + \boldsymbol{R}_e\right)^{-1} \cdot \overline{\boldsymbol{\tau}}_{\text{mot}}. \tag{10.9}$$

In practice, the weighting matrices $\boldsymbol{R}_e$ and $\boldsymbol{R}_F$ are not known precisely. However, from Equation 10.9 it is obvious that they have a major influence on the quality of the contact force estimation. While it is possible to select the matrices based on intuitive reasoning or to adjust them in a trial-and-error procedure [66], a more systematic solution is to acquire them based on calibration measurements. This is presented in Section 10.3.3.

10.3.2 Friction Identification

As discussed above, joint friction is a major issue for contact force estimation, and a friction estimation $\boldsymbol{\tau}_{\text{fric,comp}}$ is needed to compensate for it prior to the actual contact force estimation. In this approach, a simple model capturing the effects of Coulomb and viscous friction is employed.
For each joint, the friction torque $\boldsymbol{\tau}_{\text{fric,comp},i}(\dot{q}_i)$ is modeled as

$$\boldsymbol{\tau}_{\text{fric,comp},i}(\dot{q}_i) = \begin{cases} c^{+}_{\text{Coulomb},i} + c^{+}_{\text{visc},i} \cdot \dot{q}_i, & \dot{q}_i > 0, \\ c^{-}_{\text{Coulomb},i} + c^{-}_{\text{visc},i} \cdot \dot{q}_i, & \dot{q}_i < 0. \end{cases} \tag{10.10}$$

The friction model in Equation 10.10 only captures the most basic effects and can, of course, be refined (see [7, 53] for an in depth discussion on friction models in robotics). However, such a simple friction model leads to reasonable results as, for example, shown in [20, 52].

To identify the friction coefficients c, identification measurements are performed. Therein, the manipulator is moved around freely in the workspace, which leads to $\boldsymbol{\tau}_{\text{ext}} = \mathbf{0}$. Based on this, the robot dynamics in Equation 10.1 can be reformulated as

$$\boldsymbol{\tau}_{\text{fric}}(\dot{\boldsymbol{q}}) = \boldsymbol{\tau}_{\text{mot}} - \left(\boldsymbol{H}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \boldsymbol{C}(\dot{\boldsymbol{q}}, \boldsymbol{q})\dot{\boldsymbol{q}} + \boldsymbol{\tau}_{\text{grav}}(\boldsymbol{q})\right). \tag{10.11}$$

By similar reasoning, as presented in Section 10.3.1, the friction torques can be estimated from motor torques and dynamic feed-forward and gravity compensation torques as

$$\boldsymbol{\tau}_{\text{fric,comp}}(\dot{\boldsymbol{q}}) = \boldsymbol{\tau}_{\text{mot}} - \left(\boldsymbol{\tau}_{\text{ff}} + \boldsymbol{\tau}_{\text{grav,comp}}\right) = \tilde{\boldsymbol{\tau}}_{\text{mot}}. \tag{10.12}$$

The joint speeds $\dot{\boldsymbol{q}}$ are assumed to be known since they can either be measured or calculated by numerical differentiation from the joint angles.
Based on the above considerations, an identification of the friction coefficients can be obtained by solving the optimization problems

$$\underset{c^{+}_{\text{Coulomb},i}, c^{+}_{\text{visc},i}}{\text{minimize}} \sum_{k=1}^{N} \left(\tau^{+k}_{\text{fric,comp},i}(\dot{q}_i^k) - \tilde{\tau}^{+k}_{\text{mot},i}\right)^2, \ \forall i = 1, \ldots, 7, \tag{10.13a}$$

$$\underset{c^{-}_{\text{Coulomb},i}, c^{-}_{\text{visc},i}}{\text{minimize}} \sum_{k=1}^{N} \left(\tau^{-k}_{\text{fric,comp},i}(\dot{q}_i^k) - \tilde{\tau}^{-k}_{\text{mot},i}\right)^2, \ \forall i = 1, \ldots, 7. \tag{10.13b}$$

Here, $\tau^{+k}_{\text{fric,comp},i}(\dot{q}_i^k)$ and $\tilde{\tau}^{+k}_{\text{mot},i}$ are samples of the friction and motor torques for the i-th joint with positive joint speed $\dot{q}_i$, while $\tau^{-k}_{\text{fric,comp},i}(\dot{q}_i^k)$ and $\tilde{\tau}^{-k}_{\text{mot},i}$ collect all samples with negative joint speeds.
The problems given in Equation 10.13 are standard least squares problems.
For each joint the Equation 10.13a is solved by

$$\begin{bmatrix} c^{+}_{\text{Coulomb},i} \\ c^{+}_{\text{visc},i} \end{bmatrix} = \begin{bmatrix} 1 & \dot{q}_i^1 \\ \vdots & \vdots \\ 1 & \dot{q}_i^N \end{bmatrix}^{+} \cdot \begin{bmatrix} \tilde{\tau}^{+1}_{\text{mot},i} \\ \vdots \\ \tilde{\tau}^{+N}_{\text{mot},i} \end{bmatrix}, \tag{10.14}$$

with N samples with positive joint speeds available. Equation 10.13b can be treated in the same way.
Figure 10.2 shows a result of a friction identification measurement for robot joints 3 and 5.
As Figure 10.2 demonstrates, the simple friction model in Equation 10.10 allows an estimation of the friction torques with reasonable accuracy.

10.3.3 Calibration of the Weighting Matrices

Calibration measurements with a defined external wrench $\boldsymbol{f}_{\text{def}}$ are performed to tune the weighting matrices $\boldsymbol{R}_e$ and $\boldsymbol{R}_F$ automatically.
To do so, a defined load mass m_{calib} is attached at the TCP, resulting in

$$\boldsymbol{f}_{\text{def}} = \begin{bmatrix} 0 & 0 & -m_{\text{calib}} \cdot g & 0 & 0 & 0 \end{bmatrix}^{\mathsf{T}}, \tag{10.15}$$

due to the formulation of the external wrench in base frame coordinates. With the defined wrench applied, calibration measurements are performed. These can be any contact-free movement of the TCP in the workspace.

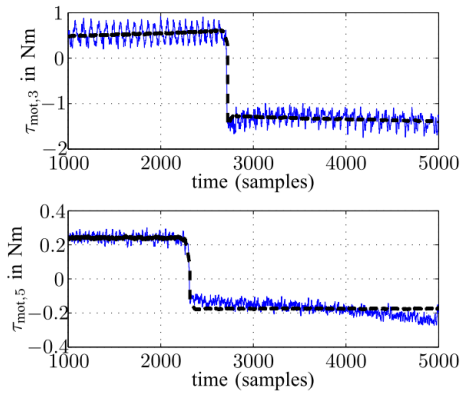


Figure 10.2: Friction identification results (—: measured motor torques with dynamic feed-forward and gravity compensation, — —: estimated friction torques) [79]

Since dynamic feed-forward torques are used to compute $\overline{\tau}_{\text{mot}}$, the motion trajectory of the TCP should not result in any jerk during the calibration measurement. This avoids large values for $\ddot{\boldsymbol{q}}$ and thus undesired peaks in $\overline{\tau}_{\text{mot}}$ are avoided.
Having performed the calibration measurement, the weighting matrices for contact force estimation following Equation 10.9 are determined in an offline optimization. The objective is to find $\boldsymbol{R}_e$ and $\boldsymbol{R}_F$, so that the estimated wrench $\hat{\boldsymbol{f}}$ matches $\boldsymbol{f}_{\text{def}}$ best in the sense of mean-squared errors over the whole measurement. This can be written as

$$\underset{\boldsymbol{R}_e,\boldsymbol{R}_F}{\text{minimize}} \sum_{k=1}^{N} \left\| \hat{\boldsymbol{f}}^k - \boldsymbol{f}_{\text{def}} \right\|_2^2 \text{ s.t.} \tag{10.16a}$$

$$\hat{\boldsymbol{f}}^k = \boldsymbol{R}_F \boldsymbol{J}(\boldsymbol{q}^k) \cdot \left(\boldsymbol{J}^{\mathsf{T}}(\boldsymbol{q}^k) \cdot \boldsymbol{R}_F \cdot \boldsymbol{J}(\boldsymbol{q}^k) + \boldsymbol{R}_e \right)^{-1} \cdot \overline{\boldsymbol{\tau}}^k_{\text{mot}}, \tag{10.16b}$$

$$\overline{\boldsymbol{\tau}}^k_{\text{mot}} = \boldsymbol{\tau}^k_{\text{mot}} - \boldsymbol{\tau}^k_{\text{ff}} - \boldsymbol{\tau}^k_{\text{grav,comp}} - \boldsymbol{\tau}^k_{\text{fric,comp}}, \tag{10.16c}$$

$$\boldsymbol{\tau}^k_{\text{fric,comp},i}(\dot{q}^k_i) = \begin{cases} c^+_{\text{Coulomb},i} + c^+_{\text{visc},i} \cdot \dot{q}^k_i, & \dot{q}^k_i > 0, \\ c^-_{\text{Coulomb},i} + c^-_{\text{visc},i} \cdot \dot{q}^k_i, & \dot{q}^k_i < 0. \end{cases} \tag{10.16d}$$

Due to the stochastic interpretation of the weighting matrices, additional contraints $\boldsymbol{R}_e \succeq \boldsymbol{0}$, $\boldsymbol{R}_F \succeq \boldsymbol{0}$ have to be taken into account. Furthermore, $\boldsymbol{R}_e$ and $\boldsymbol{R}_F$ are constrained to be diagonal matrices as $\boldsymbol{e}$ and $\boldsymbol{f}$ are mutually uncorrelated.
Figure 10.3 shows results of a calibration following Equation B.4. A load mass with $m_{\text{calib}} = 0.165g$ has been applied to the TCP with the resulting external forces ($f_x = 0, f_y = 0, f_z = 1.65N$). The estimated external forces are obtained employing the friction identification (see Section 10.3.2) and the Equation 10.9 with the calibrated weighting matrices and match the actual forces reasonably.

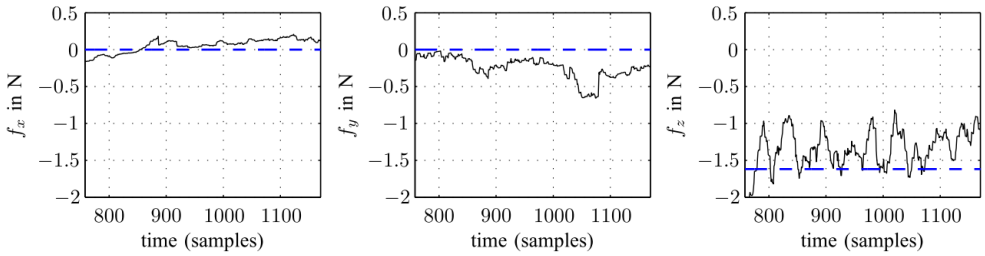


Figure 10.3: Calibration results (— —: actual forces due to attached calibration load, —: estimated external forces) [79]

10.4 Results

Summarizing the previous section, the following steps have to be completed to properly calibrate the method:

1. Run an experiment without external wrench
2. Perform the friction estimation based on measurement data and Equation 10.14
3. Run an experiment with calibration load attached at TCP
4. Perform the calibration of $\boldsymbol{R}_e$ and $\boldsymbol{R}_F$ by solving Equation 10.16 based on measurement data

While the computational effort in Step 4 is considerable, it only has to be performed once and offline in order to tune the matrices $\boldsymbol{R}_e$ and $\boldsymbol{R}_F$. In the actual application of the estimation scheme, the matrices are constant values. The operations that have to be executed online are:

1. The friction estimation $\boldsymbol{\tau}_{\text{fric,comp}}$ according to Equation 10.10, based on the constant identified friction coefficients and computed joint speeds
2. The computation of $\overline{\boldsymbol{\tau}}_{\text{mot}}$, based on the measured torques, dynamic feed-forward torques $\boldsymbol{\tau}_{\text{ff}}$, gravity compensation torques $\boldsymbol{\tau}_{\text{grav,comp}}$, and estimated friction torques $\boldsymbol{\tau}_{\text{fric,comp}}$
3. The computation of the robot's Jacobian $\boldsymbol{J}(\boldsymbol{q})$ for the current pose
4. The evaluation of the contact force estimation equation (Equation 10.9) using the calibrated matrices $\boldsymbol{R}_e$ and $\boldsymbol{R}_F$

Except for motor torques and joint angles, no other signals have to be measured.

The proposed scheme has been applied to the ABB DACR to estimate external forces arising from a contact of the gripper with the ground. In Figure 10.4 the contact forces f_x, f_y, and f_y are plotted over the z-coordinate of the TCP. In the experiment, the z-position decreases, i.e. the plots in Figure 10.4 have to be read from right to left. For $z > 0$ the gripper is moving freely and consequently, all contact forces are approximately 0. At $z = 0$, contact to the ground is established as can clearly be detected by the sharp peak in f_z. Since the gripper is moving also in x and y-direction at the contact point, there are force components in these directions due to friction between gripper and ground.

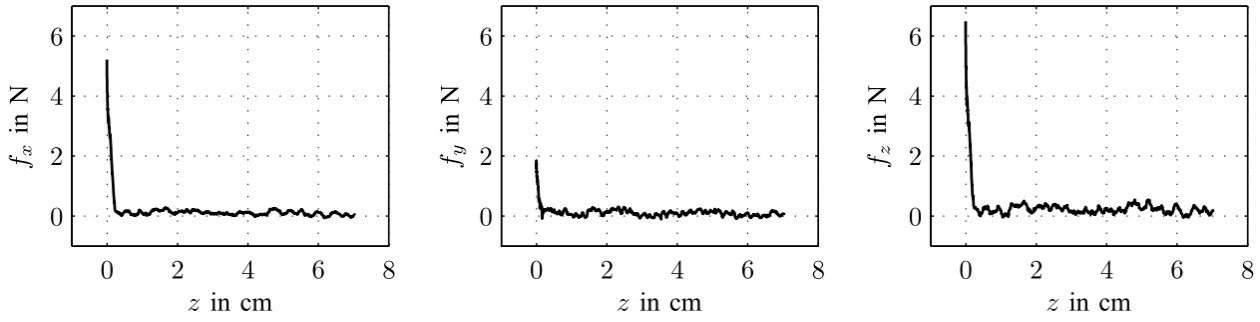


Figure 10.4: Estimated contact forces f_x, f_y, and f_z over z-position of TCP [79]

11 Implementation

In this chapter, the implementation of the concept proposed in Part II is presented. A demonstrator that provides the necessary functionality to perform an example assembly by using manipulation skill on an ABB dual-arm concept robot (see Chapter 9) was the result of the implementation.

11.1 Overall Program Structure

The overall structure of the program can be visualized by the following data flow diagram.

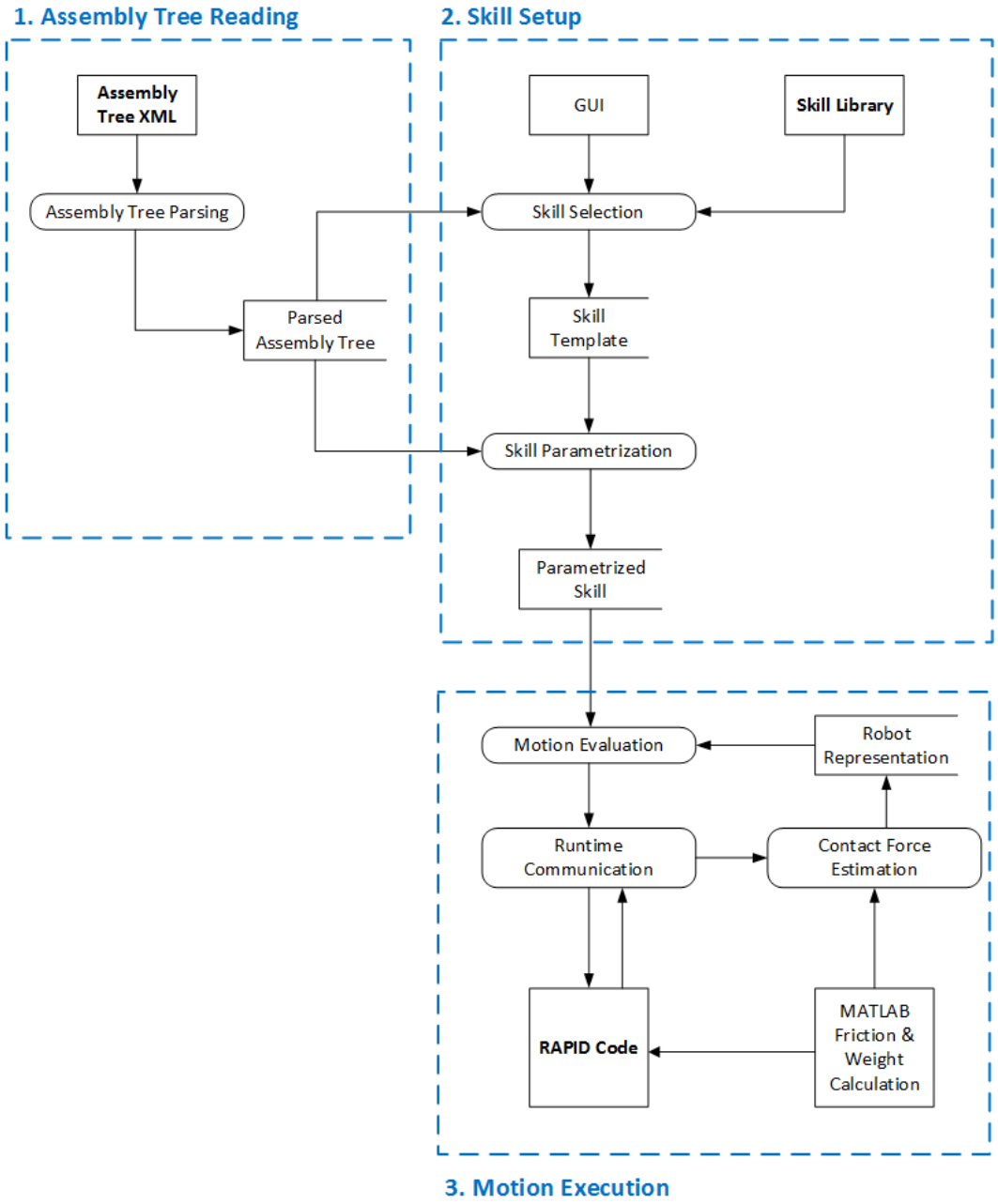


Figure 11.1: Overview of the overall program workflow, split into the execution steps 1, 2 and 3

The program provides the functionality to create a sequence of robot motions based on skill primitives and control the execution of the motions. To create a motion output, an assembly tree data structure and a library containing skill primitives is necessary as input. Three consecutive execution steps have to be performed by the program before actual robot motion is generated (see Figure 11.1):

1. **Assembly Tree Reading** The assembly tree data structure is traversed. Its content is parsed and stored in an internal representation that is used for the selection and parametrization of skill primitives.

2. **Skill Setup** Skill primitives are selected from a library containing several predefined skill templates. The selection is based on the data specified in the assembly tree. An additional manual selection based on a GUI can be performed. After a set of skill templates has been selected from the library. This set is parametrized based on the information from the assembly tree. Additional manual GUI-based parameter input is possible.

3. **Motion Execution** The motions contained in the skill primitives are executed. After they have been sent to the robot controller via a communication interface, a robot control program carries out the robot motions. During the execution, the actual motion state of the robot is continuously compared to the desired motion state defined in the skill primitives. The current state is acquired by processing data received from the robot control program via the communication interface. This data processing generates the current pose-wrench state (see Section 7.2.1) of the robot through contact force estimation and stores it in an internal robot representation.

Steps 1 and 2 are executed "offline", which means in advance to the execution of the robot motion. From a practical viewpoint, the demonstrator consists of the following main program parts to carry out the execution steps:

- A main application written in C# for the representation, coordination and execution of skill primitives
- A robot control program written in the ABB RAPID programming language to carry out robot commands provided by the main application and to acquire the current motion state of the robot
- An XML data structure representing the assembly tree, which is used as a data source to select and parametrize skills

In the following sections, the program parts and execution steps are explained in more detail.

11.2 Assembly Tree Reading

In this section, the XML representation of the assembly tree is presented. Furthermore, it is described how to convert this representation to an internal data structure in the main application.

11.2.1 Assembly Tree Representation

An XML tree data structure implements the assembly tree presented in Section 6.3. This node-based data structure consists of two components: an XSD scheme defining the structure and data types of the nodes of the assembly tree XML files, and the XML files themselves, which are derived from it. Based on the definitions in Section 6.3.2, the following main data types are specified in the XSD scheme:

Part A *Part* node contains descriptive and procedural information about an individual component used in an assembly. Descriptive information is represented by a transformation stored as a 4x4 matrix, which describes the part's goal assembly position with respect to its parent node. Furthermore, a link to a CAD representation and a picture can be stored. The procedural information describes how the part has to be handled during assembly and is represented as a sequence of *Actions*. As part nodes are the leaf nodes in the tree structure, they do not have any child nodes.

Assembly An *Assembly* node is a branch node in the tree structure and contains up to two child nodes, the left and the right child, which can be *Part* or *Assembly* nodes. They represent the components assembled in this assembly step. If such a child node contains assembly instructions in the form of *Action* nodes, their data is used for the creation of an *AssemblyTreeTask* (see Section 6.3.3). The node itself contains the same information as the part node, except the for CAD representation. A resource list stored in the node contains information on which resources are necessary to perform these assembly processes.

Action An *Action* node contains all necessary information to select and parametrize skill primitives. The *Activity* indicator plays the crucial role in skill selection as it describes the kind of action represented by a node. Process parameters like *Speed* and path information stored as *Targets* are used for the parametrization of skill primitives. Up to three targets can be stored in an *Action* node. As the system is implemented using a dual-arm robot, an indicator, which robot arm (mechanical unit) has to be used to carry out the action, is necessary. This is provided by the node's *MecUnit* descriptor. It is possible to store an interruption condition specified as a 12-dimensional vector that represents the 12 dimensions of pose-wrench space. Furthermore, a task frame can be stored as 4x4 matrix, which is used as a reference frame for the skill that executes the action.

Target A *Target* can be represented as a 12-dimensional pose-wrench vector. Furthermore, it can be specified as a *RobTarget*, which is the target data type used in ABB RAPID, or as a set of translational and rotational offset values. Tolerance values can be stored in the node for each of the 12 pose-wrench dimensions.

Beside these main types, which are defined as *ComplexTypes* in the *XML Schema* programming language, different utility types are defined. Utility types that define numerical arrays of different length are used to represent vectors and matrices. Enumeration types are used to enlist different values, for example tools or sensors. The latter two types are specified as *SimpleTypes* in *XML Schema*. Furthermore, predefined types like "Boolean" are used.

11.2.2 Assembly Tree Traversal

By traversing the assembly tree, a sequence of tasks is generated from the XML representation. This sequence is employed in the selection and parametrization of skill primitives. The traversal of the assembly tree XML and the parsing of its data to an internal data structure is carried out by the class *AssemblyTreeReader*, which is inherited from the class *ReadXML*. The latter class contains the basic functionality to parse XML data, while the former contains specific functions to read the assembly tree XML structure. To read the assembly tree, the tree's root node, which is always of type *Assembly*, is processed first. Aftwards, its child nodes are processed successively. The function *createAssemblyTreeTask()* is the central function in this processing, as it creates internal data representations from the data stored in the *Action* nodes of a part or assembly. The data generated by the function is stored internally in instances of the classes *AssemblyTreeTask* and *AssemblyAction*. The function only creates an *AssemblyTreeTask* if *Action* nodes are contained in the processed node. Each task that is generated is stored in a *stack* data structure. To generate the output task sequence, the task stack order has to be reverted after all nodes have been processed. The tree traversal process is visualized by the following flowchart:

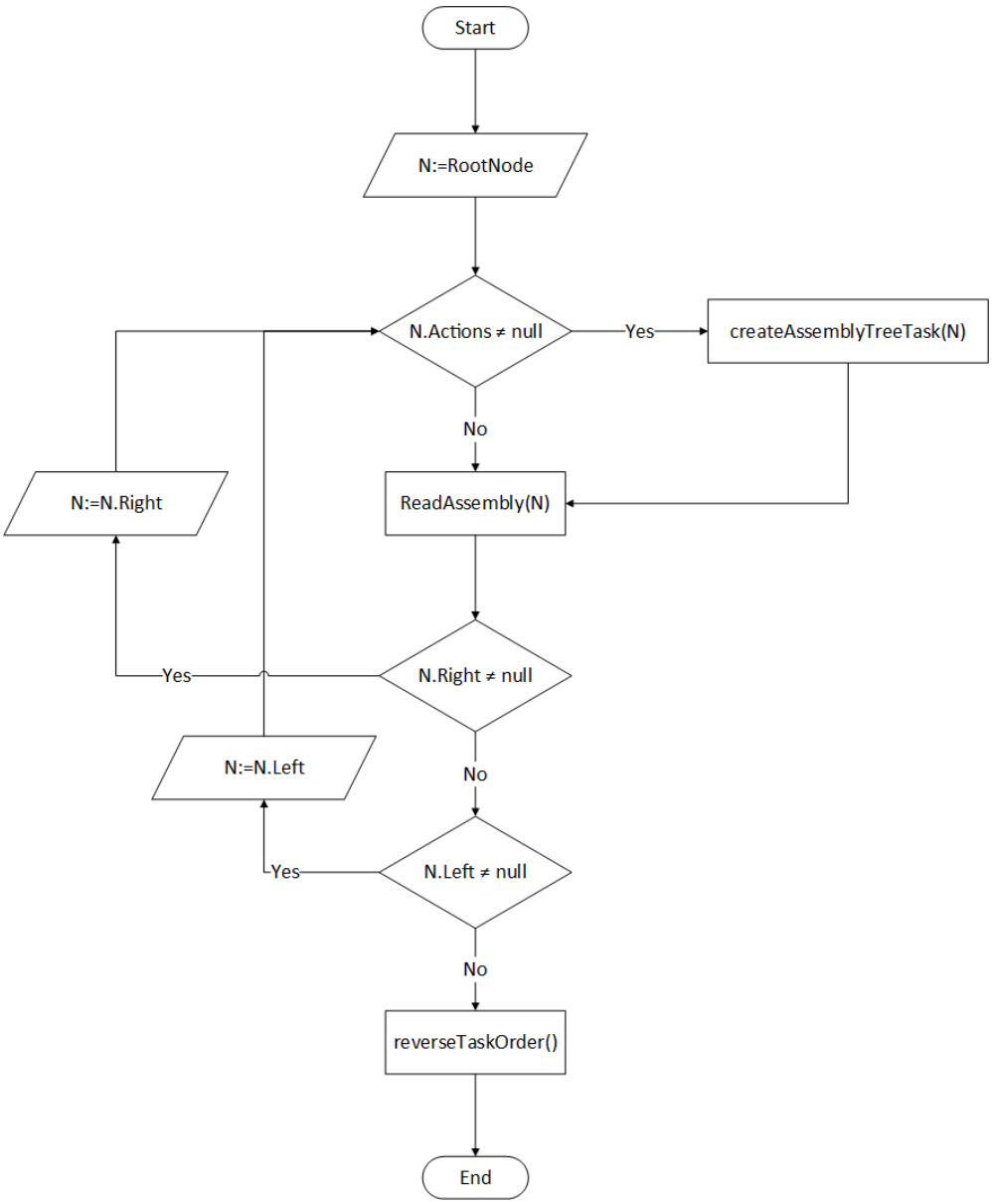


Figure 11.2: Assembly Tree Traversal Flowchart

11.3 Skill Setup

After the assembly tree has been converted from XML to an internal representation, skill primitives have to be selected and parametrized. How this process, which represents the concepts presented in Chapter 7, is implemented, is described in the following.

11.3.1 Template Skill Representation

A template skill is represented by the class *TemplateSkill* which is inherited from a general *Skill* class. The *Skill* class is the superclass for the *TemplateSkill* and *ParametrizedSkill* classes. It contains the following metadata used by both of these classes:

- A list of *Activities* the skill is suited for
- A list of *Sensors* required for the execution of the skill
- A list of *Tools* required for the execution of the skill

The *TemplateSkill* class contains the following data that refers to the motion net and trajectory template concepts presented in Chapters 7.2.5 and 7.3.3.

Trajectory Template

Trajectory templates as presented in Chapter 7.2.5 are implemented by the *TrajectoryPattern* class. This class contains a list of objects of type *TPNodeTemplate*. Each of these objects refers to a trajectory template node $\boldsymbol{N}_i'$. Their only content is a list of 12 motion types described by the enumeration type *TPNodeMotionType*. The enumeration contains the values {unsupervised, constant, finiteChange, infiniteChange, zero} as described in Section 7.2.5.

Motion Net Template

The motion net template represents the "skeleton" of a motion net as described in Chapter 7.3.3. It is implemented by the class *MotionNetTemplate* that contains a list of *MotionNetTemplateState* objects and a list of *MotionNetTemplateTrigger* objects representing the states and transitions in the template motion net. A *MotionNetTemplateState* contains a state type, which can have the values {motion, toolAction, initial, final, complete, success, failure}, and the type of the robot command it represents (if a command is represented). The values {linearMotion, offsetMotion, openGripper, closeGripper, toolOffsetMotion, collisionOffsetMotion, homeMotion, idleState}, which represent all commands implemented in the system, are valid as robot command type. Both types are stored as enumeration types. A *MotionNetTemplateTrigger* contains its initial and final *MotionNetTemplateState*s.

Trajectory Mapping

The trajectory mapping implements the mapping rule set $\boldsymbol{R1}$ described in Chapter 7.2.5, which is used to parametrize a template trajectory according to information from an assembly tree task. The mapping is implemented by a key-value pair list ("Dictionary" in C#) linking a template node of type *TPNodeTemplate* to a mapping rule of type *TrajectoryNodeMappingRule* and an *AssemblyAction* from which the parameters are derived.

Mapping rules are implemented as C# delegates. The trajectory node, represented by the node type *TPNode*, which is parametrized during the mapping, and the *AssemblyAction*, from which the mapping parameters are derived, are passed to the delegate as arguments. A mapping rule specifies which target of an *AssemblyAction* is used to parametrize a *TPNode*. How the mapping works in detail is described in Section 11.3.4.

Motion Net Mapping

The trajectory mapping implements the mapping rule set $\boldsymbol{R2}$ described in Chapter 7.2.5, which is used to parametrize a motion net according to information from a parametrized trajectory. Two different motion net mapping rule sets are stored in a *TemplateSkill* object: one for the parametrization of *MotionNetTemplateState*s referring to the set $\boldsymbol{R2.1}$ and one for the parametrization of *MotionNetTemplateTrigger*s referring to the set $\boldsymbol{R2.2}$. Both are implemented as a key-value pair list linking a template state or trigger to a mapping rule of type *MotionNetStateMappingRule* or *MotionNetTriggerMappingRule*. Furthermore, the *TPNodeTemplate* of the trajectory node used for the parametrization is stored in each list entry.

*MotionNetStateMappingRule*s are implemented as C# delegates that use the *MotionNetState* to be parametrized and the *TPNode* from which the parameters are derived as arguments. The rules define which *MotionState* of a *TPNode* is used to set the control values of a *MotionNetState*. A *MotionState* is the implementation of a 12-dimensional pose-wrench state (see Section 11.3.2).

*MotionNetTriggerMappingRule*s are implemented as C# delegates that use the *MotionNetTrigger* to be parametrized and the *TPNode* from which the parameters are derived as arguments. The rules define which entries of which *MotionState* of a *TPNode* are used to create the transition condition of the motion net trigger. Furthermore, the condition type is set during the mapping, which refers to setting the operator (>,<,=,...) of the transition condition's comparison function as described in Section 7.3.1.

Beside the described data, an instance of *TemplateSkill* can store a list of parametrizations of the represented skill template. *TemplateSkill* instances themselves can be stored in the *SkillLibrary* class. To define a *TemplateSkill* in the library, the data described above has to be set, including the metadata in the *Skill* class.

11.3.2 Parametrized Skill Representation

A parametrized skill is represented by the class *ParametrizedSkill*, which is inherited from a general *Skill* class. The *ParametrizedSkill* class contains the following data that refers to the trajectory and motion net concepts presented in Chapters 7.2 and 7.3.

Trajectory

The pose-wrench trajectory is represented by the *Trajectory* class. In this class, a sequence of trajectory nodes, each represented by the class *TPNode*, is stored.

A *TPNode* refers to a trajectory node $\boldsymbol{N_i}$ as described in Section 7.2.2. Each *TPNode* is defined by an initial and a final pose-wrench trajectory state, both of which are represented by the *MotionState* class.

The *MotionState* class refers to trajectory state $\boldsymbol{S_i}$ as defined in Section 7.2.2 and represents a state in pose-wrench space as 12-dimensional vector. A second 12-dimensional vector is included to specify a tolerance value for each dimension. Furthermore, a Boolean array with 12 entries is specified that defines the relevant subset for the evaluation of the *MotionState*. A *MotionState* contains several functions to compare it to other *MotionState*s.

An interpolation rule for 12-dimensional vectors defined by the *Interpolation12D* class is included to retrieve states between the initial and final *MotionState*. To use the interpolation with a given motion state as input, an index has to specified that defines which of the 12 dimensions of the given state is used as an interpolation point. This index is stored as an integer in the *TPNode*. It is calculated as the index of the absolute maximum difference of the initial and final *MotionState*s of the *TPNode*.

An additional *MotionState* can be specified in a node to define a condition for an interruption that can occur during the motion represented by the *TPNode*.

It has to be noted that for the storage of a trajectory in a *ParametrizedSkill*, the trajectory is split into its *TPNode*s which are stored individually in the motion net states they refer to represented by the class *MotionNetState*.

Motion Net

The motion net described here implements the elements defined in Section 7.3.1. Its functionality is encapsulated in the *MotionNetFSM* class. This class implements a Finite State Machine (FSM) by using the *Stateless*[1] library. *Stateless* is a lightweight C# library that allows the definition of FSMs in C# using a simple syntax. The simplicity and the good performance of this library were the main reason to choose it among the wide variety of FSM libraries. Like any FSM, it models a behavior as transitions between a finite set of states. Custom data types can be used to define states and transitions. In the motion net specified here, states are represented by the *MotionNetState* class and transitions are represented by the *MotionNetTrigger* class.

A *MotionNetState* contains the same data represented in a *MotionNetTemplateState*, which is a state type and a robot command type, to describe the kind of motion the state represents. Furthermore, *ControlValues* are included to control the motion represented by the state. A part of the skill's trajectory in the form of a *TPNode* is included to supervise the motion. *ControlValues* contain all necessary information for the controller to carry out a robot motion. The main part of this information are six control values, each referring to a spatial dimension of a reference frame. Each control value is specified as a tuple consisting of a numeric value and a control mode. The control mode is defined as an enumeration type and can, for example, be "force" or "position" (even though the currently employed controller only has position control available). This definition is consistent with the Task Frame Formalism (see Section 3.2.1). Furthermore, the reference frame the control values refer to is stored as a matrix. A *MechanicalUnit* descriptor specifies which arm of the dual-arm robot performs the motion.

Beside the data from the *MotionNetTemplateTrigger*, a *MotionNetTrigger* contains a list of *TransitionCondition*s that have to be met before a transition fires. *TransitionCondition*s consist of a condition represented as a *MotionState*, and a condition type defined by an enumeration type. The condition type refers to the comparison operator $\square \in \{\leq, \geq, =, <, >\}$ as defined in Section 7.3.1.

To be available for execution, parametrized skills are themselves stored in an FSM called the "Skill Net". The structure of this FSM is much simpler than the structure of the motion net, as it contains the skills in a strictly sequential order. For the implementation, the *Stateless* library was employed as well. The net of skills is represented in the *SkillNet* class. The states in the skill net represented by the class *SkillNetState* each contain a parametrized skill. The transitions specified by the class *SkillNetTrigger* do not contain any conditions but are fired if the final motion state of the skill represented by the transition's initial state is reached.

11.3.3 Skill Selection

The functionality to select skill templates from the skill library is located in the class *SkillSelection*. During the selection process, a list of skill templates for each task is created. Necessary inputs for the skill selection are the library containing the skill templates and the list of assembly tasks created during the assembly tree reading process. To generate a list of skill templates, the *selectSkills()* function is called for each task in the input task list. In this function, three consecutive steps are performed, which are described in the following.

Step 1 - Search Skills by Activity For each *AssemblyAction* contained in an *AssemblyTreeTask* given as input, all *TemplateSkill*s with fitting activities are selected. If more than one activity is specified in a *TemplateSkill*, it has to be checked if the succeeding *AssemblyAction*s of the currently processed task also fit to the skill's activities. If the actions fit, the skill can be used for a sequence of *AssemblyAction*s instead of a single one. The result of this step is a list of suitable *TemplateSkill*s for each *AssemblyAction* of an *AssemblyTreeTask*.

Step 2 - Check Robot Capabilities In this step the list resulting from the previous step is filtered with respect to the capabilities of the currently employed robotic system. For each skill in the list of suitable *TemplateSkill*s it is checked if the necessary tools and sensors defined in the *TemplateSkill* match the tools and sensors of the current robotic system. If the values do not match, the *TemplateSkill* is deleted from the list. The tools and sensors of the current system are stored in the *RobotRepresentation* class.

[1] http://code.google.com/p/stateless/

Step 3 - User Input In the last step of this process, a *TemplateSkill* has to be selected from the list of suitable *TemplateSkill*s generated in Steps 1 and 2. If only one skill is contained in the list for a certain *AssemblyAction*, this selection can be skipped. If more than one skill is contained, the selection is performed manually by the user. For this purpose, a GUI is employed that allows the selection of a *TemplateSkill* for each *AssemblyAction* from a drop-down menu.

The result of these steps is a list of *TemplateSkill*s linked to one or many *AssemblyAction*s. This list is used in the *Skill Parametrization* process to parametrize the *TemplateSkill*s according to the *AssemblyAction*s.

11.3.4 Skill Parametrization

From the list of *TemplateSkills* acquired by the skill selection, a list of *ParametrizedSkill*s is generated. For each *TemplateSkill* in the list, a new instance of the *ParametrizedSkill* is created with data from the *AssemblyAction* the template is linked to. During this parametrization, first the trajectory is created and afterwards the motion net is created based on the trajectory. These two basic steps are described in the following.

Trajectory Creation

To create the trajectory, a trajectory node represented by the *TPNode* class is created for each *TPNodeTemplate* stored in the *TrajectoryPattern* of the *TemplateSkill*. For the creation of a *TPNode*, the mapping described in 11.3.1 is employed. To do so, the delegate representing a *TrajectoryNodeMappingRule* is invoked using the *AssemblyAction* linked to the *TemplateSkill* as argument. Each mapping rule specifies, which target of the action is mapped to which *MotionState* of a *TPNode*.

Motion Net Creation

To create the motion net, *MotionNetState*s and *MotionNetTrigger*s are created from the *MotionNetTemplateState*s and *MotionNetTemplateTrigger*s stored in the *MotionNetTemplate* of the *TemplateSkill*. The *MotionNetStateMappingRule*s and *MotionNetStateMappingRule*s described in Section 11.3.1 are invoked for each state and trigger using the *TPNode* that is stored in the template with the mapping rule and the template state or trigger.

The result of the above described process is a list of *ParametrizedSkill*s. Before the parametrization is finished, one additional step is necessary: As the final position of a motion represented by a skill in the list might not coincide with the initial position of the motion represented by the skill's successor, transfer motions have to be created.
To create these motions, a transfer *AssemblyAction* is created. The final position of the first skill and the initial position of the second skill are used as *Target*s in this action. This action is then used to parametrize the "Transfer" *TemplateSkill* from the skill library. The newly created transfer skill is inserted at a position in the *ParametrizedSkill* list in between the two skills involved in the transfer.
A modification of the process described above is the manual parametrization of a *TemplateSkill*. If a skill is parametrized manually, the source data stored in the *AssemblyAction* used to parametrize the skill is altered by user input. This approach was used to keep the changes in the process at a minimum. An assembly tree is still necessary to select skills. To trigger the manual parametrization of the skill, the according *Action* node in the assembly tree XML is tagged as "manual". If this tag is set, the parameters defined in the respective action are overwritten in the internal representation of the *AssemblyAction*. The manual parameters that are used are acquired from an input mask stored with the *TemplateSkill*. These input masks are, for example, capable of acquiring targets from the current robot arm position or via written input. This way, a manual teaching process can be employed to parametrize a skill. It is also possible to set a subset of parameters manually while acquiring the rest of the data from the assembly tree.

After the list of *ParametrizedSkill*s is created, it is converted to a *SkillNet* FSM described in Section 11.3.2. This can easily be done by creating a *SkillNetState*s for each skill and a *SkillNetTransition* leading from the current state to the next state. The initial and final state in the skill net are empty dummy states.

11.4 Motion Execution

This section describes the program execution during the actual robot motion execution, i.e. the execution of the RAPID program on the robot controller. The next sub-section describes the execution of the rapid program itself, while the following sub-sections describe the workflow in the PC application. The latter is mainly based on events and is visualized in Figure 11.4.4.

11.4.1 RAPID Robot Control Code

To control the ABB dual-arm concept robot described in Section 9.1, a program written in the ABB RAPID programming language is stored on the IRC5 robot controller.
In RAPID, a program is arranged in several *tasks* that represent sub-programs that are executed in parallel on the controller in different threads. Tasks can represent robot motions or different other kinds of operations (e.g. communication or output operations). While tasks are executed independently of each other by default, they can be synchronized by adding "sync" commands to their code. Each task can consist of several *modules* containing the source code of the robot application as *procedures* and *variables*.
The robot program implemented here has two main purposes: carrying out robot commands generated by the main application and providing input data for the calculation of the current motion state of the robot by the contact force estimation module. Additionally, a collision detection is implemented to prevent damage to the robot during the motion execution. The functionality is distributed over four RAPID tasks, which are described in the following:

Signal Acquisition Task

The main purpose of the signal acquisition task is the reading of the joint angles and joint torques of the robot arms. The read values are sent to the PC application via the communication interface (see Section 11.4.2) and are used as input data for the contact force estimation. After the task's initialization, a loop starts which continuously performs the following steps:

Step 1 - Read Joint Angles The current joint angles of both robot arms can directly be acquired in RAPID by using the *CJointT()* command. In addition to the current angles, the previously read angles and the time passed since the last reading are stored for numerical speed calculation.

Step 2 - Read Joint Speeds As the joint torques have to be compensated for friction to be used in the contact force estimation (see Section 10.3.1), the angular joint speeds are necessary to calculate the viscous friction torques. They are acquired by numerical differentiation from the joint angles. To do so, the difference between the current angles and the previous angles is calculated and divided by the stored time difference. Optionally, a moving average filter can be employed for a smoother speed signal.

Step 3 - Read Joint Torques As described in Section 10.3.1, the joint torques derived from external forces acting on the robot arm are necessary for contact force estimation. These torque values can be acquired from the joint servos by employing the *TestSignal* interface in RAPID. This interface allows the definition and reading of several signals for each robot joint. To get the torques referring to the external forces, the difference between the signal representing the total motor torque and the signal representing the dynamic and gravity effects is calculated for each joint. Both of these signals are available from the *TestSignal* interface. The calculated torques are compensated for friction by calculating the friction torques as described in Section 10.3.2. To do so, the read joint speeds and predefined friction coefficients are used. The friction coefficients are acquired by calibration measurements (see Section 10.3.2). A drawback of the *TestSignal* interface is that it can only handle 12 signals at a time. As four signals are necessary for the collision detection, only eight signals remain to be used for contact force estimation. Consequently, the contact force estimation can only be performed using torques from four joints per arm (as two signals are necessary per joint). The joints with the lowest absolute joint values were omitted in the calculation.

Step 4 - Send Values to PC To send the acquired values to the PC application, a *Remote Message Queue* is used. This queue is created on the robot controller. At the end of each cycle, the read values are stored in the queue as *Remote Messages*. The queue can be accessed from the PC application as described in Section 11.4.2.

The loop takes about 15 ms for one execution cycle of these four steps.

Motion Tasks

The robot motions are controlled by two separate tasks: *T_ROB1* and *T_ROB2*. These two tasks have an identical structure and are each responsible for the execution of the motions of one of the robot's arms.
Several variables are defined in a motion task that represent necessary input data for a motion to be executed. These variables are set externally from the PC application via the communication interface (see Section 11.4.2). The following variables are used for the motion execution:

- The *motionType* describes the type of motion command that has to be executed.
- The *currentGoalPosition* describes the motion goal of a linear motion command.
- The *currentOffset* describes the motion goal of an offset motion command.
- The *currentWobjdata* describes the reference frame for the motion command (frames are defined as "Workobjects" in RAPID). The default value for this variable is the world coordinate system of the workspace.
- The *speedLevel* defines the speed to be used in the motion command (the speed is selected from several predefined speed values according to the level).
- The *collisionTorque* defines the collision threshold value that is used for collision detection during the motion command.

The key feature of a motion task is a *TRAP* function, which reacts to new motion input data from the PC application. A *TRAP* function is the RAPID equivalent to an event listener function. The *TRAP serverDataReceived* specified here is called when new data from the PC application arrives, which is indicated by a Boolean variable. The function's main purpose is the selection of a motion routine according to the specified *motionType* which is then executed by using the other input variables.
The motion routines represent the most common robot actions that can be executed in RAPID in a generic form. In the following, the implemented routines are described:

idleState: No motion is executed and the robot awaits the arrival of new server data. This is the default motion routine the motion task enters upon initialization.

linearMotion: The robot arm moves to a specified position on a straight line.

homeMotion: The robot arm moves to its home position.

openGripper: The gripper employed on the robot arm opens.

closeGripper: The gripper employed on the robot arm closes.

toolOffsetMotion: The robot arm moves to a position specified by a linear and angular offset to the current position. The offset is defined in the tool coordinate system.

offsetMotion: The robot arm moves to a position specified by a linear offset to the current position. The offset is defined in the current coordinate system.

collisionOffsetMotion: Like *offsetMotion* but with collision supervision during the motion

In addition to the motion functionality, the motion tasks contain a module that is responsible for the definition of the signals used for the signal acquisition and collision detection. They are defined in these tasks because the arrival of new server data is handled here and signal definitions may have to be changed when such an event occurs.

Collision Detection Task

Collision detection is implicitly included in the proposed manipulation skill concept. A main purpose of the supervision of the current robot state in pose-wrench space during the execution of a skill is the detection of collisions with the environment. When a collision is detected, appropriate interruption motions defined in the motion net of a skill can be triggered. In the implemented system the communication and data processing time delay is too high to use this data directly for collision detection. If a collision is not handled immediately, the robot controller's inherent motion supervision shuts down the system. This motion supervision can be deactivated, but this is not recommended. A motion without immediate motion supervision can easily cause damage to the system or the workpieces. Consequently, a workaround is necessary.

The problem is solved by implementing a "first-aid" collision detection, which triggers a small retraction motion if a critical contact situation is detected. This motion moves the robot to a position that is uncritical regarding the contact forces. The robot remains at this safe position until it receives the next motion command, which is selected based on the estimated contact forces after a delay.

To enable immediate detection of a collision, this detection is based on joint torques that do not need to be processed any further instead of estimated contact forces. Specialized collision detection torques are directly available from the *TestSignal* interface. These torques represent absolute normalized values of external torques. It has proved sufficient for robust collision detection to supervise these torques on four joints per robot arm. When such a torque exceeds a certain threshold value (which can be defined for each motion by the variable *collisionTorque* as decribed above), the current motion is stopped immediately and the retraction motion is initiated. This motion is defined as a motion along the planned motion path in reverse direction.

11.4.2 Communication Between Robot and PC

To supervise the state of the robot on the one hand and to send new execution commands to it on the other hand, a communication interface between the PC application and the robot control program located on the IRC5 controller is necessary. The communication is established by using the controllers *PC Interface*. This interface is designed to enable the controller to communicate with custom PC-based applications via an ethernet connection. To access the interface from the PC side, the PC SDK can be used, which is available with ABB RobotStudio. The PC SDK is a toolkit to develop custom PC applications that can access the IRC5 controller without employing external software. To use the SDK, the C# libraries *ABB.Robotics.dll* and *ABB.Robotics.Controllers.dll* have to be included in the software project. All functionalities that access the SDK for communication purposes are encapsulated in the *FRIDARobotConnectorPCI* class. To use the class, firstly a *Controller* object has to be initialized. This is done by scanning the network for available controllers, creating a *Controller* object from the first controller found, and logging on to this controller. After the *Controller* object has been created, it is used to send and receive robot data, which is described in the following.

Receiving Robot Data

To calculate the current state of the robot in pose-wrench space in the *Contact Force Estimation* module, its joint angles and joint torques have to be received from the robot as input values. These values have to be acquired continuously with a high frequency to enable a reliable supervision of the robot's state. The best performance was achieved by using the *Messaging* domain of the PC SDK. Alternative approaches, like the direct access to RAPID variables, significantly slow down the performance of the robot if they are used at a high frequency. The *Messaging* domain enables a message exchange between the PC and the controller by using a queue on each side. The first queue is located on the robot controller. A task in the RAPID program is continuously reading the required data, stores it as messages, and enqueues the messages in the message queue (see Section 11.4.1). The PC application accesses this queue, retrieves its content and stores the received messages in a second queue on the PC side. From this second queue, the data is successively processed by the *Contact Force Estimation* module. With this technique, which is visualized in Figure 11.3, it is possible to avoid data loss due to a communication and data processing delay. It has to be noted that the queue on the controller side can overflow and cause execution errors if the PC application is not running.

Sending Robot Data

To execute new motion commands, the values of the necessary variables to execute the command as well as command identifier and trigger signal are sent to the robot (see Section 11.4.1). Sending data is not as critical to performance as receiving it, because it is performed infrequently with great time gaps in between compared to the receiving process. Hence, direct access to the RAPID variables can be used which is the easiest way to manipulate the data of the robot. To manipulate a variable, the variable is firstly acquired from the robot controller by the *getRapidData()* function from the *Rapid* domain of the PC SDK. If its value has to be changed, write access to the variable has to be acquired by requesting controller mastership. The value can then be set to any data derived from the *IRapidData* interface of the PC SDK, which is used to represent RAPID data types. After all necessary variables have been set, mastership needs to be disposed again.

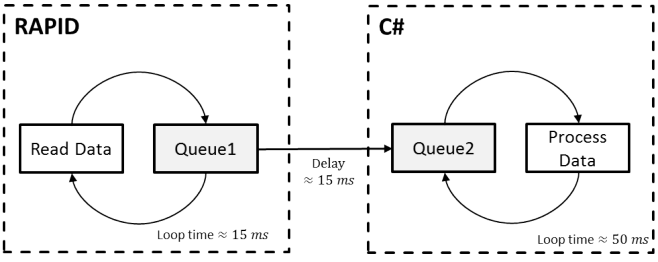


Figure 11.3: Schematic overview of how robot data is received by the main application

11.4.3 Contact Force Estimation and Robot Representation

This section describes how the current robot state in pose-wrench space is calculated, stored and passed to other program parts based on the input data from the RAPID robot control program. The workflow described here refers to the part of the workflow shown in Figure 11.4.4 marked by the upper curly bracket.
To calculate the current robot state, joint angles and joint torques are continuously read from the input data queue of the PC application as described in 11.4.2. Beside angles and torques, supervision data for the robot's tools are included in the input data. Each package of input data is stored in the *ReceivedData* class. When the data in this class is updated, the *ReceivedDataUpdated* event is fired. If the tool supervision data indicates that one of the robot's tools is finished, corresponding tool events are fired. These tool action events can directly be processed on the motion evaluation level.
The *RobotRepresentation* class, which is an internal representation of the employed robotic system, has a listener function that reacts to the *ReceivedDataUpdated* event. When this listener function is called, the joint angle and joint torque data of the robot representation is updated. Furthermore, the calculation of the contact force based on these values is initiated.
The contact force estimation is encapsulated in the class *ContactForceEstimation*, which implements the concept described in Chapter 10. To use this class, it has to be initialized with a set of parameters containing the weighting matrices $\boldsymbol{R}_e$ and $\boldsymbol{R}_F$, which are calculated by an offline optimization (see Section 10.3.3).
To estimate the contact forces, the following steps have to be performed during runtime:

1. Calculation of the robot's forward kinematics from the joint angles and the DH-parameters stored with the *RobotRepresentation*

2. Calculation of the robot's Jacobian from the forward kinematics

3. Estimation of the contact force by using the Jacobian, the weighting matrices and the input joint torques in Equation 10.9

It has to be noted that the input torques are already compensated for friction, as the friction torque is calculated on the RAPID level.
The calculated contact force is stored in the *RobotRepresentation* class. Each time these calculations have been performed, the *RobotRepresentationUpdate* event is fired, which is processed on the motion evaluation level.

11.4.4 Motion Evaluation and Robot Command Execution

The motion evaluation is a crucial part of the PC application, as it supervises the current motion state of the robot, compares it to the desired motion state and triggers transitions in the motion net if necessary. A change of the motion net state leads to the execution of robot motions. This functionality is encapsulated in the class *MotionEvaluation*. It is visualized by the lower part of Figure 11.4.4.
The evaluation is event-driven. Four different events can start an evaluation:

- The skill net enters a new state (event: *FSMSkillStateUpdated*).

- The motion net enters a new state (event: *FSMMotionStateUpdated*).

- The current robot state changes (event: *RobotRepresentationUpdate*).

- A tool action is finished (event: *toolActionFinishedEvent*).

These four cases are described in the following.

New Skill Net State

When the skill net enters a new state, this is indicated by the *FSMSkillStateUpdated* event. The event denotes the start of the execution of a new skill.
If such an event occurs, the *evaluateSkillNetState()* function is called by a referring listener function. This function evaluates the type of the skill net state. A *SkillNetState* only contains a *ParametrizedSkill*, if it is not the initial or the final state in the skill net. If this is not the case, the new skill is initialized for evaluation by the *initializeSkill-State()* function. This function mainly registers the motion net contained in the *ParametrizedSkill* for evaluation and calls the *updateMotionNetState()* function.

New Motion Net State

The *updateMotionNetState()* function is also called when the *FSMMotionStateUpdated* indicates that the current skill's motion net enters a new state. The *evaluateMotionNetState()* function is called, which evaluates the type of the motion net state. If it is "success" or "failure", the currently active transitions in the motion net are evaluated directly without previous motion state comparison. If the state is "final", a transition in the skill net is fired and the next skill is activated. If the motion net state type is "motion" or "tool action", the event listeners for *RobotRepresentationUpdate* and *toolActionFinishedEvent* are initialized.

New Robot State

A *RobotRepresentationUpdate* indicates that the calculation of a new robot motion state is finished as described in 11.4.3. Each time a new robot state occurs, it has to be evaluated and compared to the desired robot state. This evaluation refers to the concept decribed in Section 7.2.4.
To do so, the updated robot state is stored in an internal representation in the *MotionEvaluation* class and an internal event named *currentStateUpdated* is fired. This event causes the execution of the *compareMotionStates* function. Before the desired and the current motion state can be compared, it has to be checked whether the current motion net state is of type "motion". If the type of the motion net state is different, the active transitions in the motion net can directly be evaluated.
If the type is "motion", it is checked if the current state lies within the tolerance region of the desired state. Both states are represented by an instance of the *MotionState* class. In case the current state is outside the allowed region, the currently active transitions are evaluated with respect to the current state.
The transitions are evaluated in the function *evaluateTransitions()*. Essentially, this process is equal to the comparison of motion states, as a transition condition is also represented by a *MotionState*. Contrary to the former process, different operators except "equals" can be used.
If all conditions of a transition are evaluated to be true, the referring condition in the motion net fires. A new state in motion net is entered. Upon entering, a robot command referring to the command type and the control values stored in the state is executed.
To execute a robot command, the *RobotCommandExecution* class is used. For each robot command defined on the RAPID level, an equivalent representation is stored in the *RobotCommandExecution* class. This representation consists of a list of RAPID variables that have to be changed to execute the motion. These variables are set to the values from the *ControlValues* stored in the motion net state. These command specifications are sent to the RAPID program by the *sendRapidData()* function of the *RobotConnection* class.

Tool Action Finished

If the *toolActionFinishedEvent* indicates the end of a tool action, the transitions in the motion net can be evaluated directly as a "tool action" state does not contain motion that has to be supervised.

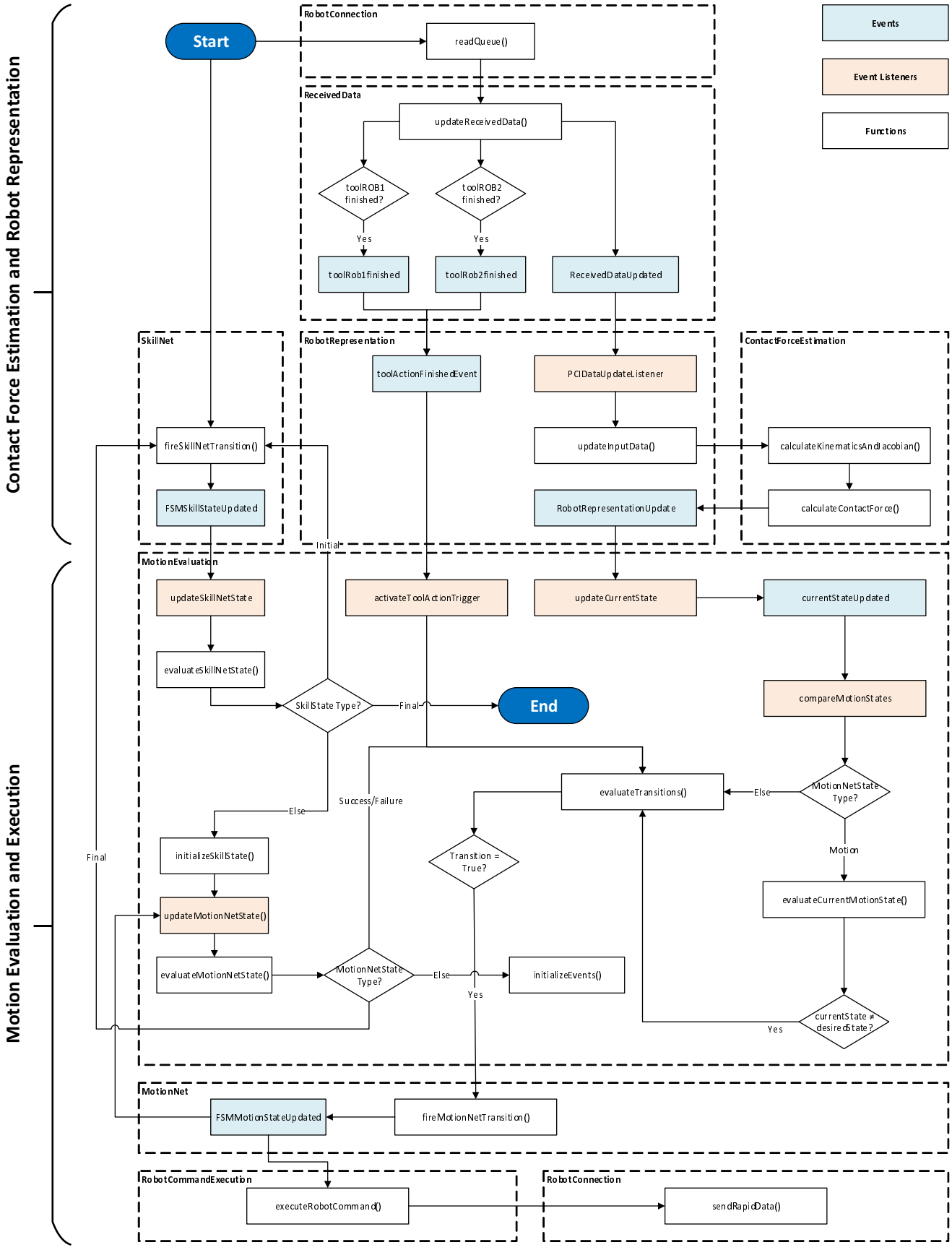


Figure 11.4: The program workflow during the execution of robot motions. This includes the acquisition and evaluation of the current robot state as well as the coordination and execution of the robot's motions.

12 Results

In this chapter, the results obtained from using the implemented concept in an example application are presented. The first section describes the performance of an example assembly sequence, while the second section compares the manual parametrization of a skill template to traditional robot teaching.

12.1 Performance of an Example Assembly Application

This section describes how the example assembly application described in Section 9.3 can be performed by using a set of implemented skill primitives.

12.1.1 Reference Frames and Robot Setup for the Assembly

To perform the example assembly, both arms of the ABB dual-arm concept robot presented in Section 9.1 with 7 degrees of freedom each are used. The right arm is denoted as "Arm 1", while the left arm is denoted as "Arm 2". Both arms can be modeled by the same set of DH parameters as they are identical. Each arm is equipped with an open/close gripper as a tool, which is sufficient for all tasks in this assembly sequence. To allow the grasping of differently sized parts, two different clamps are employed on the gripper, which can be seen in Figure 12.1.
Even though the concept as well as the implemented system can handle the use of a different *Task Frame* for the execution of each parametrized skill, it was sufficient to use one task frame for the execution of the whole assembly. As the motions are controlled by position control in the present implementation, the choice of the task frame is not very critical. All motions in the assembly sequence are coordinated as motions of the tool frames **T1** and **T2**, which are located at the wrist of each arm's tool, relative to the task frame **TF** located at the base of the robot. The kinematic chain between **TF** and **T1** and **T2** describing the robot's forward kinematics is used to calculate Cartesian task frame motions from joint angles. The chain is available from the total transformation matrix between task frame and tool frames. This matrix can be obtained from the arm's DH parameters. Furthermore, a "mounting matrix" has to be considered describing the arm's mount position with respect to the task frame. Figure 12.1 shows the frame locations of **TF**, **T1**, and **T2** at the robot.

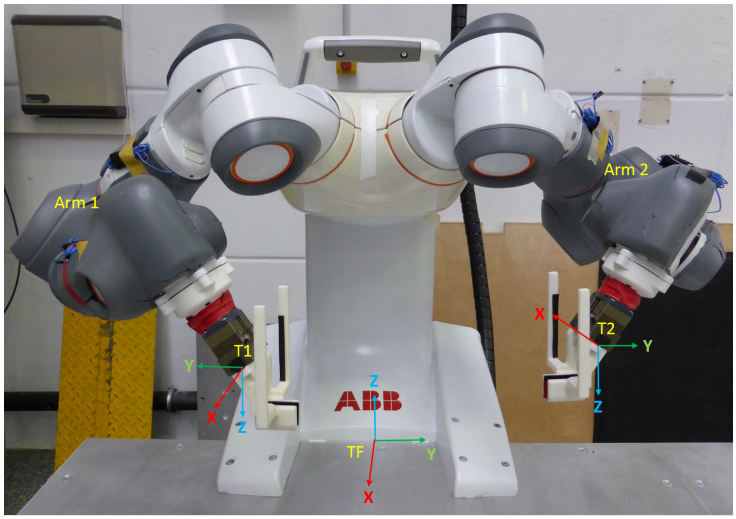


Figure 12.1: Locations of the task frame TF and the tool frames T1 and T2 in the robot workspace

12.1.2 Used Skills in the Assembly Sequence

The overall *ABB PLC I/O Module* assembly sequence described in Section 9.3 was performed using a set of skills implemented in the skill library. Beside the skills *Insert* and *Snapfit* described in Sections 8.1 and 8.2, the simple position-based skills *PickUp*, *Place*, and *Transfer* were used in the sequence (see Appendix C).
Table 12.1 shows an overview of the assembly application described in Chapter 9.3 divided into *Tasks* and *Actions*. This refers to the way a task is described in an assembly tree as presented in Section 6.3. An action is described by its activity as well as the robot arm and the TCP speed it is performed with. For each action, the skill template as well as the index of its parametrization used for the execution of the action is listed. Furthermore, the execution time of each step is presented. The total execution time of the assembly sequence is two minutes, which is slower than the execution of the sequence by a standard position-based RAPID program. The reason for the increase in execution time is the delay caused by the evaluation of transition conditions between the motions. The total execution time includes transfer times for transfers between the various skills.

Task	**Action**			**Skill Template**	**Nr.**	**Time**	**Step**
	Activity	**Arm**	**Speed**				
Provide Housing	PickUp	2	100 mm/s	PickUp	1	3 s	1
	Transfer	2	400 mm/s	Transfer	1	3 s	2
	Place	2	100 mm/s	Place	1	3 s	3
Insert PCB1	PickUp	1	100 mm/s	PickUp	2	3 s	4
	Insert	1	50 mm/s	**Insert**	**1**	5 s	5
Insert PCB2	PickUp	1	100 mm/s	PickUp	3	4 s	6
	Insert	1	50 mm/s	**Insert**	**2**	5 s	7
Insert PCB3	PickUp	1	100 mm/s	PickUp	4	4 s	8
	Insert	1	50 mm/s	**Insert**	**3**	6 s	9
Install Light Guide	PickUp	2	100 mm/s	PickUp	5	3 s	10
	Place	2	100 mm/s	Place	2	2 s	11
Install Light Guide Cover	PickUp	2	100 mm/s	PickUp	6	2 s	12
	Transfer	2	400 mm/s	Transfer	2	2 s	13
	Place	2	100 mm/s	Place	3	3 s	14
	Snap	1	50 mm/s	**Snapfit**	**1**	4 s	15
	Snap	1	50 mm/s	**Snapfit**	**2**	4 s	16
Install Cover	PickUp	1	100 mm/s	PickUp	7	6 s	17
	Transfer	1	100 mm/s	Transfer	3	5 s	18
	Place	1	100 mm/s	Place	4	4 s	19
	Snap	1	50 mm/s	**Snapfit**	**3**	7 s	20

Table 12.1: Skills used in the execution of the *ABB PLC I/O Module* assembly sequence described in Section 9.3

12.1.3 Performance of the Insert Skill

The *Insert* skill is used for the assembly steps 5, 7, and 9 described in Table 12.1. These steps represent the insertion of the PCBs into the housing. To perform the insertions, the PCBs are grasped at their center by the small clamp of the tool and moved into a slot in the housing in z-direction.
The skill template of *Insert* is implemented according to its description presented in Chapter 8.1. Although the general structure coincides with the presented template, some adjustments were made to account for the challenging geometry in the assembly setup.
The clearance in y-direction between the PCB and the housing is very small (1 mm on either side of the PCB). As Figure 12.2 shows, even a small tilt angle θ about the x-axis leads to wall contact and resulting contact forces in z and y-direction, F_z and F_y. As a small tilt is always present, also a contact situation at the ground of the hole leads to an additional contact force F_y and not only in F_z as presented in Chapter 8.1.

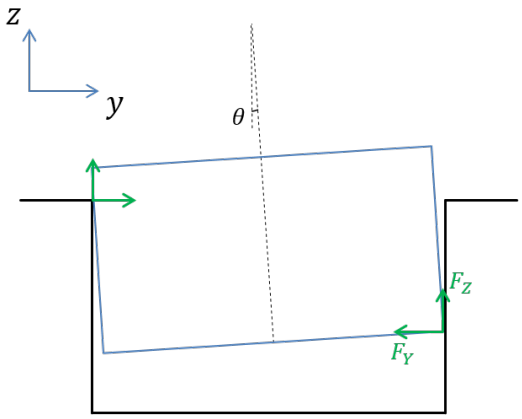


Figure 12.2: PCB Insertion Geometry

Insertion experiments showed that it is more convenient to use F_y for the coordination of the insert motion instead of F_z as presented in Section 8.1. As the part moves in z, the magnitude of F_z during an unconstrained motion is higher than the one of F_y, due to friction and dynamic effects. The compensation of these parts of the total force is only implemented in a basic version, so a certain amount of friction and dynamic forces is still present in the signal. In contrast, forces in y-direction are relatively small during a motion in z-direction. Consequently, it is easier to identify peaks in the force profile of F_y to distinguish contact situations from unconstrained motions. The force profile in y-direction during the consecutive performance of the parametrizations 1,2, and 3 of the insert skill is presented in Figure 12.3.
The execution process of an insertion can be described as follows:

- After the initial position of the insertion, described by the **Precondition**, is reached, a motion in z-direction is started. The motion is specified by an offset. It is not necessary to define this offset precisely, but it needs to be big enough that the bottom of the hole can be reached.
- If a contact force F_y larger than a force threshold $F_{crit,success}$ is detected (see Figure 12.3), the **Completion Criterion** is fulfilled. After completion, the gripper opens and retracts. This mostly happens due to wall contact and not due to ground contact. It is not problematic if the gripper opens in advance to the ground contact, as the PCB falls into place due to the guidance provided by the hole.
- To evaluate the **Quality Criteria** of the skill performance, it is checked if the force is lower than a force threshold $F_{crit,failure}$. If it is larger, the execution failed, otherwise it was successful. The idea behind this definition is to account for jamming situations (see Section 8.1). If F_y exceeds $F_{crit,failure}$, it is very likely that jamming occurred. In this case the PCB will not fall into place after opening the gripper, but is jammed between the walls of the housing.
- The same condition that describes the quality can be used as an **Interrupt Condition** during the insertion. If $F_{crit,failure}$ is exceeded during insertion, a motion to handle the interruption can be triggered. As different interruption motions are imaginable, like for example a reorientation of the PCB or a retry with reduced speed, a simple retry was implemented. The PCB is moved back to the initial position of the insertion and the insertion motion is executed again.

The following values were determined for the force thresholds: $F_{crit,success} = 0.8N$, $F_{crit,failure} = 1.0N$. It has to be noted that these values might differ from the actual acting forces as they represent estimated values.

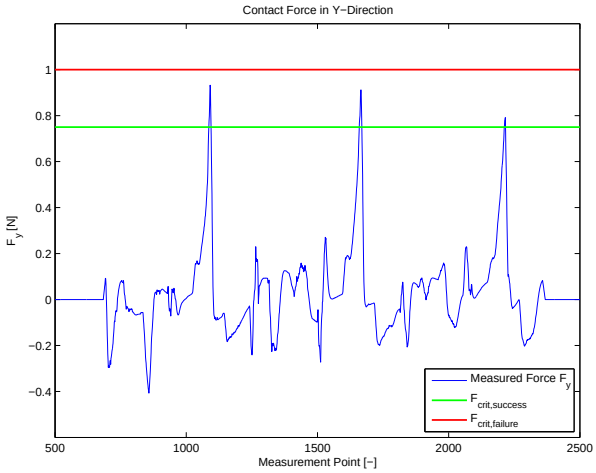


Figure 12.3: MATLAB plot of the estimated force in y-direction during the execution of the *Insert* skills 1, 2, and 3. The force was recorded at a frequency of 40 Hz. The shown force values might differ from the actual acting force as they are estimated forces based on the presented contact force estimation scheme (see Section 10).

12.1.4 Performance of the Snapfit Skill

The *Snapfit* skill is used for the assembly steps 15, 16, and 20 described in Table 12.1. These steps represent the tightening of the light guide cover to the cover plate, as well as the fitting of the cover to the housing. Snaps 1 and 2 are performed by pushing the tip of the gripper onto the snap-fit latches of the light guide cover. Snap 3 is performed by pushing the cover down with one the robots foam paddings as visualized in Figure 12.4. All motions are performed in z-direction.

The skill template of *Snapfit* is implemented according to its description presented in Chapter 8.2. As presented in that chapter, the trajectory of *Snapfit* can be simplified to a single force peak if the phase between the bending of the latch and the final ground contact is very short. In this case, the total height difference the latch has to pass during the snap is only 2 mm for the snaps 1 and 2 and 3 mm for snap 3. It can be assumed that the simplification is admissible under these circumstances. As the main motion direction of the snap-fits is z, the contact force F_z is used to coordinate it. The force profiles for snaps 1, 2, and 3 are visualized in Figure 12.5.

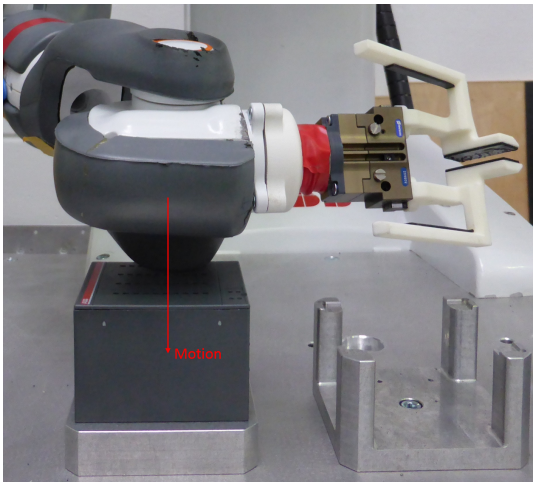


Figure 12.4: Attaching the cover to the housing by executing *Snapfit* skill 3

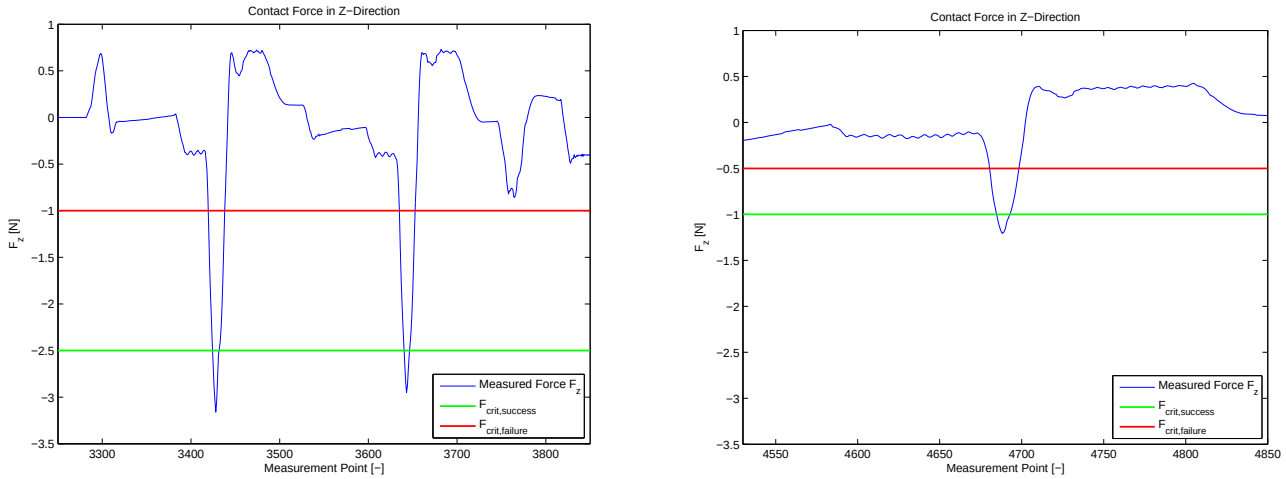
The execution process of a snap-fit can be described as follows:

- After the initial position of the insertion, described by the **Precondition**, is reached, a motion in z-direction is started. The motion is specified by an offset. It is not necessary to define this offset precisely, but it needs to be big enough so that the part to be snapped is reached.

- If a contact force F_z is detected that is larger than the force threshold $F_{crit,failure}$ (see Figure 12.5), the **Completion Criterion** is fulfilled. After completion, the robot arm retracts. $F_{crit,failure}$ has to be large enough to ensure that the latch is bent and passes the guidance.

- To evaluate the **Quality Criteria** of the skill performance, it is checked, if the force is large enough. If the force is smaller than the force threshold $F_{crit,success}$, it can be assumed that the snap was not performed successfully. In this case, a retry of the skill can be triggered. Otherwise, the snap is assumed to be successful and the execution of the next skill can be started.

As it can be seen in Figure 12.5, the trajectories for snaps 1 and 2 differ from the one for snap 3. The different trends of the curve can be explained by the different material pairings involved in the snap: While in snaps 1 and 2 the plastic gripper meets the plastic light guide cover, in snap 3 the soft foam padding meets the plastic cover as can be seen in Figure 12.4. This leads to slower increase of the contact force and a smoother curve resulting from it. On the one hand, the lower magnitude can be explained by the different geometry, as the latches employed in snap 3 are smaller than the latches in 1 and 2. On the other hand, it has to be taken into account that the contact force is estimated at the tool frame located at the robot arm's wrist while the contact in snap 3 is established with a padding on the arm. Obviously, the force measured at the tool frame has to be smaller than in a direct contact situation.
For snaps 1 and 2, force thresholds of $F_{crit,success} = -2.5N$ and $F_{crit,failure} = -1.0N$ were used, while for snap 3 thresholds of $F_{crit,success} = -1.0N$ and $F_{crit,failure} = -0.5N$ were used. It has to be noted that these values might differ from the actual acting forces as they represent estimated values.



(a) Snapfits 1 and 2

(b) Snapfit 3

Figure 12.5: MATLAB plot of the estimated force in z-direction during the execution of different *Snapfit* skills. The force was recorded at a frequency of 40 Hz. The shown force values might differ from the actual acting force as they are estimated forces based on the presented contact force estimation scheme (see Section 10).

12.2 Comparison of Manual Skill Parametrization and Robot Teaching

One of the main reasons to use encapsulated skill primitives is to ease the programming of a robot (see for example Chapter 1.1). In this section, the traditional way to teach a motion procedure to a robot is compared to the manual parametrization of a skill primitive designed to fulfill a similar motion. For this comparison, the execution of the snap-fits 1 and 2 (see Table 12.1) were programmed on the ABB dual-arm concept robot (DACR) used in the scenario described in the previous sections.
Traditionally, the easiest way to teach a motion procedure to an ABB robot is by modifying a sequence of motion targets predefined in a RAPID program by using programming by demonstration. To do so, the following steps have to be performed for each target contained in the motion sequence:

- A rough positioning of the arm is done by manually moving it to a position. This can only be done on the DACR or other small robots, as the arms of traditional industrial robots are usually too heavy to carry them manually.
- The fine positioning of the arm is done by jogging it to the desired position using the joystick on the *FlexPendant* controller input device (see Section 9.2). Before jogging can be used, the motors have to be switched on and the incremental jogging mode has to be selected to gain a higher precision. Instead of using jogging, which is very slow, the fine positioning can also be done manually. In most situations this requires two persons to hold the robot arm to gain a decent precision.
- The position has to be stored in a predefined RAPID variable. Usually the structure of a motion sequence is programmed in advance by defining a series of motion commands which each use a target variable. The values of the target variable can be set to the current position of the robot arm in the "Program Editor" mode on the *FlexPendant*. This editor is capable of editing the RAPID programs currently stored on the robot controller. If no target variable to modify is available, a RAPID variable can be created in the editor.

In the case of a snap-fit motion, three targets have to be programmed using the previously described process. The snap-fit motion is specified by one position in advance to the snap, one position for retraction from the snap and one position where the actual snap happens. While the former two positions have lower requirements on precision, the latter has to be set precisely. By the positioning, contact between the robot's gripper and the latch of the snap-fit has to be enabled. If the snap position is not set precisely, either the snap does not work because the contact forces are too small, or the motion supervision of the controller stops the program execution because the contact forces are too high.
If skill primitives are used, the contact force during the motion is supervised. This fact eliminates the need for high precision position teaching. To teach a motion sequence by using a skill template, a manual parametrization process can be employed to parametrize the template. In this process, the assembly tree is used instead of a RAPID program to define the structure of a motion sequence. Contrary to the automatic parametrization, the parameters defined in the assembly tree to parametrize a skill template are partially or completely overwritten with manually set values. It is possible to set a value by moving the robot arm to a desired position and use this position for the parametrization. Also, values can be set by written input.
For the parametrization of one *Snapfit* skill template, only two parameters have to be set: The position prior to the snap and the force threshold denoted as $F_{crit,success}$ in the previous section. The position does not have high precision requirements, so it can be set completely manual without using jogging. It has to be noted that the setting of the offset is omitted here. As described in Section 12.1.4, its value does not need to be set precisely, it only has to be large enough to reach the snap position. For the sake of simplicity, the value defined in the assembly tree was used in the manual parametrization of the *Snapfit* skill template. Another possibility would have been to predefine the offset as a very large value in the skill template.
The manual parametrization was evaluated for the installation of the light guide cover. This includes the snaps 1 and 2 specified in Table 12.1. Using traditional teaching, it took **1 min 30 s** to teach one snap-fit motion. This time was reduced to **30 s** per snap-fit by employing the manual parametrization of skill templates.

Part IV
Summary

13 Conclusion

In this thesis, a new *Manipulation Skill* concept to fulfill robotic assembly tasks was developed, implemented and used for an example application. To accomplish this, several steps were necessary that are described in the following.

Firstly, the state-of-the-art of *Manipulation Skill* concepts was captured and summarized. In Part I, the requirements and individual components necessary for a skill concept as well as existing skill approaches were presented. It was determined that most approaches employ compliant motion specifications like the Task Frame Formalism, advanced sensor-based control strategies and Finite State Machines to coordinate robot motions.

A new skill representation was proposed, in which skills are stored as reusable templates that can be applied to new situations by parametrization. The parameters are derived from an assembly specification. Each skill has two main components: a 12-dimensional trajectory describing compliant motions in pose-wrench space and a Finite State Machine (FSM) to execute the motions that are derived from this trajectory. As the trajectory is not explicitly dependent on time, it can represent robot motion independently of time-related parameters like the robot's speed. A qualitative node-based concept was proposed to store the trajectory in a generic way. The motions in the FSM are based on the Task Frame Formalism. While the motions represented by states in the FSM are position-controlled, the transitions in the FSM can also be based on force measurements.

Skill templates were created and categorized according to the proposed representation. Two of the skill templates, *Insert* and *Snapfit*, were investigated and modeled in detail. Other skills were presented in brief, like the position-based skills *Transfer*, *PickUp*, and *Place*, which were necessary for the example application. Beside these templates, which were all implemented in the demonstrator, the two skills *Screw* and *Bayonet Mount*, which were not implemented, were presented in short.

A framework was developed to embed the proposed skill representation in a system that is able to coordinate robot motions. This includes mapping procedures to map parameters between the assembly specification, the trajectory representation and the Finite State Machine. Furthermore, interfaces to robot control and sensing as well as to the assembly specification are included. A main task of the framework is to compare a desired motion state derived from the skill representation to the current robot state derived from the robot itself. Another task is the execution of the robot motions defined in the skill representation.

A new assembly specification used as a data source for the parametrization of skill templates was introduced. This specification is an assembly tree structure which incorporates descriptive information about an assembly as well as instructions on how to perform an assembly.

The proposed concept was implemented as a demonstrator consisting of an assembly tree XML structure, a C# application containing the skill representation and framework as well as a robot control program written in the ABB RAPID programming language to carry out robot commands. By using the RAPID language as an interface to the robot, the applicability of the system to standard industrial control software was shown.

The implemented demonstrator was used to carry out an example application using an ABB dual-arm concept robot equipped with an ABB IRC5 robot controller. In the example application, an ABB PLC I/O Module was assembled using the skills *Insert*, *Snapfit*, *Transfer*, *PickUp*, and *Place*. Beside the general applicability of the concept to such an example application, the ease of use was demonstrated for the *Snapfit* skill. For this purpose, a comparison was made between the time it takes to manually parametrize a *Snapfit* skill (30 s) and the time it takes to teach an alike motion by only using RAPID (1 m 30 s).

Finally, a novel approach to estimate the robot's external wrench solely based on motor torques and joint angles was proposed to circumvent the need for dedicated force/torque sensors. It employs the identification of a simple friction model as well as the tuning of covariance matrices in a Bayesian approach by means of calibration measurements and the solution of an optimization problem. The results accomplished in this area were published in [79].

14 Discussion

In this chapter, advantages and drawbacks of the different components of the proposed concept are discussed.

Skill Representation And Categorization

The proposed skill representation, especially the 12-dimensional pose-wrench trajectory representation, is a novel way to describe motions or other robot actions as primitives. Its main advantage is that it is a very general representation that is well suited to motion supervision. By analyzing the trajectory, complex patterns can be detected and used to trigger actions. The proposed generalization as a trajectory template allows the use in generic reusable skill primitives. A trajectory representation is also well-suited to be acquired by demonstration and refined by learning. The drawback at this point is that the full potential of this representation is currently not used. The trajectory is currently created manually, like the rest of a skill template. Furthermore, instead of complex pattern analysis, only thresholds are considered.

Beside the novel trajectory representation, a tool is used in the skill representation that is well known and easily extendable: The Finite State Machine motion net. By adding this more concrete level to the abstract trajectory skill representation, the usability of the concept is increased. A drawback is that parameter mappings between the two representations are necessary.

The motion specification employed in the motion net is based on the Task Frame Formalism (TFF), which allows a very intuitive specification of motion commands. It is flexible and applicable to a wide variety of assembly tasks as it supports different control modes (like for example force or position) for different motion directions. The full potential of the TFF could not be used in the implementation, as the currently employed robot controller only supports position control. Furthermore, the used contact force estimation scheme is not yet reliable enough to be used as a reference for force control.

The skill categorization based on the proposed skill representation provides a set of properties by which skills can be classified. This, together with the list of skills presented in this thesis is a good starting point for future skill developments.

Assembly Tree Specification

The proposed assembly tree specification contains all necessary information that are necessary to parametrize skill templates. The concept and its implementation as an XML data structure is kept minimal, easy to understand, and intuitive. While all necessary data for the present application is included, the approach lacks generality as it was created based on the requirements of the *Manipulation Skill* concept. Another drawback is that it has to be created manually. A systematic approach to create it automatically, for example from CAD data, was out of the scope of this thesis.

While the results in this area are nicely applicable for the proposed concept, they remain a tailored solution. The representation and creation of assembly specifications are extensive topics on their own that cannot be solved in a general way in one thesis alone. The proposed representation is a valuable starting point for further developments of assembly specifications for skill-based concepts.

Implementation and Example Application

The implementation applies the proposed concept to an example application. All basic properties and functions of the concept are shown in the example. Only standard industrial robot control software and hardware was used in the implementation, which shows the applicability of the concept to such resources. The implementation has to be understood as a demonstrator software, showing the basic functionality of the system and not a complete software package, which is able to carry out all kinds of robotic tasks. It is, for example, currently tailored to a specific robotic platform, the ABB dual-arm concept robot, and needs some adaptions before it can be used on other platforms.

The performance of the demonstrator is currently mainly limited by the used communication interface. Because of the communication delay introduced by this interface, the collision detection workaround described in Section 11.4.1 is necessary and contact forces cannot be used directly for collision detection. Furthermore, the interface used for the acquisition of the input signals for the contact force estimation has some major drawbacks. Most importantly, it is limited to the definition of 12 signals at a time. Also, it is very difficult to identify the correct signals with this interface.
The shown example application demonstrates all important functions, but to demonstrate the full generality of the concept, a more complex task featuring a wider selection of skills is necessary.
While the implementation is able to demonstrate the basic functionality, there are several program parts where modifications could increase the performance and generality of the implementation. The performance could easily be increased if different interfaces for signal acquisition and communication were available.

Contact Force Estimation

The proposed contact force estimation scheme only used motor torques and joint angles to estimate the external wrench of the robot, which is a novelty in the context of skill. It enabled the use of the proposed concept without any dedicated force/torque sensor. As the measurements presented in Sections 12 and 10.4 showed, force peaks can easily be identified. While this qualitative information can be extracted, it cannot be guaranteed that the quantitative force values are consistent with the real acting forces. The reasons are that there is no guaranteed information about the scaling of the torque values acquired from the *Testsignal* interface. As no sensor that could be used for comparison was available, it could not be verified if the calibration process described in Section 10.3.3 is scaling the estimated forces to the correct magnitudes.
Another drawback is that a new calibration process is currently necessary for each motion type (e.g. vertical and horizontal motion). This decreases the generality of the approach. Furthermore, the optimization process of the calibration is currently tailored to the estimation of forces instead of torques (see Appendix B), which means that the estimated contact torques are currently not useful.
It has to be noted that the employed friction model in the contact force estimation is a very simple one (Coulomb and viscous friction), which leads to remaining friction disturbances in the force signal.
The obtained results were very useful for this specific project, as force-based motion executions were made possible without a dedicated force/torque sensor. Before the approach can be used in general, some modifications to solve the previously described problems are necessary though. The contact force estimation will be further investigated and refined in a dedicated project in the future.

In summary, the created concept works well for the shown example application and bears great potential for other more complex applications. It demonstrates how a *Manipulation Skill* interface, which was build from scratch, can be used to coordinate different individual components of a robotic manufacturing system. The main drawbacks of the system are in the various interfaces and individual components and not at skill level. Despite the presented drawbacks, a strong foundation for future developments was built.

15 Outlook

This chapter presents possible extensions to different components of the *Manipulation Skill* concept. The presented topics were not considered in this thesis but are potential subjects to future projects.

Automatic Skill Acquisition

A skill template, including the trajectory and motion net representation, in the present concept is created manually. It would increase the usability drastically, if skill templates could be acquired automatically, for example from human demonstrations. In [31], a method is proposed to use programming by demonstration for the acquisition of robotic skill. The presented approach uses human performances of specific tasks to record sets of training data that are used to learn skills. The data contains the humans motion data. This learning also includes the generation of a suited control strategy. An alike approach could be employed to create the trajectory representation featured in the concept presented in this thesis. By recording the object poses and acting contact forces during a human performance of a task that refers to a specific skill, a specific trajectory can be recorded. If multiple recordings are created, this data can be used as training data to refine the recorded trajectory by learning approaches. To obtain a generic template from the recorded trajectory, patterns like peaks or constant sections have to be recognized in the multiple dimensions of the trajectory. From these patterns, trajectory nodes can be generated. To convert a recorded trajectory to a motion net, unique mapping rules have to be defined. These rules can possibly also be acquired by learning based on a set of example trajectory and motion net pairs.

Automatic Assembly Planning

An alternative to the manually created assembly tree representation presented in Section 6.1 is an assembly specification that can automatically be derived from assembly planning systems. In [50] an approach that automatically creates assembly plans from CAD data and maps these plans to skill primitives is presented. It uses a relational assembly model to determine feasible assembly tasks. This model includes the assembly components as constructive solid geometry (CSG) representations but other types of CAD data can also be used. From the relational model, an assembly sequence is generated by employing an assembly-by-disassembly strategy combined with a cut set method. Local and global assembly constraints like geometry feasibility or assembly stability are considered in the sequence generation. The set of all feasible assembly sequences is represented in an AND/OR graph. As there can be many possible sequences, the best one has to be selected by employing evaluation criteria. Afterwards, the operations in the AND/OR graph are classified and mapped to robot tasks. The robot tasks are then decomposed into skill primitives.

Advanced Force Sensing

The contact force estimation scheme presented in Section 10 is a novel way to work with contact forces without employing a dedicated force sensor. Even though it worked well for the presented example application, it is still just a basic version that has a lot of potential for further extensions. One possible improvement is the employment of a more sophisticated friction model as described in [7]. In addition to Coulomb and viscous friction, the Stribeck effect can be considered in the friction model. Dynamic effects like hysterese can be captured by dynamic friction models. A friction model that takes all known effects into account is the *LuGre* friction model. To further improve the friction identification, an online identification instead of an offline process can be employed.

Another improvement to the contact force estimation scheme would be to increase the quality of the input data. This can be accomplished by employing a different interface to obtain the input signals. As presented in Section 11.4.1, the employed *Testsignal* interface can only provide a limited amount of signals and unreliable signal scaling. Furthermore, the frequency at which forces can be estimated would be increased by providing a faster communication interface.

To further improve the estimation scheme, it is necessary to provide a force/torque sensor as a baseline to compare the estimated values to. A sensor could also be used to ease the weighting matrix calibration process described

in Section 10.3.3. An optimized way to obtain forces would open new possibilities like using force-controlled motions.

Advanced Force Analysis

To supervise the robot's motion during assembly based on the 12-dimensional pose-wrench trajectory, the current concept only considers the use of threshold values in one or many of the 12 dimensions. A greater flexibility can be gained by using complex pose-wrench signatures to recognize special events like contacts during the assembly. In [58], the force signature recorded by a force sensor is used to determine the success or failure of a small part assembly process. A supervised learning approach using Support Vector Machines (SVM) is employed. Using this SVM, a classifier is trained that automatically learns the decision rule between success and failure. A set of signatures of successful and unsuccessful processes is necessary as training data. This approach can be extended to signatures in multiple dimensions, like for example the 12 dimensions of the pose-wrench trajectory, which would increase dimensionality of the feature space of the SVM.

Workspace Object Recognition

To parametrize a skill primitive, explicitly defined workspace positions are necessary in the concept presented in this thesis. The positions are passed to a skill primitive in the parametrization process. An easier way to use skill primitives would be a parametrization based on objects. This would enable the robot to understand commands like "PickUp Red Box". To permit the handling of objects like "Red Box" on a robotic platform, an object recognition and localization scheme as well as suitable sensing hardware is necessary.
Objects can be recognized by identifying features in images obtained from visual sensors. In [6] an approach is presented where the color and depth image information from an RGB-D camera is used to identify features. The features itself are obtained by an unsupervised learning approach from raw image training data. An extension to this passive recognition approach is the introduction of an active recognition scheme. In [10] a scheme is proposed that links perception directly to action and enables the robot to either move the part or the camera during the recognition process. The idea behind this approach is to resolve the ambiguity between different 3D objects that look similar in a 2D projection.
For the spatial localization of objects in the workspace, the depth information used for feature recognition in [6] can be employed as well. Instead of depth image data, also stereo image data can be used for the localization task like, for example, employed in [42]. To obtain information about a 3D location of an object from 2-dimensional images, a triangulation is performed on two images that represent the object captured from different directions. The intrinsic parameters of the cameras as well as the rigid body transformations between the cameras and the robot reference frame have to be known to perform the triangulation. The presented approach employs Neural Network machine learning to acquire the camera parameters and transformations, which eliminates the need for prior calibration.

Appendix

A Derivation of the Contact Force Estimation

This appendix describes how the scheme for the estimation of the external wrench $\boldsymbol{f}$ is derived. The disturbances $\boldsymbol{e}$ and the external wrench $\boldsymbol{f}$ are modeled as random variables, where $\mathbb{E}[\boldsymbol{e}] = \boldsymbol{0}$, $\mathbb{E}\left[\boldsymbol{f}\right] = \boldsymbol{0}$ is assumed. While zero mean is thus assumed for both variables, the covariance matrices are denoted by $\mathrm{Var}[\boldsymbol{e}] = \mathbb{E}\left[\boldsymbol{e}\cdot\boldsymbol{e}^{\mathsf{T}}\right] = \boldsymbol{R}_e$ and $\mathrm{Var}\left[\boldsymbol{f}\right] = \mathbb{E}\left[\boldsymbol{f}\cdot\boldsymbol{f}^{\mathsf{T}}\right] = \boldsymbol{R}_F$, and the variables $\boldsymbol{f}$ and $\boldsymbol{e}$ are assumed to be uncorrelated, i.e. $\mathbb{E}\left[\boldsymbol{f}\cdot\boldsymbol{e}^{\mathsf{T}}\right] = \boldsymbol{0}$, $\mathbb{E}\left[\boldsymbol{e}\cdot\boldsymbol{f}^{\mathsf{T}}\right] = \boldsymbol{0}$.
To incorporate this knowledge into the estimation of $\boldsymbol{f}$ given $\overline{\boldsymbol{\tau}}_{\mathrm{mot}}$, the error between the actual wrench $\boldsymbol{f}$ and the estimated wrench $\hat{\boldsymbol{f}}$ as $\boldsymbol{\delta} = \boldsymbol{f} - \hat{\boldsymbol{f}}$ is denoted. The objective then is to find a matrix $\boldsymbol{W}$ minimizing the total variance of $\boldsymbol{\delta}$ resulting from $\hat{\boldsymbol{f}} = \boldsymbol{W}\cdot\overline{\boldsymbol{\tau}}_{\mathrm{mot}}$. Since the total variance of $\boldsymbol{\delta}$ is given by $\mathrm{tr}\left(\boldsymbol{R}_{\boldsymbol{\delta}}\right) = \mathrm{tr}\left(\mathbb{E}\left[\boldsymbol{\delta}\cdot\boldsymbol{\delta}^{\mathsf{T}}\right]\right)$ and $\boldsymbol{\delta} = \boldsymbol{f} - \boldsymbol{W}\overline{\boldsymbol{\tau}}_{\mathrm{mot}}$ is valid, the above reasoning can be summarized as

$$\underset{\boldsymbol{W}}{\text{minimize}}\ \mathrm{tr}\left(\mathbb{E}\left[\boldsymbol{\delta}\cdot\boldsymbol{\delta}^{\mathsf{T}}\right]\right)\ \text{s.t.} \tag{A.1a}$$

$$\boldsymbol{\delta} = \boldsymbol{f} - \boldsymbol{W}\overline{\boldsymbol{\tau}}_{\mathrm{mot}}. \tag{A.1b}$$

With Equation 10.6, the error covariance $\boldsymbol{R}_{\boldsymbol{\delta}}$ can be expressed as

$$\begin{aligned}
\boldsymbol{R}_{\boldsymbol{\delta}} &= \mathbb{E}\left[\left(\boldsymbol{f} - \boldsymbol{W}\overline{\boldsymbol{\tau}}_{\mathrm{mot}}\right)\cdot\left(\boldsymbol{f} - \boldsymbol{W}\overline{\boldsymbol{\tau}}_{\mathrm{mot}}\right)^{\mathsf{T}}\right] \\
&= \mathbb{E}\Bigg[\left(\left(\boldsymbol{I}_6 - \boldsymbol{W}\boldsymbol{J}^{\mathsf{T}}\right)\boldsymbol{f} - \boldsymbol{W}\boldsymbol{e}\right)\cdot\ldots \\
&\qquad\qquad \cdot\left(\left(\boldsymbol{I}_6 - \boldsymbol{W}\boldsymbol{J}^{\mathsf{T}}\right)\boldsymbol{f} - \boldsymbol{W}\boldsymbol{e}\right)^{\mathsf{T}}\Bigg] \\
&= \left(\boldsymbol{I}_6 - \boldsymbol{W}\boldsymbol{J}^{\mathsf{T}}\right)\mathbb{E}\left[\boldsymbol{f}\cdot\boldsymbol{f}^{\mathsf{T}}\right]\left(\boldsymbol{I}_6 - \boldsymbol{W}\boldsymbol{J}^{\mathsf{T}}\right)^{\mathsf{T}} + \ldots \\
&\quad - \left(\boldsymbol{I}_6 - \boldsymbol{W}\boldsymbol{J}^{\mathsf{T}}\right)\mathbb{E}\left[\boldsymbol{f}\cdot\boldsymbol{e}^{\mathsf{T}}\right]\boldsymbol{W}^{\mathsf{T}} + \ldots \\
&\quad - \boldsymbol{W}\mathbb{E}\left[\boldsymbol{e}\cdot\boldsymbol{f}^{\mathsf{T}}\right]\left(\boldsymbol{I}_6 - \boldsymbol{W}\boldsymbol{J}^{\mathsf{T}}\right)^{\mathsf{T}} + \ldots \\
&\quad + \boldsymbol{W}\mathbb{E}\left[\boldsymbol{e}\cdot\boldsymbol{e}^{\mathsf{T}}\right]\boldsymbol{W}^{\mathsf{T}},
\end{aligned} \tag{A.2}$$

where the argument of the Jacobian matrix has been omitted for the sake of notational brevity. Due to the prior assumptions on $\boldsymbol{f}$ and $\boldsymbol{e}$, Equation A.2 can be simplified to

$$\boldsymbol{R}_{\boldsymbol{\delta}} = \left(\boldsymbol{I}_6 - \boldsymbol{W}\boldsymbol{J}^{\mathsf{T}}\right)\boldsymbol{R}_F\left(\boldsymbol{I}_6 - \boldsymbol{W}\boldsymbol{J}^{\mathsf{T}}\right)^{\mathsf{T}} + \boldsymbol{W}\boldsymbol{R}_e\boldsymbol{W}^{\mathsf{T}}. \tag{A.3}$$

To analytically solve Equation A.1, the matrix derivative rules (cf. e.g. [56])

$$\frac{\partial\,\mathrm{tr}\left(\boldsymbol{A}\boldsymbol{X}^{\mathsf{T}}\right)}{\partial\boldsymbol{X}} = \boldsymbol{A}, \tag{A.4a}$$

$$\frac{\partial\,\mathrm{tr}\left(\boldsymbol{X}\boldsymbol{A}\right)}{\partial\boldsymbol{X}} = \boldsymbol{A}^{\mathsf{T}}, \tag{A.4b}$$

$$\frac{\partial\,\mathrm{tr}\left(\boldsymbol{X}\boldsymbol{A}\boldsymbol{X}^{\mathsf{T}}\right)}{\partial\boldsymbol{X}} = \boldsymbol{X}\boldsymbol{A}^{\mathsf{T}} + \boldsymbol{X}\boldsymbol{A} \tag{A.4c}$$

are employed. They allow the computation of

$$\frac{\partial\,\mathrm{tr}\left(\boldsymbol{R}_{\boldsymbol{\delta}}\right)}{\partial\,\boldsymbol{W}} = -2\boldsymbol{R}_F\boldsymbol{J} + 2\boldsymbol{W}\left(\boldsymbol{J}^{\mathsf{T}}\boldsymbol{R}_F\boldsymbol{J} + \boldsymbol{R}_e\right). \tag{A.5}$$

Since $\partial \operatorname{tr}\left(\boldsymbol{R}_{\delta}\right) / \partial \boldsymbol{W}=\mathbf{0}$ is a necessary condition for an extremum in Equation A.1, the matrix $\boldsymbol{W}$ is given by

$$\boldsymbol{W}=\boldsymbol{R}_F \boldsymbol{J}\left(\boldsymbol{J}^{\mathsf{T}} \cdot \boldsymbol{R}_F \cdot \boldsymbol{J}+\boldsymbol{R}_e\right)^{-1}. \tag{A.6}$$

Note that due to the fact that Equation A.1 is a convex problem, $\partial \operatorname{tr}\left(\boldsymbol{R}_{\delta}\right) / \partial \boldsymbol{W}=\mathbf{0}$ is also sufficient.
The optimal solution for the contact force estimation based on $\overline{\boldsymbol{\tau}}_{\text{mot}}$ and the prior knowledge on $\boldsymbol{f}$ and $\boldsymbol{e}$ is given by

$$\hat{\boldsymbol{f}}=\boldsymbol{R}_F \boldsymbol{J}(\boldsymbol{q}) \cdot\left(\boldsymbol{J}^{\mathsf{T}}(\boldsymbol{q}) \cdot \boldsymbol{R}_F \cdot \boldsymbol{J}(\boldsymbol{q})+\boldsymbol{R}_e\right)^{-1} \cdot \overline{\boldsymbol{\tau}}_{\text{mot}}. \tag{A.7}$$

B Remarks on Contact Force Estimation

Remark 1 *For the special case of* $\boldsymbol{R}_F = \boldsymbol{I}_6$, $\boldsymbol{R}_e = \boldsymbol{0}$, *Equation 10.9 reduces to Equation 10.8, since*

$$\boldsymbol{W} = \boldsymbol{J}(\boldsymbol{q}) \cdot \left(\boldsymbol{J}^{\mathsf{T}}(\boldsymbol{q}) \cdot \boldsymbol{J}(\boldsymbol{q})\right)^{-1} = \left(\boldsymbol{J}^{\mathsf{T}}(\boldsymbol{q})\right)^{+} \tag{B.1}$$

is obtained. The interpretation of this fact is that the estimation scheme (Equation 10.8) implicitly assumes that the gravity compensation as well as the dynamic feed-forward compensation and friction estimation are exact, resulting in $Var\left[\boldsymbol{e}\right] = \boldsymbol{0}$. *Furthermore, Equation 10.8 implies that the variance of all elements in* $\boldsymbol{f}$ *is the same.*

Remark 2 *Note, that*

$$\begin{aligned} &\boldsymbol{R}_F \boldsymbol{J}(\boldsymbol{q}) \left(\boldsymbol{J}^{\mathsf{T}}(\boldsymbol{q}) \cdot \boldsymbol{R}_F \cdot \boldsymbol{J}(\boldsymbol{q}) + \boldsymbol{R}_e\right)^{-1} \\ &\quad = \left(\boldsymbol{J}(\boldsymbol{q}) \cdot \boldsymbol{R}_e^{-1} \cdot \boldsymbol{J}^{\mathsf{T}}(\boldsymbol{q}) + \boldsymbol{R}_F^{-1}\right)^{-1} \boldsymbol{J}(\boldsymbol{q}) \cdot \boldsymbol{R}_e^{-1} \end{aligned} \tag{B.2}$$

can be shown by employing the Woodbury matrix identity (cf. e.g. [56]). Thus, Equation 10.9 is equivalent to the estimation formula

$$\hat{\boldsymbol{f}} = \left(\boldsymbol{J}(\boldsymbol{q}) \cdot \boldsymbol{R}_e^{-1} \cdot \boldsymbol{J}^{\mathsf{T}}(\boldsymbol{q}) + \boldsymbol{R}_F^{-1}\right)^{-1} \cdot \boldsymbol{J}(\boldsymbol{q}) \cdot \boldsymbol{R}_e^{-1} \overline{\boldsymbol{\tau}}_{mot} \tag{B.3}$$

proposed in [66] for invertible matrices $\boldsymbol{R}_e$, $\boldsymbol{R}_F$. *Note, that Equation 10.9 has the advantage of avoiding inversions of* $\boldsymbol{R}_e$ *and* $\boldsymbol{R}_F$ *and thus only positive semi-definiteness of the weighting matrices is necessary. Furthermore, it is emphasized that while the contact force estimation equation is similar, the matrix* $\boldsymbol{R}_e$ *has a different interpretation in our approach compared to [66]. It does not only describe the covariance of friction torques but of the complete disturbance torque* $\boldsymbol{e}$ *comprised of errors in the dynamic feed-forward, gravity compensation and friction compensation.*

Remark 3 *In many applications, only contact forces (and no torques) have to be estimated. In order to exploit this in the calibration, the optimization problem (Equation 10.16) can be slightly modified to*

$$\underset{\boldsymbol{R}_e, \boldsymbol{R}_F}{minimize} \sum_{k=1}^{N} \left(\hat{\boldsymbol{f}}^k - \boldsymbol{f}_{def}\right)^{\mathsf{T}} \cdot \boldsymbol{M} \cdot \left(\hat{\boldsymbol{f}}^k - \boldsymbol{f}_{def}\right) \; s.t. \tag{B.4a}$$

$$\hat{\boldsymbol{f}}^k = \boldsymbol{R}_F \boldsymbol{J}(\boldsymbol{q}^k) \cdot \left(\boldsymbol{J}^{\mathsf{T}}(\boldsymbol{q}^k) \cdot \boldsymbol{R}_F \cdot \boldsymbol{J}(\boldsymbol{q}^k) + \boldsymbol{R}_e\right)^{-1} \cdot \overline{\boldsymbol{\tau}}_{mot}^k, \tag{B.4b}$$

$$\overline{\boldsymbol{\tau}}_{mot}^k = \boldsymbol{\tau}_{\text{mot}}^k - \boldsymbol{\tau}_{ff}^k - \boldsymbol{\tau}_{grav,comp}^k - \boldsymbol{\tau}_{fric,comp}^k, \tag{B.4c}$$

$$\boldsymbol{\tau}_{fric,comp,i}^k(\dot{q}_i^k) = \begin{cases} c_{Coulomb,i}^{+} + c_{visc,i}^{+} \cdot \dot{q}_i^k, & \dot{q}_i^k > 0, \\ c_{Coulomb,i}^{-} + c_{visc,i}^{-} \cdot \dot{q}_i^k, & \dot{q}_i^k < 0. \end{cases} \tag{B.4d}$$

with an additional matrix $\boldsymbol{M}$. *By choosing* $\boldsymbol{M} = \text{diag}\,(1,1,1,0,0,0)$, *estimated contact torques do not contribute to the error term in Equation B.4. Thus, the degrees of freedom provided by* $\boldsymbol{R}_e$ *and* $\boldsymbol{R}_F$ *are completely exploited in the optimization to match the estimated contact forces to the force components of* $\boldsymbol{f}_{def}$. *Note, that for* $\boldsymbol{M} = \boldsymbol{I}_6$, *Equation B.4 reduces to the original problem Equation 10.16.*

C Position-Based Skills Used in the Example Assembly Application

The skills presented in this appendix represent basic position-based motions necessary to fulfill the example assembly application presented in Section 12.1. It has to be noted that these skills are only described by a symbolic trajectory representation. This representation omits all scalings and dimensions. The vertical axis of the trajectory represents the wrench dimensions, while the horizontal axis represents the pose dimensions. The shown progression represents the force/position relationship in the predominating motion direction. Red lines indicate borders between trajectory nodes.

Transfer

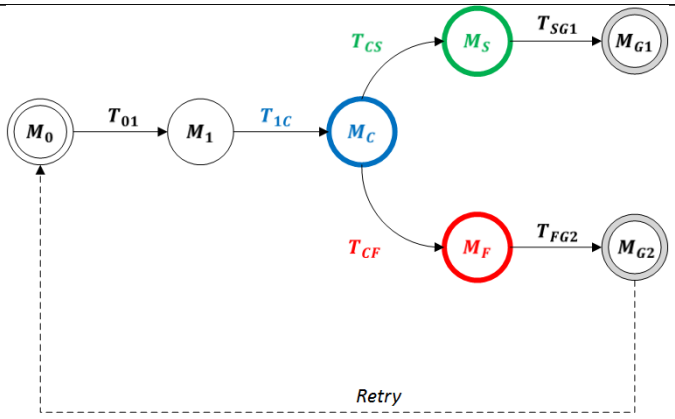
Name	Transfer	
Category	Motion	
Description	Skill to fulfill tasks where a part is transfered between two points on a linear motion path	
Trajectory		
Motion Net		
DOF	Supervised : x, y, z, rx, ry, rz	Controlled : x, y, z, rx, ry, rz
Precondition	Start position reached	
Interruptions	-	
Completion	End position reached	
Quality	End position reached	
Resources	Tools: none	Sensors: position
Skill Dependencies	Subskills: none	Preceeding Skills: none

Table C.1: *Transfer* Data Sheet

PickUp

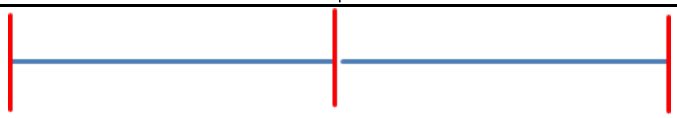
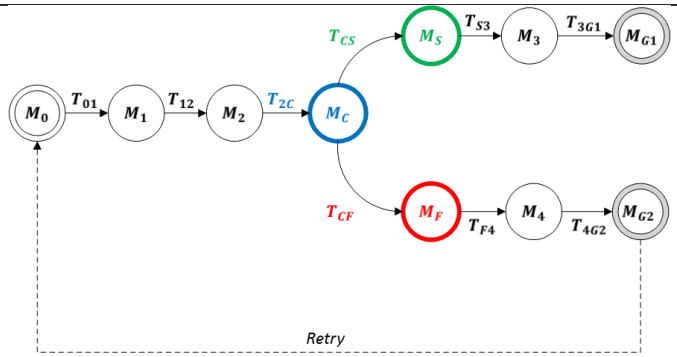
Name	PickUp	
Category	Tool Action	
Description	Skill to fulfill tasks where a part is picked up by a binary controlled tool (on/off)	
Trajectory		
Motion Net		
DOF	Supervised : z	Controlled : z
Precondition	Start position reached	
Interruptions	-	
Completion	Part in gripper	
Quality	Part gripped correctly	
Resources	Tools: none	Sensors: position
Skill Dependencies	Subskills: none	Preceeding Skills: none

Table C.2: *PickUp* Data Sheet

Place

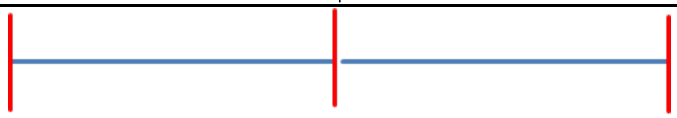
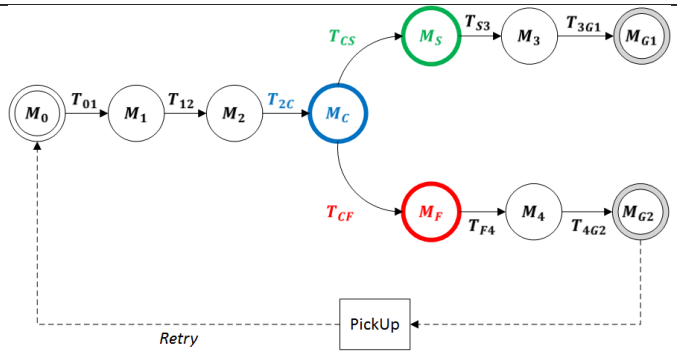
Name	Place	
Category	Tool Action	
Description	Skill to fulfill tasks where a part is placed by a binary controlled tool (on/off)	
Trajectory		
Motion Net		
DOF	Supervised : z	Controlled : z
Precondition	Start position reached	
Interruptions	-	
Completion	Part released from gripper	
Quality	Part placed at correct position	
Resources	Tools: none	Sensors: position
Skill Dependencies	Subskills: none	Preceeding Skills: none

Table C.3: *Place* Data Sheet

Bibliography

[1] Cantilever hook snapfit. http://www.gotstogo.com/misc/engineering_info/snap_design_files/image024.gif. Accessed: 2014-04-18.

[2] Remote center of compliance for peg-in-hole assembly. http://commons.wikimedia.org/wiki/Commons:\Valued_image_candidates/Remote_Center_of_Compliance.svg. Accessed: 2014-01-10.

[3] A. P. Ambler and R. J. Popplestone. Inferring the positions of bodies from specified spatial relationships. *Artificial Intelligence*, 6(2):157–174, 1975.

[4] Y. Bar-Shalom and X.-R. Li. *Multitarget-multisensor tracking : Principles and techniques*. 1995.

[5] S. Bøgh, O. Nielsen, M. Pedersen, V. Krüger, and O. Madsen. *Does your Robot have Skills?* VDE Verlag GMBH, 2012.

[6] L. Bo, X. Ren, and D. Fox. Unsupervised feature learning for rgb-d based object recognition. In *In International Symposium on Experimental Robotics (ISER)*, 2012.

[7] B. Bona and M. Indri. Friction compensation in robotics: an overview. In *IEEE Conference on Decision and Control and European Control Conference*, pages 4360–4367, Seville, Spain, 2005.

[8] P. R. Bonenberger. *The First Snap-Fit Handbook*. Carl Hanser Verlag GmbH & Co. KG, 2005.

[9] L. Brignone and M. Howarth. A geometrically validated approach to autonomous robotic assembly. In *IROS*, pages 1626–1631. IEEE, 2002.

[10] B. Browatzki, V. Tikhanoff, G. Metta, H. Bulthoff, and C. Wallraven. Active object recognition on a humanoid robot. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 2021–2028, 2012.

[11] H. Bruyninckx and J. De Schutter. Specification of force-controlled actions in the task frame formalism - a synthesis. *Robotics and Automation, IEEE Transactions on*, 12(4):581–589, 1996.

[12] C. Carøe, M. Hvilshøj, and C. Schou. Intuitive programming of aimm robot. Master's thesis, Aalborg Universitet, 2012.

[13] J. De Schutter, T. De Laet, J. Rutgeerts, W. Decre, R. Smits, E. Aertbelien, K. Clase, and H. Bruyninckx. Constraint-based task specification and estimation for sensor-based robot systems in the presence of geometric uncertainty. *International Journal of Robotics Research*, 26:433–455, 2007.

[14] J. De Schutter and H. van Brussel. Compliant robot motion ii. a control approach based on external control loops. *Int. J. Rob. Res.*, 7(4):18–33, 1988.

[15] D. F. Dementhon and L. S. Davis. Model-based object pose in 25 lines of code. *Int. J. Comput. Vision*, 15(1-2):123–141, 1995.

[16] R. Desai, J. Xiao, and R. Volz. Contact formations and design constraints: A new basis for the automatic generation of robot programs. In B. Ravani, editor, *CAD Based Programming for Sensory Robots*, volume 50 of *NATO ASI Series*, pages 361–395. Springer Berlin Heidelberg, 1988.

[17] H. Ding and B. Matthias. Manipulation skills for robotic assembly scope & workplan. ABB internal presentation, 2014.

[18] T. Dong, R. Tong, J. Dong, and L. Zhang. Knowledge-based assembly sequence planning system. In *Computer Supported Cooperative Work in Design, 2004. Proceedings. The 8th International Conference on*, volume 2, pages 516–521 Vol.2, May 2004.

[19] D. DuBois. *Fuzzy Sets and Systems: Theory and Applications*. Academic Press, Inc., Orlando, FL, USA, 1997.

[20] K. S. Eom, I. Suh, W. Chung, and S.-R. Oh. Disturbance observer based force control of robot manipulator without force sensor. In *IEEE International Conference on Robotics and Automation*, pages 3012–3017, 1998.

[21] T. L. D. Fazio and D. E. Whitney. Simplified generation of all mechanical assembly sequences. *Robotics and Automation, IEEE Journal of*, 3(6):640–658, 1987.

[22] B. Finkemeyer, T. Kröger, and F. Wahl. The adaptive selection matrix – a key component for sensor-based control of robotic manipulators. In *IEEE International Conference on Robotics and Automation*, pages 3855–3862, 2010.

[23] D. A. Forsyth and J. Ponce. *Computer Vision: A Modern Approach*. Prentice Hall Professional Technical Reference, 2002.

[24] A. Gill. *Introduction to the Theory of Finite State Machines*. McGraw-Hill, New York, 1962.

[25] B. Hannaford and P. Lee. Hidden markov model analysis of force/torque information in telemanipulation. In V. Hayward and O. Khatib, editors, *Experimental Robotics I*, volume 139 of *Lecture Notes in Control and Information Sciences*, pages 135–149. Springer Berlin Heidelberg, 1990.

[26] T. Hasegawa, T. Suehiro, and K. Takase. A model-based manipulation system with skill-based execution. *Robotics and Automation, IEEE Transactions on*, 8(5):535–544, Oct 1992.

[27] H. Hirukawa, Y. Papegay, and T. Matsui. A motion planning algorithm for convex polyhedra in contact under translation and rotation. In *in Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, pages 3020–3027, 1994.

[28] L. Homem de Mello and A. Sanderson. A correct and complete algorithm for the generation of mechanical assembly sequences. *Robotics and Automation, IEEE Transactions on*, 7(2):228–240, Apr 1991.

[29] A. J. Ijspeert, J. Nakanishi, and S. Schaal. Learning attractor landscapes for learning motor primitives. In *in Advances in Neural Information Processing Systems*, pages 1523–1530, 2003.

[30] X. Ji and J. Xiao. Automatic generation of high-level contact state space. In *ICRA*, pages 238–244. IEEE Robotics and Automation Society, 1999.

[31] M. Kaiser, A. Retey, and R. Dillmann. Robot skill acquisition via human demonstration. *7 th International Conference on Advanced Robotics (ICAR '95), Sant Feliu de Guixols, Spain*, 1995.

[32] S. Kaufman, R. Wilson, R. Jones, T. Calton, and A. Ames. The archimedes 2 mechanical assembly planning system. In *Robotics and Automation, 1996. Proceedings., 1996 IEEE International Conference on*, volume 4, pages 3361–3368 vol.4, Apr 1996.

[33] O. Khatib. A unified approach for motion and force control of robot manipulators: The operational space formulation. *Robotics and Automation, IEEE Journal of*, 3(1):43–53, February 1987.

[34] J. Kober and J. Peters. Learning motor primitives for robotics. In *Proceedings of the 2009 IEEE International Conference on Robotics and Automation*, pages 2509–2515, 2009.

[35] S. Kock, T. Vittor, B. Matthias, H. Jerregard, M. Kallman, I. Lundberg, R. Mellander, and M. Hedelind. Robot concept for scalable, flexible assembly automation: A technology study on a harmless dual-armed robot. In *IEEE International Symposium on Assembly and Manufacturing*, pages 1–5, 2011.

[36] D. I. Kosmopoulos. *A Design Framework for Sensor Integration*, pages 1–22. Advanced Robotic Systems International, 2006.

[37] T. Kröger, B. Finkemeyer, U. Thomas, and F. M. Wahl. Compliant motion programming: The task frame formalism revisited. In *Mechatronics and Robotics*, pages 1029–1034, Aachen, Germany, September 2004.

[38] T. Kröger, B. Finkemeyer, and F. M. Wahl. A task frame formalism for practical implementations. In *Robotics and Automation, 2004. Proceedings. ICRA '04. 2004 IEEE International Conference on*, volume 5, pages 5218–5223, April 2004.

[39] C. Laugier. Planning fine motion strategies by reasoning in the contact space. In *Robotics and Automation, 1989. Proceedings., 1989 IEEE International Conference on*, pages 653–659 vol.2, May 1989.

[40] S. Lee. Subassembly identification and evaluation for assembly planning. *IEEE Transactions on Systems, Man, and Cybernetics*, 24(3):493–503, 1994.

[41] T. Lefebvre, J. Xiao, H. Bruyninckx, and G. De Gersem. Active compliant motion: A survey. *Advanced Robotics*, 19(5):479–500, 2005.

[42] J. Leitner, S. Harding, M. Frank, A. Forster, and J. Schmidhuber. Learning spatial object localization from vision on a humanoid robot. *International Journal of Advanced Robotic Systems*, 9, 2012.

[43] M. Linderoth. *On Robotic Work-Space Sensing and Control*. PhD thesis, Lund University, 2013.

[44] M. Linderoth, A. Stolt, A. Robertsson, and R. Johansson. Robotic force estimation using motor torques and modeling of low velocity friction disturbances. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Tokyo, Japan, 2013.

[45] T. Lozano-Pérez. Spatial planning: A configuration space approach. *IEEE Transactions on Computers*, C-32:108–120, 1983.

[46] M. T. Mason. Compliance and force control for computer controlled manipulators. *IEEE Transactions on Systems, Man and Cybernetics*, 11(6):418–432, 1981.

[47] M. T. Mason. *Mechanics of Robotic Manipulation*. MIT Press, Cambridge, MA, August 2001.

[48] H. G. Mayer, I. Nagy, A. Knoll, E. U. Braun, R. Lange, and R. Bauernschmitt. Adaptive control for human-robot skilltransfer: Trajectory planning based on fluid dynamics. In *Robotics and Automation (ICRA), 2007 IEEE International Conference on*, pages 1800–1807, 2007.

[49] B. J. McCarragher, G. Hovland, P. Sikka, P. Aigner, and D. Austin. Hybrid dynamic modeling and control of constrained manipulation systems. *Robotics Automation Magazine, IEEE*, 4(2):27–44, Jun 1997.

[50] H. Mosemann and F. M. Wahl. Automatic decomposition of planned assembly sequences into skill primitives. *IEEE T. Robotics and Automation*, 17(5):709–718, 2001.

[51] T. Nagai and S. Aramaki. The representation method of robotic assembly task with click action. In *Power Electronics, Electrical Drives, Automation and Motion, 2008. SPEEDAM 2008. International Symposium on*, pages 534–538, June 2008.

[52] K. Ohishi, M. Miyazaki, M. Fujita, and Y. Ogino. H infinity observer based force control without force sensor. In *International Conference on Industrial Electronics, Control and Instrumentation*, pages 1049–1054, 1991.

[53] H. Olsson, K. J. Astrom, C. Canuda de Wit, M. Gäfert, and P. Lischinsky. Friction models and friction compensation. *European Journal of Control*, 4(3):176–195, 1998.

[54] F. Pan and J. M. Schimmels. Efficient contact state graph generation for assembly applications. In *Robotics and Automation, 2003. Proceedings. ICRA '03. IEEE International Conference on*, pages 2592–2598. IEEE, 2003.

[55] J. Peters, K. Mülling, J. Kober, D. Nguyen-Tuong, and O. Krömer. Robot skill learning. In L. D. Raedt, C. Bessière, D. Dubois, P. Doherty, P. Frasconi, F. Heintz, and P. J. F. Lucas, editors, *ECAI*, volume 242 of *Frontiers in Artificial Intelligence and Applications*, pages 40–45. IOS Press, 2012.

[56] K. B. Petersen and M. S. Pedersen. *The Matrix Cookbook*. 2012.

[57] T. Reisinger. Robotics & manufacturing group presentation. ABB internal presentation, 2014.

[58] A. Rodriguez, D. Bourne, M. T. Mason, G. F. Rossano, and J. Wang. Failure detection in assembly: Force signature analysis. In *Automation Science and Engineering (CASE), 2010 IEEE Conference on*, pages 210–215. IEEE, 2010.

[59] J. Salisbury. Active stiffness control of a manipulator in cartesian coordinates. In *Decision and Control including the Symposium on Adaptive Processes, 1980 19th IEEE Conference on*, volume 19, pages 95–100, Dec 1980.

[60] C. Samson, B. Espiau, and M. L. Borgne. *Robot Control: The Task Function Approach*. Oxford University Press, 1991.

[61] T. Schulteis. *Automatic Identification of Remote Environments and Calibration of Virtual Models*. Storming Media, 1997.

[62] B. Siciliano and O. Khatib, editors. *Handbook of Robotics*. Springer, 2008.

[63] R. Smits. *Robot Skills: Design of a Constraint-Based Methodology and Software Support*. PhD thesis, Katholieke Universiteit Leuven, 2010.

[64] R. Smits, T. D. Laet, K. Claes, H. Bruyninckx, and J. D. Schutter. itasc: A tool for multi-sensor integration in robot manipulation. In H. K. H. Hahn and S. Lee, editors, *Multisensor Fusion and Integration for Intelligent Systems*, volume 35 of *Lecture Notes in Electrical Engineering*, pages 235–254. Springer, 2009.

[65] M. Stenmark and A. Stolt. A system for high-level task specification using complex sensor-based skills, 2013. Conference Abstract.

[66] A. Stolt, M. Linderoth, A. Robertsson, and R. Johansson. Force controlled robotic assembly without a force sensor. In *IEEE International Conference on Robotics and Automation*, pages 1538–1543, Minnesota, USA, 2012.

[67] L. D. Stone, T. L. Corwin, and C. A. Barlow. *Bayesian Multiple Target Tracking*. Artech House, Inc., Norwood, MA, USA, 1st edition, 1999.

[68] R. Sturges and S. Laowattana. Fine motion planning through constraint network analysis. In *Assembly and Task Planning, 1995. Proceedings., IEEE International Symposium on*, pages 160–170, Aug 1995.

[69] R. C. W. Sung, R. Chun, and W. Sung. Automatic assembly feature recognition and disassembly sequence generation. Technical report, 2001.

[70] U. Thomas, M. Barrenscheen, and F. Wahl. Efficient assembly sequence planning using stereographical projections of c-space obstacles. In *Assembly and Task Planning, 2003. Proceedings of the IEEE International Symposium on*, pages 96–102, July 2003.

[71] U. Thomas, B. Finkemeyer, T. Kröger, and F. M. Wahl. Error-tolerant execution of complex robot tasks based on skill primitives. In *Robotics and Automation, 2003. Proceedings. ICRA '03. IEEE International Conference on*, pages 3069–3075, 2003.

[72] U. Thomas, G. Hirzinger, B. Rumpe, and C. Schulze. A new skill based robot programming language using uml/p statecharts. In *IEEE International Conference on Robotics and Automation*, 2013.

[73] U. Thomas and F. Wahl. A system for automatic planning, evaluation and execution of assembly sequences for industrial robots. In *Intelligent Robots and Systems, 2001. Proceedings. 2001 IEEE/RSJ International Conference on*, volume 3, pages 1458–1464, 2001.

[74] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005.

[75] L. Villani, C. Canudas de Wit, and B. Brogliato. An exponentially stable adaptive control for force and position tracking of robot manipulators. *Automatic Control, IEEE Transactions on*, 44(4):798–802, Apr 1999.

[76] A. Wahrburg. Contact force estimation for robotic assembly using motor torques - summary slide. ABB internal presentation, 2014.

[77] A. Wahrburg. Skill pictograms and trajectories. ABB internal presentation, 2014.

[78] A. Wahrburg. Snap-fit skill. ABB internal presentation, 2014.

[79] A. Wahrburg, S. Zeiss, B. Matthias, and H. Ding. Contact force estimation for robotic assembly using motor torques. In *IEEE International Conference on Automation Science and Engineering*, 2014.

[80] D. Whitney. Force feedback control of manipulator fine motions. *Journal of Dynamic Systems, Measurement, and Control*, 99:91–97, 1977.

[81] D. E. Whitney. Quasi-static assembly of compliantly supported rigid parts. *Journal of Dynamic Systems, Measurement, and Control*, 104(1):65–77, Mar. 1982.

[82] J. Wolter. On the automatic generation of assembly plans. In *Robotics and Automation, 1989. Proceedings., 1989 IEEE International Conference on*, pages 62–68 vol.1, May 1989.

[83] Y. Xia, Y. Yin, and Z. Chen. Dynamic analysis for peg-in-hole assembly with contact deformation. *The International Journal of Advanced Manufacturing Technology*, 30(1-2):118–128, 2006.

[84] J. Xiao. Automatic determination of topological contacts in the presence of sensing uncertainties. In *Robotics and Automation, 1993. Proceedings., 1993 IEEE International Conference on*, pages 65–70. IEEE Computer Society Press, 1993.

[85] J. Xiao and R. A. Volz. On replanning for assembly tasks using robots in the presence of uncertainties. In *Proc. of the 1989 IEEE International Conference on Robotics and Automation (Vol. 2)*, pages 638–645, Scottsdale, AZ, 1989.

[86] Z.-P. Yin, H. Ding, H.-X. Li, and Y.-L. Xiong. A connector-based hierarchical approach to assembly sequence planning for mechanical assemblies. *Computer-Aided Design*, 35(1):37–56, 2003.

[87] T. Yoshikawa, T. Sugie, and M. Tanaka. Dynamic hybrid position/force control of robot manipulators-controller design and experiment. *Robotics and Automation, IEEE Journal of*, 4(6):699–705, Dec 1988.