CRANFIELD UNIVERSITY

MINH HOANG TO

A Framework for Flexible Integration in Robotics and its Applications for Calibration and Error Compensation

SCHOOL OF ENGINEERING

PhD THESIS
Academic Year: 2009 - 2012

Supervisor: Prof. Phil Webb
June 2012

CRANFIELD UNIVERSITY

SCHOOL OF ENGINEERING

PhD THESIS

Academic Year 2009 – 2012

MINH HOANG TO

A Framework for Flexible Integration in Robotics and its Applications for Calibration and Error Compensation

Supervisor: Prof. Phil Webb

June 2012

This thesis is submitted in partial fulfilment of the requirements for the degree of PhD

ABSTRACT

Robotics has been considered as a viable automation solution for the aerospace industry to address manufacturing cost. Many of the existing robot systems augmented with guidance from a large volume metrology system have proved to meet the high dimensional accuracy requirements in aero-structure assembly. However, they have been mainly deployed as costly and dedicated systems, which might not be ideal for aerospace manufacturing having low production rate and long cycle time. The work described in this thesis is to provide technical solutions to improve the flexibility and cost-efficiency of such metrology-integrated robot systems.

To address the flexibility, a software framework that supports reconfigurable system integration is developed. The framework provides a design methodology to compose distributed software components which can be integrated dynamically at runtime. This provides the potential for the automation devices (robots, metrology, actuators etc.) controlled by these software components to be assembled on demand for various assembly applications.

To reduce the cost of deployment, this thesis proposes a two-stage error compensation scheme for industrial robots that requires only intermittent metrology input, thus allowing for one expensive metrology system to be used by a number of robots. Robot calibration is employed in the first stage to reduce the majority of robot inaccuracy then the metrology will correct the residual errors. In this work, a new calibration model for serial robots having a parallelogram linkage is developed that takes into account both geometric errors and joint deflections induced by link masses and weight of the end-effectors.

Experiments are conducted to evaluate the two pieces of work presented above. The proposed framework is adopted to create a distributed control system that implements calibration and error compensation for a large industrial robot having a parallelogram linkage. The control system is formed by hot-plugging the control applications of the robot and metrology used together. Experimental results show that the developed error model was able to improve the 3σ positional accuracy of the loaded robot from several millimetres to less than one millimetre and reduce half of the time previously required to correct the errors by using only the metrology. The experiments also demonstrate the capability of sharing one metrology system to more than one robot.

Keywords:

Airframe Assembly Automation, Metrology-Integrated Robots, Plug and Produce, Robot Accuracy, Kinematic Identification

ACKNOWLEDGEMENTS

First of all, I would like to thank my supervisor Prof. Phil Webb for giving me a wonderful opportunity to study in the United Kingdom. Without his guidance and patience during my slow writing-up, this thesis would never have been completed.

I would also thank Dr. Chen Ye, former researcher at the University of Nottingham, for his contributions at the early stage of this research and Dr. Amir Kayani, lead manufacturing engineer at Airbus UK, for his valuable comments.

I would like to thank all the people at Cranfield University who helped and supported me during my PhD study, especially Mr. John Thrower who has been a great friend and colleague to me. Special thanks would also be sent to Mrs. Rachael Wiseman for helping me submit this thesis.

Finally, I would thank my dearest wife and parents in Vietnam for their support during the last three years.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ACE	Adaptive Communication Environment
ADM	Absolute Distance Meter
API	Application Programming Interface
CAD	Computer Aided Design
CCD	Charged Couple Device
CCR	Concurrency and Coordinate Runtime
CMM	Coordinate Measuring Machine
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
DCOM	Distributed Component Object Model
DH	Denavit - Hartenberg
DLL	Dynamic Link Library
DOF	Degree of Freedom
DSS	Decentralized Software Services
F/T	Force/Torque
FIFO	First In First Out
I/O	Input / Output
IDL	Interface Definition Language
IFM	Interferometer
IP	Internet Protocol
HTTP	Hypertext Transfer Protocol
MiRPA	Middleware for Robotic and Process Automation
OLP	Offline Programming
ORB	Object Request Broker
ORiN	Open Robot / Resource interface for the Network
OROCOS	Open Robot Control Software
OS	Operating System
PCI	Peripheral Component Interconnect
RDS	Robotics Developer Studio
RPC	Remote Procedure Call
RTAI	Real-time Application Interface for Linux
PnP	Plug and Produce

SOAP	Simple Object Access Protocol
SDK	Software Development Kit
SMR	Spherical Mounted Reflector
TAO	The ACE Orb
TMAC	Tracker-Machine control sensor
TCP	Tool Centre Point
TCP	Transmission Control Protocol (as in TCP/IP)
UDP	User Datagram Protocol
USB	Universal Serial Bus
XML	eXtensible Markup Language

1 INTRODUCTION

1.1 Context

The work presented in this thesis is based within the field of manufacturing engineering, specifically the field of robotics. This PhD was undertaken as part of a completed research project sponsored by Airbus UK and the Engineering and Physical Science Research Council (EPSRC) toward the wide adoption of automation in the aerospace industry by the Aero-structure Assembly and Systems Installation Research Group, located at Cranfield University.

1.2 Motivation

Traditionally, airframe assembly is labour-intensive. Mark Summers, Head of Manufacturing Engineering Research, Airbus UK stated that nearly half of 50 millions of holes drilled per year during manufacturing and assembly processes of Airbus wings are done manually (Warwick, 2007). Ideally, more of these processes should be carried out using automation, but this presents many difficulties due to the physical size and shape of the parts involved and the accuracy of the alignments required. To meet these challenges, current automation solutions in airframe assembly must rely on very large machine tools, often gantry-mounted drilling systems and monolithic jigs. This is referred to as the fixed automation solution since these bespoke machines and tooling are dedicated for only a specific aircraft model and product type, very expensive and require a long lead time for design and manufacture.

A solution to the flexibility and cost issues in airframe assembly automation is possible through the use of robotics. Industrial robots have been well-established in the automotive industry and their technological maturity has earned them their way into aerospace applications (Jamshidi *et.al.*, 2010). Unlike custom-designed gantry systems and other monumental pieces of fixed automation, industrial robots are mass-produced and are highly flexible. Designed to be versatile manipulators, they are considered viable for many repetitive processes such as drilling, fastening, composite layup as well as

painting and coating in the aerospace industry (Webber, 2007). In addition, these relatively light weight structures do not require special foundations and hence, can be relocated throughout the shop floor to meet new production needs. They can be put on a mobile platform that augments their reach to service a large aero-structure, and yet are small and dexterous enough to perform operations in confined spaces within a wing box. The flexibility is also achieved through their ease of programming: industrial robots are usually enhanced with programming languages, offline programming (OLP) and simulation software. These technologies help shorten the lead time by letting manufacturers design the work-cell, tools, manufacturability before the actual production is put into place, thus providing the robots with the possibility to be deployed quickly for various assembly processes.

One of the main reasons for delays in the widespread use of robots in the aerospace industry has been their insufficient accuracy. Compared to the proven gantry systems, off-the-shelf industrial robots possess much lower accuracy due to their articulated structures. The accuracy is further degraded by structural deflections induced by applied forces and loads during an assembly process, resulting in significant errors of the end-effectors. The accumulated errors are often in the order of several millimetres, far beyond the tolerance allowances for aerospace applications, which are in many cases selected as ±0.2mm (Kihlman, 2005). To meet such high demand for accuracy, current error compensation technique for the robots is through metrology guidance, in which an external measurement system is utilized to track the location of the robot end-effector continuously and send this data via a feedback loop to the robot controller. Based on the feedback data, the robot position is adjusted iteratively until a satisfactory level of deviation from its nominal position is met. It has been reported that such a metrology-guided robot system is able to deliver the positional accuracy greater than 0.1mm over the working volume of several metres (Calder, 2011). The system can also be combined with other optical sensors and localisation techniques to compensate for the dimensional and positional variations of the part due to temperature fluctuations and

misalignments, thus reducing the reliance on monolithic jigs and manual setup process previously required to fix the part accurately in space.

Despite the important advancements in addressing the accuracy, there are a number of issues related to the cost and flexibility of the metrology-integrated robotics solution (Morey, 2007). The cost issue is firstly due to the expense for the external metrology system that positions the robot. Such a high-end large measuring volume instrument (e.g., laser tracker, photogrammetry, Indoor GPS) might cost hundreds of thousand pounds, several times more expensive than the robot alone (Saadat *et.al.*, 2002). A full-scale assembly cell might consist of multiple robot and metrology systems and hence, requires intensive capital investments. This has motivated the search for an economically-feasible solution undertaken in this thesis that allows for several robots to share one metrology system to reduce the cost of investment.

The problem of flexibility is inherited from those of the support technologies that facilitate the robotics solution, such as end-effectors, tooling and most of all, system integration. The abovementioned robot assembly work-cell with enabling sensory capabilities requires physical and functional integration of multi-vendor equipment (e.g., robots, actuators, metrology etc.). Typically, this leads to the development of a dedicated control system (largely implemented in software) that links these incompatible devices and performs correction activities for the robots and the part. This kind of control architecture, which is explored in this thesis, is application-specific and thus, might not be flexible enough for the aerospace industry. Unlike the automotive industry where the production rates are so high that it allows robots to perform a single task repeatedly, aerospace manufacturing has to deal with much lower production volumes, longer cycle times and thus, demands robots that can perform more than one function (Webber, 2007; Lott, 2011). One approach toward this direction is a “Plug and Produce” manufacturing work-cell that is able to alter its original setups with varying number of machines/robots whilst the robots themselves are reconfigurable with various end-effectors to accommodate different products and operations (Minhas *et.al.*, 2011). Under these

circumstances, a cell control system with strongly-coupled components and hard-coded control logic dedicated for a specific application will cause huge production downtime and cost for software modification. This shortcoming has resulted in the consideration for a more flexible software environment that allows fast and easy changeover, eventually making robotics a good fit in terms of cost and flexibility for airframe assembly. This thesis is attempting to provide a stepping stone towards achieving this goal.

1.3 Research Objectives

The primary aim of the work described within this thesis is to improve the flexibility and cost-efficiency of robotics in airframe assembly. This has led to two equally important research topics presented in this thesis:

- To improve the flexibility, the thesis proposes a software framework that supports reconfigurable integration of automation equipment in a robot work-cell. The framework provides a design template to develop modular, distributed software components, namely services, each of which controls one or a subset of the above hardware components. These services can be hot-plugged at runtime to form a control system on demand for the equipment commissioning, making it possible to assemble the work-cell dynamically for various assembly applications.

- To reduce the cost of investment, this thesis proposes a two-stage error correction scheme that promotes the use of one expensive piece of large volume metrology for several robots. For each robot position, the accuracy of the robot is firstly improved by an error compensation model which narrows down drastically the error band of the robot from several millimetres to sub-millimetre then the residual errors can be corrected by the metrology system. Since the metrology system does not have to be operational the whole time, it is able to support a number of robots. When one robot has been accurately positioned and, for example, starts machining, the metrology system can be deployed for the others.

The error compensation model in the first stage is obtained from a robot calibration procedure. The idea of utilizing kinematic calibration as a cost-saving method to improve the robot accuracy is not new. Indeed, the technique has been intensively studied in literature and well-established for standard serial (e.g. the elbow type) robot manipulators. Nevertheless, there has been no simple calibration model for serial robots having a parallelogram linkage due to the complication in modelling error propagation in the linkage. In this thesis, a simple yet accurate error model for this type of robots will be introduced that takes into account not only kinematic errors of the robot but structural deflections induced by its link masses and applied load (e.g., the weight of the end-effector).

The research objectives of this thesis are as follows:

a. Develop and implement the software framework:

- Identify the technologies that enable the development of distributed control systems for the purpose of flexible integration, including their advantages and limitations.
- Develop the proposed framework.
- Implement the framework.

b. Error modelling and compensation for industrial robots:

- Review error compensation techniques for industrial robots including robot calibration for elbow-type manipulators.
- Develop an efficient calibration model for the parallelogram linkage-type manipulators.
- Develop the aforementioned two-stage error compensation algorithm for representative robot and metrology systems.

c. Adopt the proposed software framework to develop a control system that implements the calibration and error compensation processes:

- Demonstrate the application of the framework for the flexible integration of the robot and metrology systems used.

- Validate the implemented error modelling and compensation strategy and the possibility of utilizing one metrology system for multiple robots.

1.4 Thesis overview

Chapter Two – BACKGROUND

This chapter provides background information pertaining to the main research areas of the thesis, including industrial robots, metrology, error compensation and system integration for a robot work-cell.

Chapter Three – LITERATURE

This chapter provides a review of relevant research including error compensation techniques employed to improve the accuracy of robotic assembly processes and system integration in robotics, particularly distributed control frameworks/systems. The chapter also addresses the implications and remaining issues of these works that the research will attempt to address.

Chapter Four – METHODOLOGY

This chapter presents the methodology used to achieve the objectives of the research. The chapter firstly outlines desirable features of the proposed application framework then describes the approach to these features. Next, details on the robot calibration technique for the elbow type manipulators are discussed. Remaining challenges and the author's approach to the calibration of the parallelogram linkage type manipulators are described.

Chapter Five – DEVELOPMENT OF THE FRAMEWORK

This chapter presents in detail the architecture of the proposed application framework. Assessment of the performance of the framework is also provided.

Chapter Six - CALIBRATION AND ERROR COMPENSATION FOR SERIAL ROBOTS HAVING A PARALELLOGRAM LINKAGE

This chapter presents the theoretical work on calibration and error compensation developed for parallelogram linkage type manipulators, taking into account kinematic and compliance (deflection) errors.

Chapter Seven – EXPERIMENTAL SETUP

This chapter describes the experimental setup developed using the proposed framework given in Chapter 5. The system will be used to implement the robot calibration and error compensation presented in Chapter 6.

Chapter Eight – SIMULATION, EXPERIMENT RESULTS AND ANALYSIS

This chapter presents results and analysis of the simulation and experiments undertaken to validate the works presented in previous chapters.

Chapter Nine – CONCLUSSIONS

This chapter concludes the work and gives some suggestions for future work.

2 BACKGROUND

This chapter provides the reader essential background information pertaining to the main research areas in this thesis. Firstly, section 2.1 will briefly introduce aircraft assembly processes and their current automation solutions. The subsequent sections will discuss in more detail the technologies enabling flexible automation in the area, i.e., industrial robots, offline programming, metrology as well as system integration of these manufacturing resources.

2.1 Aircraft Assembly

2.1.1 Basic aircraft structure

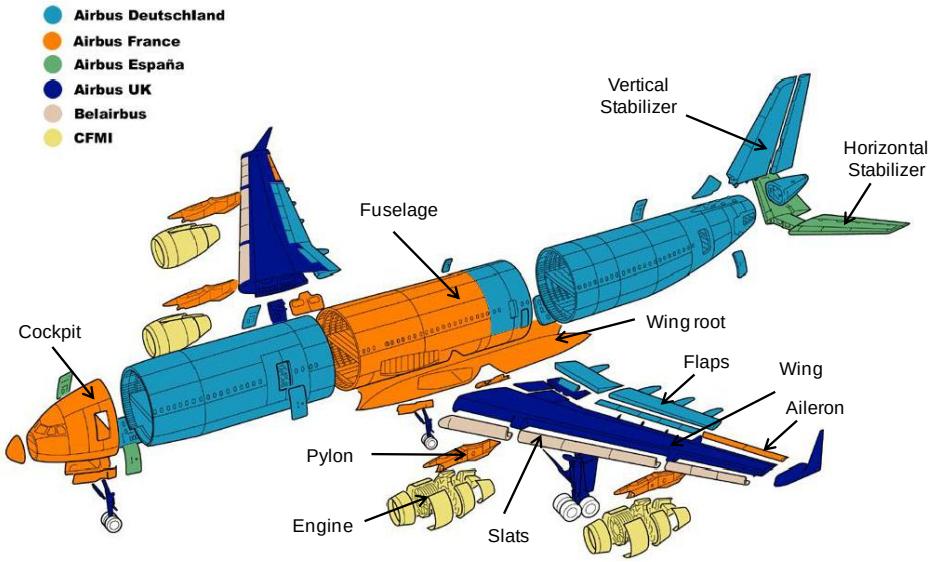


Figure 2-1 Basic aircraft structure (Kayani *et.al.*, 2008)

An airplane is a transportation device designed for carrying people and cargo from one place to another. Figure 2-1 depicts its basic components, many of which are produced on different sites and are brought together for final assembly. Their functionalities are described briefly as follows:

- Wings: to generate most of the lift that holds the airplane in the air,
- Turbine engines: to generate the thrust that pushes the airplane forward through the air,

- Stabilizers: to provide stability to the airplane, i.e., to control its orientation. The yaw and pitch of the aircraft are controlled by the vertical and horizontal stabilizer respectively whereas the roll of the aircraft is manoeuvred by the ailerons located at the wings,
- Fuselages: the main body of the airplane – for carrying passengers and cargo,
- Cockpit: contains the control centre of the airplane and pilot seats.

A more detailed construction of the aircraft wing is illustrated in Figure 2-2. A wing box is made up of spars, ribs, stringers and skin panels. The spars and ribs are skeleton structures of the wing, providing it with the longitudinal and lateral stiffness whereas the stringers are for strengthening the skin panels with thousands of rivets and bolts.

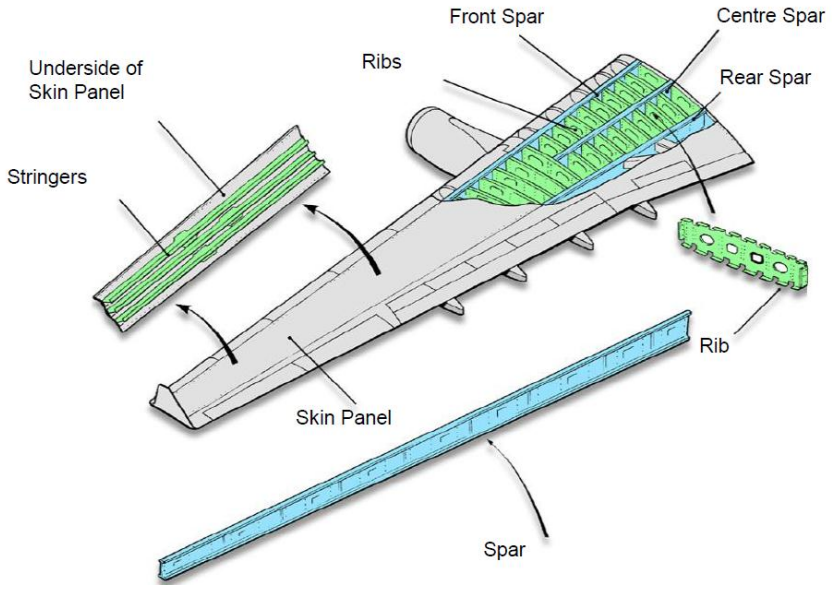


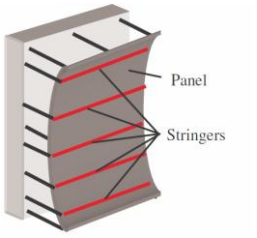
Figure 2-2 Components of a wing box (Kayani *et.al.*, 2008)

2.1.2 Assembly process

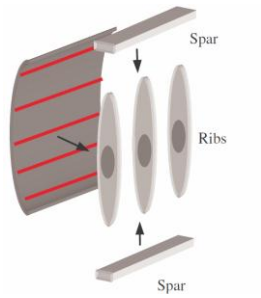
In aircraft assembly, airframe parts and substructures are joined together to form product families, such as wings, fuselages, cockpit etc. that eventually become the complete aircraft. As opposed to automotive assembly, in which the parts are mostly welded together, the joining elements used in aircraft parts

are rivets and thus, the assembly process is carried out by drilling holes, followed by fastening (Rooks, 2001). Airframe assembly sequence has been classified into four levels (Kihlman, 2005):

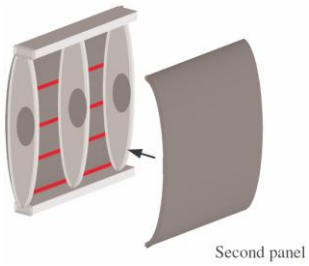
- At the first level, stringers and stiffeners (frame) are attached to the skin panel. A large number of holes are drilled, followed by fastener installation. Tooling (fixtures and jigs) is used to position and hold the structure from one side whereas the other side is open for automated drilling and fastening machines.



- At the second level, the skeleton structures are jointed and combined with the first panel from at least two directions. While the Level 1 Assemblies are part of the Level 2 Assemblies, the latter has lower throughput and hence is suited for industrial robots for part loading and drilling.



- At the third level, the structure from Level 2 is closed with an additional panel. This panel is usually pre-fixed manually and the remaining drilling will be carried out automatically. After this process, the entire structure is disassembled, de-burred, cleaned from the inside and re-assembled again by fastening.



- At the final level, the substructures from Level 3, such as wing boxes, fuselages, cockpit etc., are brought together to form the aircraft. The substructures are lifted by crane or gantry and positioned accurately to each other. This assembly has a high level of human activity due to lack of innovation and development effort to find flexible and adaptable alternatives (Figure 2-3).

Figure 2-3 Assembly levels (Kihlman, 2005)

2.1.3 Automation in aircraft assembly

To guarantee the high demand for accuracy and large volume coverage, current automation solutions for airframe assembly rely on large gantry type machines. One example of such machine is the E4380 produced by ElectroImpact for drilling and riveting upper and lower surfaces of the A380 wing panels (Figure 2-4). The work volume of the machine is 4m in vertical, 1.67m in horizontal and 175m in length, designed to access the whole surface of the wing panel. The machine utilizes a yoke arm articulated in five axes to position process tools for drilling and fastening operations. The accuracy of the machine is guaranteed by its sturdy structure, weighing 160 tons, combined with a linear glass scale as the secondary encoder on each servo axis and other sensors for adjusting the normality of the tools with wing panel surfaces (Zieve *et.al.*, 2004). These large machines are typically employed at the aforementioned Level 1 Assembly (Kihlman, 2005).

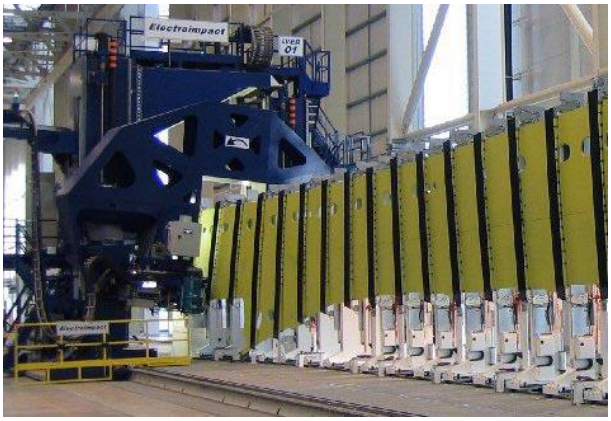


Figure 2-4 The E4380 auto-riveting machine (ElectroImpact)

Despite the extreme accuracy and stiffness, these custom-designed machines exhibit a number of disadvantages (Kihlman, 2005; Webb *et.al.*, 2005):

- Inflexibility: these machines are dedicated for a specific product. Change to another product that the machine was not designed for has proven a difficulty. They might become obsolete and stand still like monuments,
- High investment cost: these machines cost between £2.5 million to £3 million,

- High maintenance cost: they require hardened foundations and take up floor space.

The aerospace industry, where the above large scale machines have been the common method for automation, is now striving to reduce costs and shorten lead times. Industrial robots have been considered promising alternatives to these capital intensive machines to reduce the manufacturing costs and increase flexibility. However, compared to these machines, robots possess much lower accuracy and stiffness and thus, they must have improved accuracy which can be delivered through some form of error compensation from metrology systems. Issues relating to this metrology integrated robotic approach will be presented in the subsequent sections.

2.2 Industrial robots

2.2.1 Mechanical structures

A robot is an automatically controlled, reprogrammable multifunctional mechanical system (Spong *et.al*, 2004). The mechanical structure of a robot manipulator is composed of a sequence of *links* connected by *joints*. The joints typically are rotary (revolute) or linear (prismatic), allowing relative rotation or translation between adjacent links. Depending on the how the links and joints are interconnected, the types of robot manipulators are categorized into *open kinematic chain* or *closed kinematic chain* (Siciliano *et.al*., 2007). In the former, the links are serially connected via joints to form a single open chain. One end of the chain is fixed and is called the *base* while the other end is freely moveable. An *end-effector* is attached to this end, allowing the robot to perform some interactions with its environment. In the latter, the end-effector is connected to the base via several links working in parallel and hence, the robot contains several closed loops. These two types of robots are well known as serial and parallel robots (Figure 2-5).

Serial robots are the most commonly used in industry. According to the report of the International Federation of Robotics, up to 2005, more than 99% of installed

industrial robots worldwide are serial manipulators (Siciliano *et.al.*, 2007). Though parallel kinematic machines have the advantages of higher structural stiffness and accuracy over serial ones, their applications are limited due to the relatively small work envelopes, difficult access to complex structures and high costs and thus, they are mostly employed as 6-axis machine tools rather than versatile manipulators (Dombre *et.al.*, 2007; Angeles, 2003). In addition, the accuracy of these machines is not sufficient for airframe assembly and would require positional correction from a metrology system anyway (Kihlman, 2005). The following sections will mainly discuss about serial manipulators.

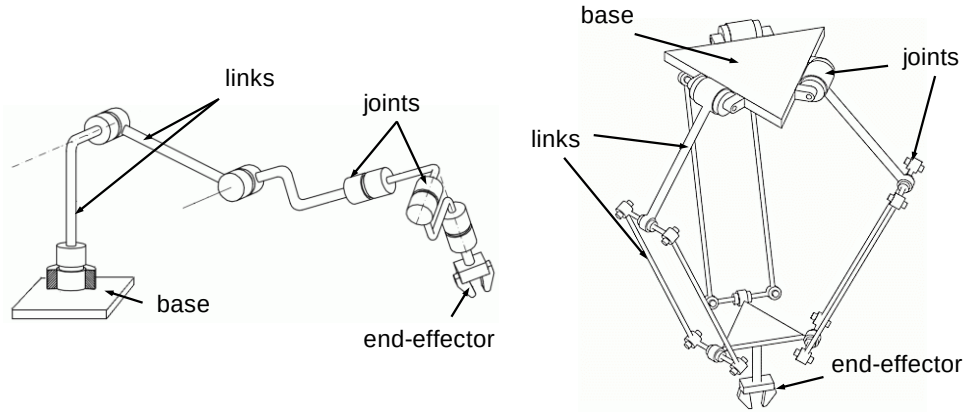


Figure 2-5 Two types of robots: serial (left) and parallel (right) robots (Angeles, 2003)

Serial manipulators may have several kinematic configurations, classified on the basis of the first three joints whether they are prismatic (P) or revolute (R) joints (Spong *et.al.*, 2004). Among them, the most popular is the *articulated* structure, which employs 6R joints: the first three are for positioning the arm in space and the last three are for providing the wrist full orientation. In addition, the axis of the first joint which rotates the whole structure in a horizontal plane is perpendicular to the axes of the second and the third joints which elevate the upper-arm (the second link) and the forearm (the third link) in a vertical plane. The joints are typically actuated by electric motors.

Articulated manipulators are further categorized into the elbow and the parallelogram linkage types (Figure 2-6). The notable difference between the elbow and parallelogram linkage manipulators is the mounting location of the

motor that drives the forearm. In the former, the motor is mounted directly on the upper arm whereas in the latter, it is mounted on the first link and drives the forearm via a parallelogram mechanism. Joint 2 of the robot of this type has two co-axial motors and hence, is referred to as “double revolute joint” in Figure 2-6 whereas other joints of the parallelogram linkage are un-actuated. By resorting to the closed kinematic chain of parallelogram type, this hybrid design offers advantages of higher stiffness and loading capacity over the purely open kinematic chain while still maintaining a large working volume (Siciliano *et.al.*, 2007).

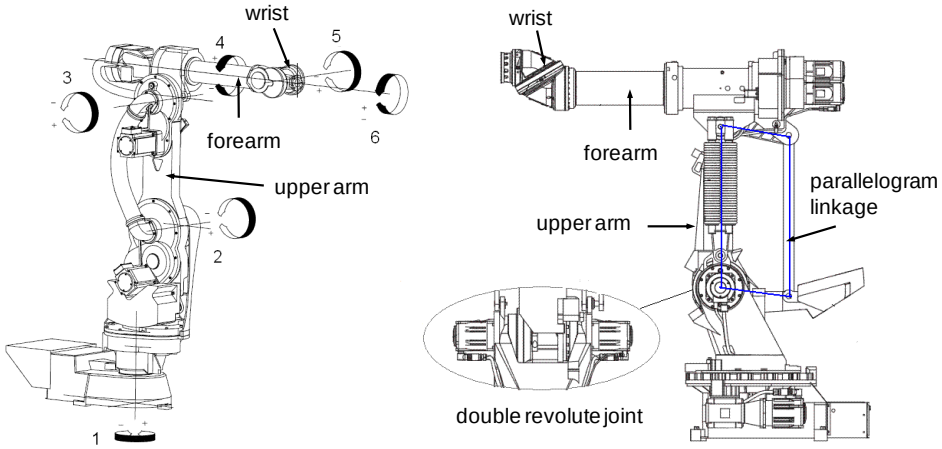


Figure 2-6 Articulated manipulators: the elbow type (left) and the parallelogram linkage type (right)

An articulated manipulator usually possesses at least six independent degrees of freedom (DOF): three for position and three for orientation. In this work, the combination of position and orientation of the robot end-effector is sometimes referred as location or pose for brevity. In order to specify the end-effector location, a coordinate system or frame, is rigidly attached to it. The location of this mobile frame, referred to as the Tool Centre Point (TCP), is then calculated with respect to a fixed reference frame, which could either be the Robot base frame, the World frame of the environment or the User frame attached to the part on which the robot is working (Figure 2-7). Since the robot mobilizes its end-effector by actuating the joints, there are necessary *kinematic models* to establish the relationships between the location of the end-effector and positions of the joints. They are:

- *forward kinematics* : to find the location of the end-effector relative to the base, given the positions of all the joints,
- *inverse kinematics:* to find the positions of the joints, given the targeted location of the end-effector relative to the base.

These kinematic models are mathematical formulations of which the variables are the joint positions and the constants are Denavit-Hartenberg (DH) parameters (Spong *et.al*., 2004) that describe geometry of the robot, such as link lengths and angles between joint axes. Further representation of the robot kinematics will be described in Appendix A.

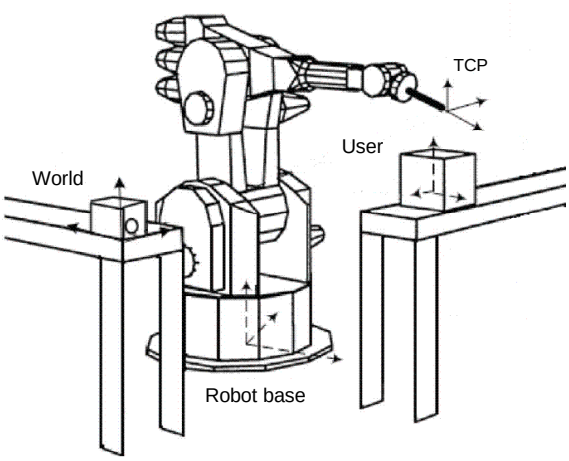


Figure 2-7 Types of reference frames

The performance characteristics of a robot manipulator are specified by several factors, including the two most important, position accuracy and position repeatability (Figure 2-8):

- *Accuracy*: the ability of the robot to position its end-effector to a programmed position in space. It is measured as the difference between the commanded position and the mean of attained positions when the robot visits the commanded position several times from different initial positions (ISO 9283, 1998).
- *Repeatability*: the ability of the robot to repeatedly return to the same position. It is measured as the distance between the mean of the attained positions and the furthermost attained position (ISO 9283, 1998).

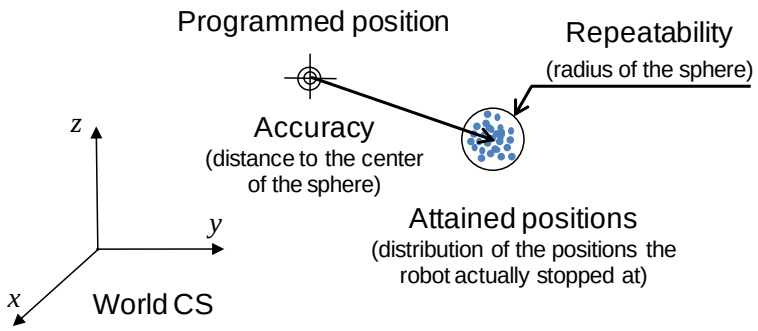


Figure 2-8 Robot accuracy and repeatability (Khalil *et.al.*, 2004)

In addition to the repeatability and accuracy, other measures of robot performances are:

- *Workspace*: the total volume swept by the end-effector as the robot executes all possible motions. It is determined by the limits of the joints and links employed by the robot mechanical structure.
- *Payload*: maximum load the robot can carry.
- *Resolution*: the smallest increment of motion that can be achieved by the joint or the end-effector.

2.2.2 Control architecture

An industrial manipulator is not just a mechanical structure. Indeed, it is a complex system, including:

- the mechanism, constituted by the links and joints described thus far;
- *actuators* (servomotors) that transmit their motion to the joints using some suitable transmission systems;
- joint *sensors* (encoders or resolvers) that measure the joint positions;
- *controller*, which realizes motion of the manipulator by generating input signals to the actuators and receiving feedback signals from the sensors through closed loop control techniques;
- *work-cell* and *peripheral devices*, which constitute the environment in which the robot works. The robot's end-effector is a type of peripheral device since it is not originally equipped by robot manufacturers;

- *communication interface*, including the operator interface through which the user plans the tasks of the robot and the periphery interface through which the robot communicates with external systems (Khalil *et.al*., 2004).

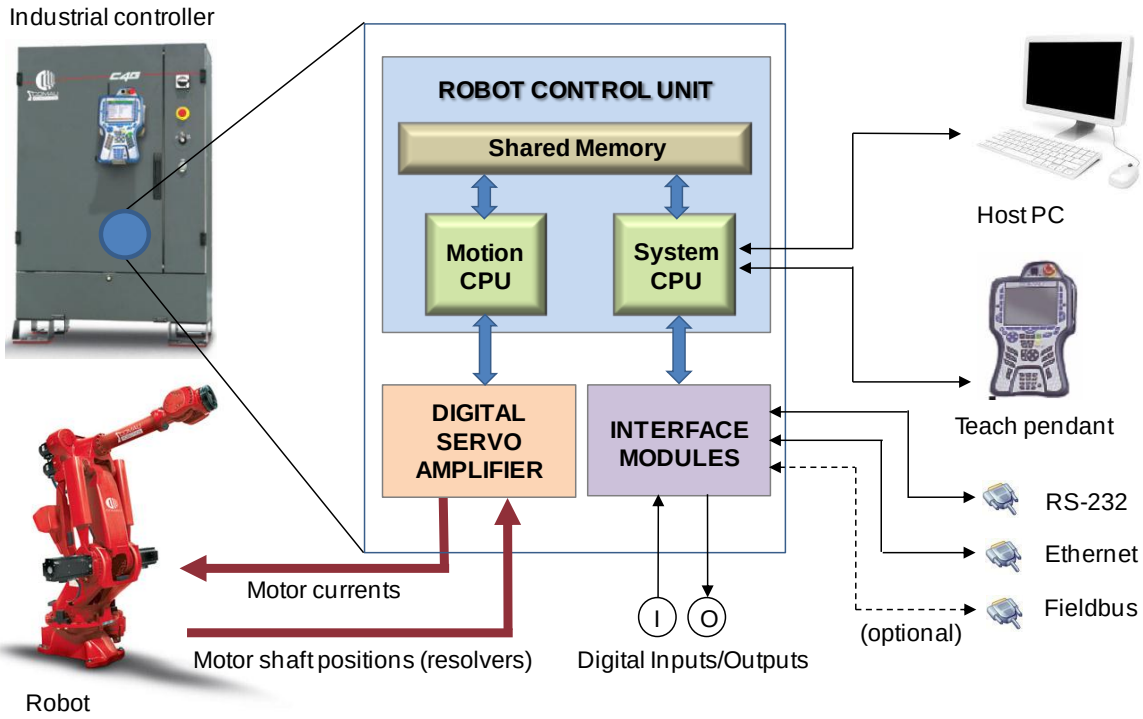


Figure 2-9 Components of a robot system (Comau Robotics)

The hardware structure of a typical robot system is depicted in Figure 2-9. At the heart of the robot industrial controller is the control unit having multiple microprocessors: one (the System CPU) for overall system control and the other (the Motion CPU) specialized for robot motion control. Specifically, the System CPU is responsible for the management of:

- the operator interface, via a handheld device named *teach pendant* and a computer (PC). The teach pendant is a joystick allowing the user to jog the robot to a location and record its relevant coordinates.
- the periphery interface to external systems, e.g., sensors and actuators of the end-effector, by using digital signal processing (DSP) electronic circuits, namely the Interface Modules. Most of robot industrial controllers provide parallel digital inputs/outputs (I/O), serial communications (RS-232/485) and Ethernet to communicate with these auxiliary devices. Other industrial protocols such as the fieldbuses (e.g., DeviceNet,

Profibus-DP, Interbus etc.) might also be supported via optional plug-in boards.

By using the PC, the user is able to assign the tasks that the robot system executes. This action is referred to as *task planning*, where the user specifies the desired end-effector locations and the type of motion of the robot when travelling between these locations, which can be either path free (known as *point-to-point* motion) or a continuous path, e.g., a line or an arc. All industrial robots are provided with high level *robot programming languages* for intuitive task planning. Generated robot programs are downloaded from the PC to the controller, interpreted into machine codes by the System CPU then transferred to the Shared Memory of the Motion CPU via an internal bus.

The Motion CPU is in charge of *trajectory generation* and joint position *servo control* as follows.

- Trajectory generation: given the programmed locations of the end-effector in Cartesian space and its travelling path, the CPU calculates a time sequence of corresponding variables (position, velocity and acceleration) in joint space. At first, the CPU performs the inverse kinematics to transform the coordinates from Cartesian to joint space. Second, from the start and end locations of a path segment in joints, trajectory points are interpolated at a certain rate, typically at 250 Hz in modern controllers, serving as reference inputs to the servo control.
- Servo control: on the basis of the given motion trajectory, the CPU implements a control algorithm to provide the driving signals to the servomotors. At first, the reference trajectory is micro-interpolated at high rate, typically at 1-2 kHz. Second, the closed loop control algorithm operates on the error signals between the micro-interpolated reference values $(\theta_r, \dot{\theta}_r, \ddot{\theta}_r)$ and the actual values $(\theta_a, \dot{\theta}_a, \ddot{\theta}_a)$ of joint position, velocity and acceleration measured by joint sensors. Its outputs, the torque signals τ, are finally sent through the Digital Servo Amplifier to generate currents/voltages that drive the motors. The functional block diagram of a robot control system is depicted in Figure 2-10.

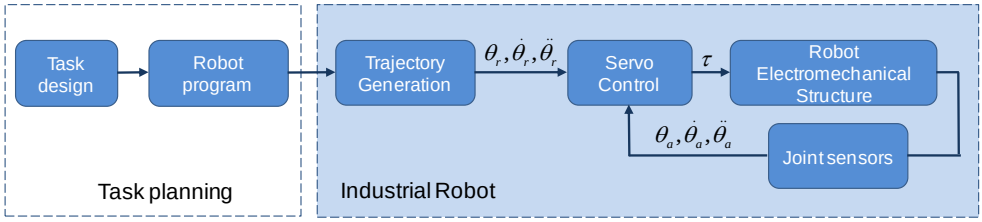


Figure 2-10 Functional block diagram of a robot system

2.2.3 End-effector

In robotics, an end-effector is a peripheral device connected to the end of a robot arm to interact with the environment. The specific design, mechanical structure and components of this device depend on the robot's application.

Since airframe assembly is primarily focused on drilling, fastening and accurate positioning and inspection of aero-structures, the end-effectors of dedicated robots usually have processing tools and sensors for such operations (Kihlman, 2005; DeVlieg, 2010). Moreover, because it is difficult to command a robot to perform one operation, e.g., drilling, change its end-effector and return to the exact position to perform subsequent operations, e.g., fastening, the end-effector is usually designed to be multi-functional. The robots are kept static at the programmed location whereas the end-effector comprises the components to perform all necessary functions. Figure 2-11 illustrates such a multifunctional end-effector which incorporates two main functions: drilling and hole inspection. The basic platform of the end-effector consists of the base that attaches to the robot, a shuttle table, frame/pressure foot, a nosepiece, and process tools (drill, probe). When the robot is in position, the nosepiece is pressed against the work-piece by the pressure foot to provide the system overall stiffness and prevent burrs from forming between stacked materials. The shuttle table drives the tools to the nosepiece where the tools will perform drilling and inspection of the quality of the hole.

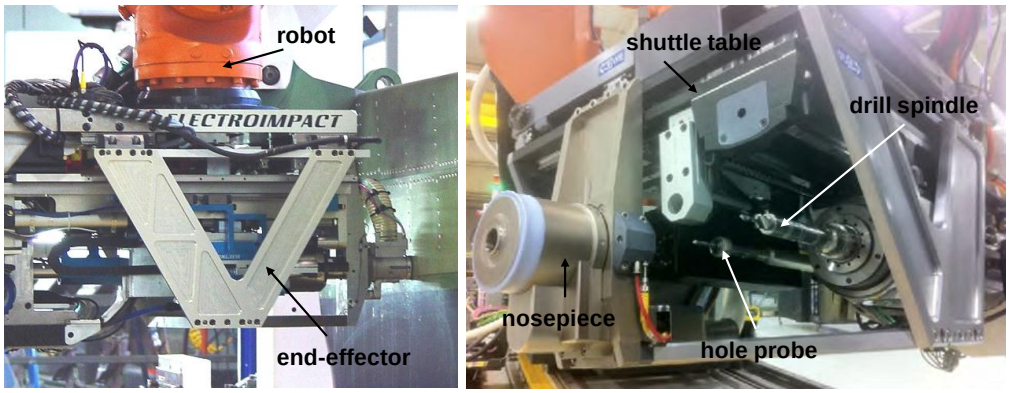


Figure 2-11 A multifunctional robot end-effector for drilling and hole inspection (ElectroImpact)

2.2.4 Programming

Similar to computer numerical controlled machine tools, industrial robots can be programmed either by manual coding in robot programming languages or using simulation software to generate the code automatically. The two methods are given the names online and offline programming to differentiate whether the programming process actually involves the physical robot system.

In online programming, the initial robot program can be created by using a text editor on the PC then downloaded to the controller or directly on the teach pendant, but the main point of this method is that the robot is taught position data. The operator uses the teach pendant to step through the program and jog the robot along the desired path. At each path point, position and orientation of the end-effector is visually corrected and recorded in the robot controller. When the program is executed in automatic mode, the robot can return to these taught locations accurately, owing to its repeatability. This teach-and-repeat method, though straightforward, is time consuming and only suitable when the path contains a few numbers of points. It is not viable, for example, if the task is a machining operation on a part having complex surfaces (e.g., a wing skin) since no visual aided method can guarantee the required normality between the TCP and the machined surface in such cases.

Offline programming (OLP) software, on the other hand, generates a robot's paths automatically based on the 3D CAD model of the part. OLP software can be provided by robot manufacturers or a third party vendor, such as DELMIA from Dassault Systems, the vendor of the popular CATIA software in CAD/CAM technology. Herein, a virtual environment that models the actual work-cell, including the robots, the part, the end-effector and other supporting structures such as linear rails, jigs, fixtures etc., is constructed using computer graphics (Figure 2-12). Within the environment, robot TCP locations are extracted from the CAD model of the part by assigning corresponding *name tags*. Additional tags such as home points, approach and retract points as well as the paths along which the robot travels from one tag to another also need to be specified. Notice that these points are usually defined in the User frame associated with the CAD model of the part. The whole process can be simulated for accessibility and collision detection then fine-tuned for cycle time optimization before the robot program is generated and used by the robot system.

OLP offers several advantages over the traditional manual online programming. Firstly, since the programming stage does not involve the actual robots, it can be carried out before or in parallel with production, and hence, reduces the production down–time. Secondly, through visualization and simulation, OLP is helpful for designing the work cell layout, selecting the right robot models, tooling and equipment needed as well as verifying the robots' reaches and accesses before the actual process takes place. Moreover, the generated programs are less error prone and robots' movements are optimized, thereby increasing safety and productivity (Pan *et.al.*, 2012). OLP, however, also comes with shortcomings, i.e., discrepancy between the virtual and the real worlds. Therefore, rigorous calibration must be carried out to find exactly offsets in translation and orientation between the actual robot Base and the part's User frames before the generated OLP program can be used.

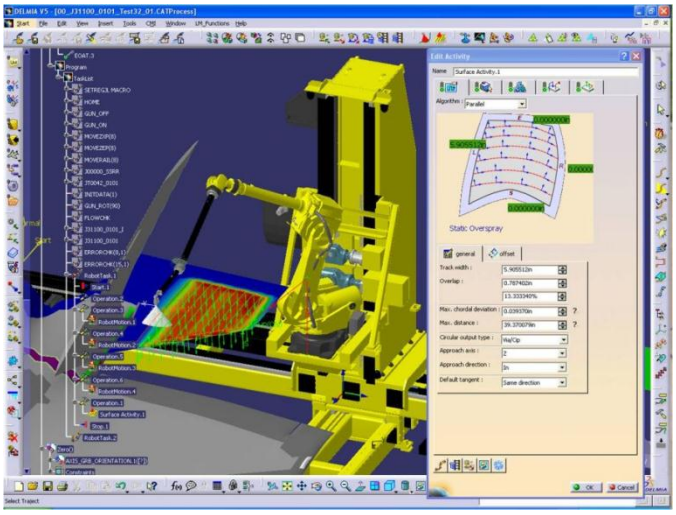


Figure 2-12 Path generation and simulation of a robotic painting process for Lockheed Martin‘s F35-Lighting II aircraft by OLP software DELMIA (Ponticel, 2011)

2.2.5 Robot accuracy and the challenge in airframe assembly

Industrial robots are ideal for repetitive tasks like spot welding or pick-and-place in automotive assembly. The high production rates in the automotive industry allow robots to be used for a single process or in some cases, a single component thousands of times. In that sense, robots only need to be repeatable but not necessarily accurate since the accuracy can be manually corrected with the aid of the teach pendant during online programming. Robot manufacturers have managed to improve robot repeatability by increasing the interpolation rates and resolutions of the joint position sensors. Current industrial robots possess quite good repeatability, ranging from ±0.05 to ±0.3mm. The absolute accuracy usually is not documented by robot manufacturers but is much worse, from ±10mm to a few millimetres for off-the-shelf industrial robots (Siciliano *et.al.*, 2008).

In aircraft assembly, unfortunately, absolute accuracy is mandatory. In contrast to the automotive industry, aerospace manufacturing has much lower production rate but enormous number of contained parts and operations required for the final assembly of an aircraft. To give a concrete comparison, an

automobile usually comprises 20000 components while that number for the Boeing 777 is 4 million (Kihlman, 2005). During the assembly process of the Airbus A340-500/600 wing panels, approximately 65000 holes have to be drilled on each skin (Rooks, 2001). On top of that, airframe assembly demands for tighter tolerances: while a spot weld gun in car assembly is usually positioned within ±1-2mm (Axelsson, 2002), the tolerance for a drilled hole in aero-structure assembly is usually ±0.2mm. In such scenarios, manual compensation of robot inaccuracy by the teach-and-repeat method is too costly, tedious or even impossible. OLP, on the other hand, must rely on robot absolute accuracy and accurate modelling of the virtual environment.

Why do robots have poor accuracy? During trajectory generation for motion control, a robot controller uses the kinematics to infer the end-effector locations from joint positions. For the sake of computability, the kinematic models used by robot controllers are based on several assumptions, such as nominal link lengths and perpendicularity or parallelism between joint axes but in fact, these parameters are subjected to manufacturing tolerances. In addition, joint position sensors also have errors and since they are located at the back of servo motors, any errors in transmission components ahead of them, such as compliance and backlash, go unaccounted for. These errors are usually small but due to the nature of coupling of links in serial manipulators, they are amplified throughout the remaining links and finally yield significant errors in the tool pose. In general, the internal errors can be categorized into:

- Geometric errors: errors in the kinematic parameters, such as offsets in joint positions, dimensional and angular variations in link lengths and angles resulted from imprecise manufacturing. They are constant or position-independent.
- Non-geometric errors: including compliance, backlash, eccentricity and wear in gear transmission as well as thermal expansion etc. that deteriorate robot accuracy. They are variable but somewhat predictable (Renders *et.al.*, 1991, Karan *et.al.*, 1994).

DeVlieg (2010), a researcher at ElectroImpact on robotic applications for airframe assembly has made several assessments on the robot accuracy. He stated that, on a typical 3 meter serial robot, geometric errors may cause errors in the TCP of about ±2mm to ±4mm. Compliance (or deflection) is another main source of inaccuracy for serial manipulators due to the inherent lack of stiffness. Since the structure of a serial manipulator is a concatenation of links, the majority of compliance occurs at the joints and could be induced by both link masses and applied payload, e.g. weight of the end-effector used. The author claimed that these deflections might contribute to an error of 3mm or more at the end-effector's TCP. When the robot performs machining, e.g., drilling, its accuracy is further degraded under the effect of contact forces. The contact forces in this case include the pre-pressured static force applied before drilling when the nosepiece is pressed against the work-piece surface and the dynamic thrust force during drilling. Under the effects of these contact forces, the robot may exhibit further errors up about 2mm. Table 2-1 summarizes the error sources that degrade the performance of a robot.

In general, the accuracy of the robot system depends not only on the robot but other factors:

- Accuracy of the coordinate transformation between the robot Base frame and the part's User frame;
- Dimensional variations and deformations of the part; this is originated from the fact that the parts in airframe assembly are usually large and compliant structures (Sadaat *et.al.*, 2009);
- Accuracy of the transport system, e.g., linear track or gantry platform used for expanding the workspace of the robot.

Table 2-1 Error budget of an industrial robot

Sources of error	Type	Characteristics	TCP error
Joint offsets	Geometric errors	Constant, ever-present	2-4 mm
Manufacturing tolerances			
Joint and link elasticity	Non-geometric errors	Variable, presences and magnitudes depending on the property and status of the robot mechanical structure and working environment	3mm
Thermal effects			1mm
Backlash, wear etc.			

An economically feasible solution for improving the accuracy of the robot is via calibration: by identifying and compensating for the above geometric and non-geometric errors in the robot structure. Since these internal errors usually cannot be measured directly, they must be identified via a mathematical model relating them with the measurable tool pose errors. This model is referred to as the *error model* of the robot. After the model has been constructed, the robot is commanded to a number of programmed locations in the workspace and its actual tool poses are measured by a sensor. Tool pose errors are determined from the measurements and hence, the internal errors of the robot structure can be solved from the error model. If the model is accurate, its predicted tool poses, which are the nominal plus the predicted errors, must be close to the measured values (Figure 2-13). After calibration, the error model can be used as a virtual sensor that "measures" and compensates for robot inaccuracy without the need for the actual one. It has been reported that through proper calibration technique, the accuracy of a serial manipulator could be improved up to 0.5mm (Schröer *et.al.*, 1997). Further details of the technique will be described in Chapter 4.

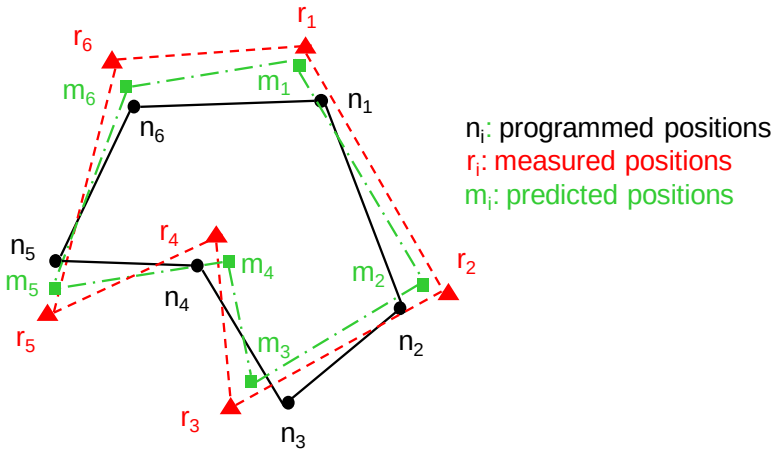


Figure 2-13 Principle of robot calibration for accuracy improvement

It can be seen that industrial robots, though properly calibrated, still require in-process error correction from an actual metrology system to meet the ±0.2mm required tolerance in airframe assembly. The general idea of utilizing metrology in the area is to provide the robots direct perceptions on the positions of their

end-effectors or the part, from which fine adjustments can be performed. The metrology that has been used for improving robot accuracy will be presented in the next section.

2.3 Metrology for robotics

2.3.1 Global sensors

The term “global sensors” in this work refers to large volume metrology such as laser and vision systems located separately from the robots. Large-volume metrology is used conventionally in aircraft manufacturing industry for integration and assembly of aircraft structures (Rooks, 2001), verification and calibration of jigs, fixtures, and tooling (Saadat *et.al.*, 2002), as well as part conformity inspection (Saadat *et.al.*, 2009). Due to the capability of providing accurate measurements in three or even six dimensions, such a global metrology system can be used to measure robot positions and form a closed loop control to improve its positional accuracy. The following sections will introduce the technologies that have been used for this purpose.

2.3.1.1 Laser tracking system

A laser tracker is a non-contact coordinate measuring machine (CMM), capable of taking a large volume of measurements with an accuracy of few micrometers over a range of tens of meters. The main part of a laser tracker is a Laser Interferometer (IFM) having a beam-steering mirror driven about horizontal and vertical axes to direct the laser beam in a wide range of directions. When the beam is pointed at a retro-reflective target [e.g., a spherical mounted reflector (SMR)], it returns along its original path back to the IFM. Based on classical interferometry, the IFM determines the relative distance between the reflector and the tracker, providing one dimensional measurement. It is then combined with the other two dimensions, the azimuth and elevation angles, measured by optical encoders at the motorized stage that drives the beam-steering mirror (Figure 2-14). These measured polar coordinates can be transformed to give the Cartesian coordinates of the SMR target.

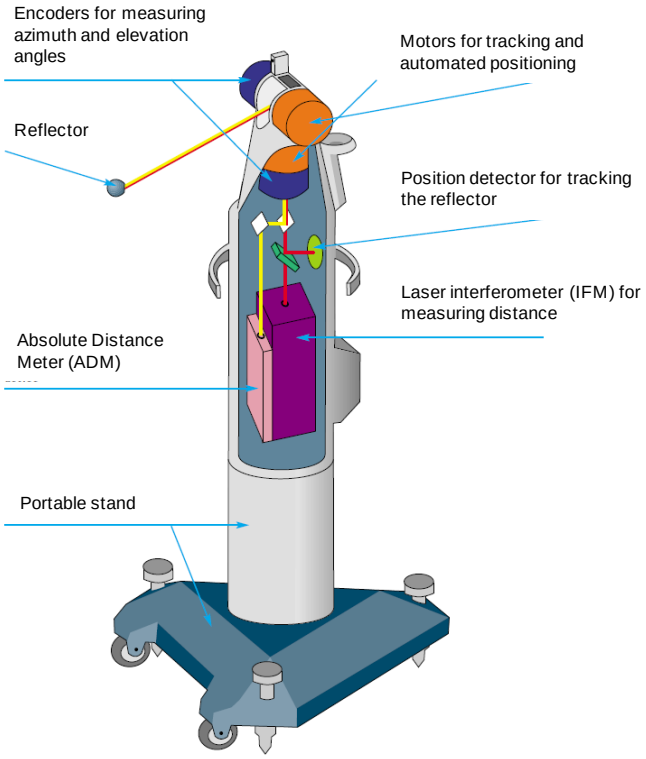


Figure 2-14 Components a laser tracker (Leica Geosystems)

A notable feature of a laser tracker is the ability to follow the movements of the target. When the reflector moves, the beam hits the target off-centre, causing a lateral displacement between the emitted beam and the returned beam. A two dimensional position detector in the laser tracker measure this displacement and generates a signal to adjust the steering mirror until the beam is centered back to its desired coaxial state (Figure 2-15). This mechanism allows the device to keep track of the target movements of up to 5 meters per second.

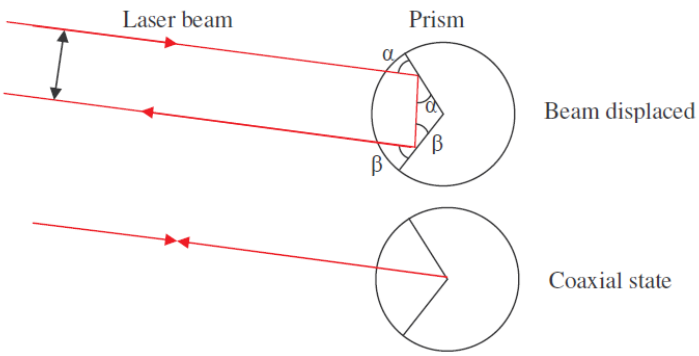


Figure 2-15 Principle of target tracking

Newly developed laser trackers by Leica Geosystems are capable of measuring both positions and orientations of an object when they are equipped with a camera and use a special reflector called TMAC (Tracker-Machine control sensor). This reflector is a versatile device which can be mounted on a robot or machine spindle to provide offset positions of their TCPs in six degrees of freedom. It comprises of a prism located at the centre of an aluminium housing and a pattern of ten light emitting diodes (LEDs). While the prism is measured by the tracker to provide three position parameters, images of the LEDs captured by the camera are used to determine three orientation parameters of the TMAC around the principal X, Y and Z axes (Figure 2-16).

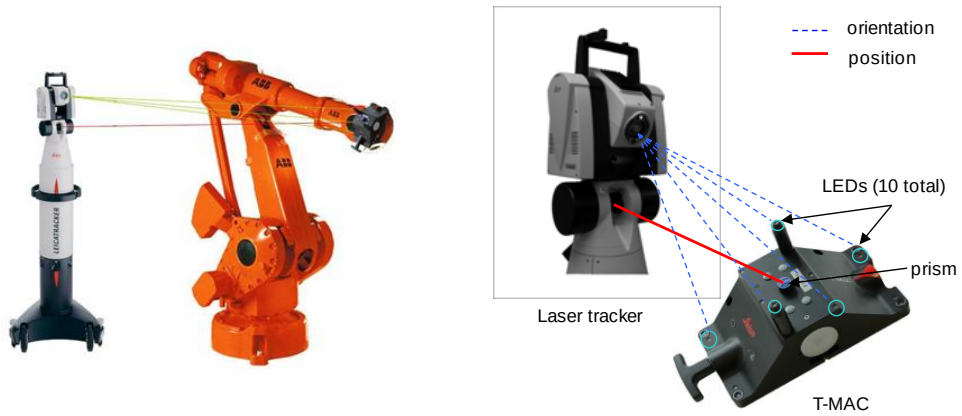


Figure 2-16 Working principle of the laser tracker and T-MAC to provide 6D measurements

Another advanced technology, the Absolute Distance Meter (ADM) also patented by Leica Geosystems, was introduced into the latest versions of their laser trackers. The conventional laser interferometer IFM, in fact, can only measure relative distance between the reflector positions when it moves. To determine the absolute distance of a new reflector position, distance to a known starting point in space must be measured in advance. In earlier versions of laser tracker, this required the operator to bring the reflector to a pre-calibrated 'home' position before any measurement takes place or whenever the laser beam is interrupted. This manual action prevents the measuring process from being fully automated. ADM, a laser technique utilizing the polarization modulation of the laser light (Leica, 2008), was implemented in new laser

trackers to determine the absolute distance without the requirement to relocate the reflector to a home position. By exploiting this useful feature, the work presented in this thesis proposed a method using one laser tracker for multiple robots, in which the laser beam can be disconnected selectively from one robot and pointed to another to measure and correct its end-effector's position.

2.3.1.2 Other technologies

Other large scale metrology technologies widely used in airframe assembly are photogrammetry and Indoor GPS (iGPS). These technologies rely on the principle of sensor triangulation to determine position of a target in space. In a photogrammetric system, the 3D position of a target (e.g., a light emitting diode or a retro-reflective target) is constructed from the images taken by a system of cameras at pre-calibrated configurations. Several targets can be used to infer both the position and orientation of the object, i.e., the part or the robot end-effector, to be measured. An example of a photogrammetric system is the K-series Optical CMM of Nikon Metrology which has been used to correct robot positions (Figure 2-17). An iGPS system, on the other hand, uses a network of transmitters emitting laser and infrared lights to determine the position of a receiver in its working volume. Signals from two or more transmitters to the receiver will be used to triangulate three dimensional coordinates of the receiver, assuming the relative distances between transmitters are known beforehand (Nikon Metrology, 2011).

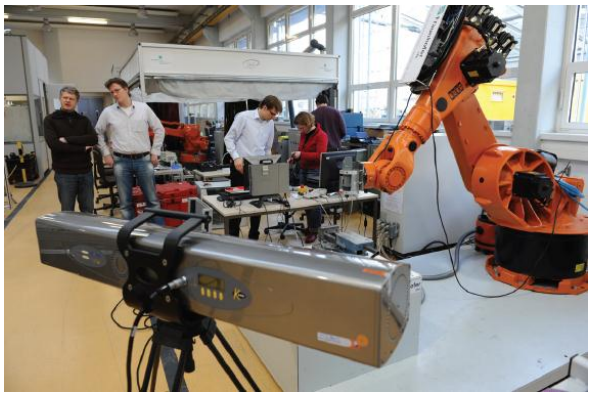


Figure 2-17 The K-series Optical CMM (photogrammetric system) is used for tracking the position of a KUKA robot (Nikon Metrology)

Each large volume technology presented in this section has its advantages and disadvantages when used for the purpose of robot positioning. The key factors of laser tracking technology are its superb accuracy, large measurement volume and speed. A typical laser tracker can measure a target with the accuracy of 15μm in its near field (within 7m) and with the sampling speed of 1000 points per second. Its shortcoming is the requirement of continuous line-of-sight between the tracker and the robot end-effector when the robot is moving. The ADM technology of new laser trackers can be used to overcome this problem when the laser beam is accidently broken, but it needs to know accurately the robot's new position in order that the connection can be re-established. Photogrammetric and iGPS systems, on the other hand, do not suffer from this problem and have the capability of measuring several targets simultaneously. The downsides of a photogrammetric system are its limited measurement volume, constrained by the field of views of the cameras and the degraded accuracy when the measured target is far from near field of the sensors (Kihlman, 2005). Accuracy and measurement speed are also the main drawbacks of iGPS systems despite their large measurement volume (Wang *et.al.*, 2010). To summarize, a comparison between current large volume metrology technologies is given in Table 2-2.

Table 2-2 Large volume metrology used in airframe assembly (Saadat *et.al.*, 2002)

Technologies	Laser Tracker	Photogrammetry	iGPS
Measurement Range	45m	17m	200m
Accuracy	15μm + 6μm/m	90μm + 10μm/m	170μm within 12m
Sampling rate	1000Hz	1000Hz	40Hz
Working volume	Large	Limited	Large
Multiple targets	No	Yes	Yes
Cost (€)	150K	120K	250K

It can be inferred from the table that laser tracking apparently is the proper technique when accuracy is a prerequisite. It is also worth noticing that all of these technologies, though necessary to improve robot accuracy, require capital

investment. This is somewhat contradictory to the original purpose of utilizing industrial robots in airframe assembly for cost reduction. Therefore, having one accurate metrology system capable of serving multiple robots is desirable.

2.3.2 Local sensors

The term “local sensors” in this context refers to a variety of sensors mounted on the robot arms. They are categorized into:

- *Proprioceptive sensors*: to measure the internal state of the manipulator. Common proprioceptive sensors are encoders and resolvers for measuring joint positions and tachometers for measuring joint velocities. These sensors are integrated as parts of a robot system; their measurements are used as position feedback in the servo control as presented in section 2.2.2;
- *Exteroceptive sensors*: to provide information about the external environment in terms of distance to the part, its size and shape, interaction forces, and so forth.

According to (Siciliano *et.al*., 2007), common exteroceptive sensors used for industrial robots are:

- Stress sensors: including wrist Force/Torque (F/T) sensors and shaft torque sensors used in-process for measuring the stress induced by the contact between the robot and the part. Other sensors of this type are tactile sensors and other sensorized compliant devices;
- Range sensors: including laser sensors, vision systems and mechanical probes to measure dimensional quantities of the part;
- Other types of sensors used for a specific process, such as proximity sensors, temperature sensors, accelerometers and gyroscopes etc.

These sensors can also be classified on the basis of sensing mechanisms into contact and non-contact sensors (Gupta *et.al.*, 2007) or type of output signals into digital and analog sensors (Kurfess, 2005).

A typical stress sensor is the wrist F/T sensor usually mounted between the outer link of the manipulator and the end-effector. The main component of the

sensor is an elastic structural element which deflects proportionally to the applied force or torque when the end-effector is in contact with the environment, e.g., drilling or picking an object. Either strain gages or piezoelectric materials are used to measure the deflection of the elastic structure to provide complete contact force information: three translational forces and three torque components around the principal X, Y and Z axes (Figure 2-18).

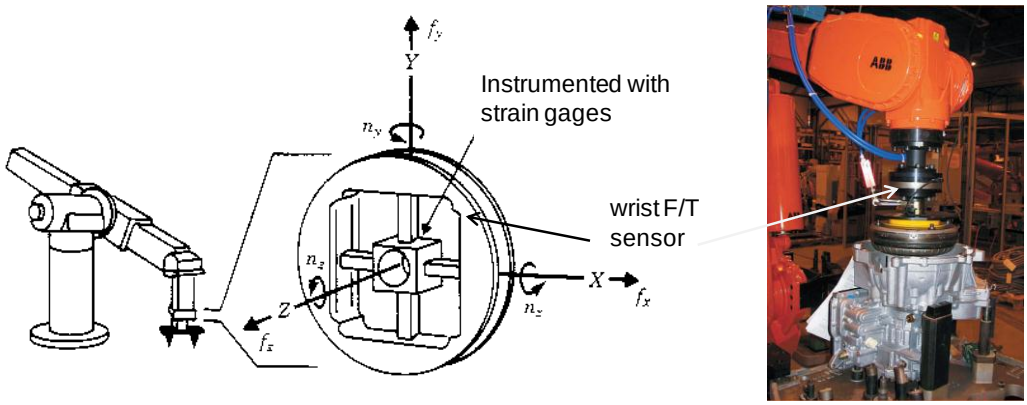


Figure 2-18 Wrist F/T sensor (Craig, 1989)

A typical range sensor is the laser sensor. Laser sensors can operate on the principles of time-of-flight, optical triangulation and interferometry (Siciliano *et.al.*, 2007), which basically are the technologies employed in the iGPS, photogrammetric and laser tracking systems presented in section 2.3.1. The operating principle of optical triangulation laser sensors is illustrated in Figure 2-19. Light emitted from a laser diode is projected on the object, usually in the shape of a point or a stripe. The reflected beam is focused by a converging lens onto a photo-detector, which usually is an array of Charge Couple Devices (CCD). Once the relative distance and orientation between the CCD array and the laser is known precisely, e.g., through a calibration procedure, it is possible to determine from the captured image the distance between the sensor and the object by simple geometrical calculation. A special sensor of this type commonly used in robotics is the seam tracking sensor for welding applications. In such a sensor, the sensor head has a built-in electronic circuit that detects and calculates simple geometries of the joint between two parts to be welded, e.g., its shape, height, gap size and position of the centre point. Based on these

data provided in real-time, robot motions can be continuously adjusted such that the welding torch always follows the seam (Gooch, 1998).

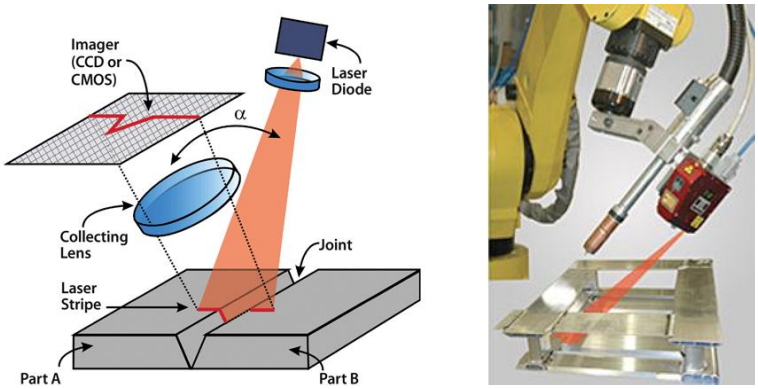


Figure 2-19 Laser sensor based on optical triangulation

2.3.3 The use of sensors in robotics

Sensors in robotics are classified according to their functions into the following categories (Duelen *et.al.*, 1987).

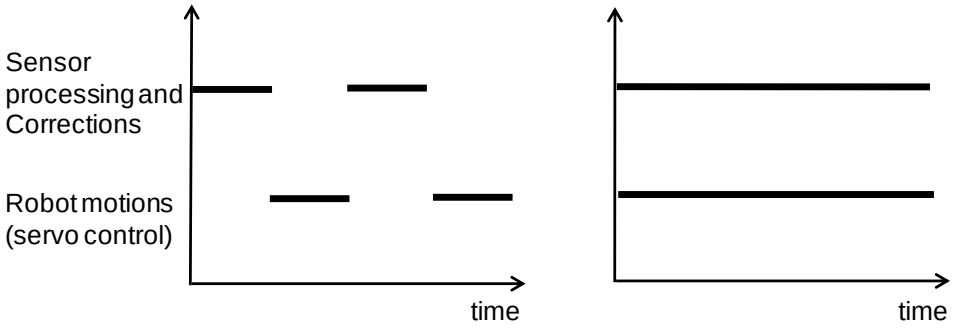
2.3.3.1 Process control

The sensors supervise the work in progress or the work-cell in order to sequence the tasks between multiple robots and machine stations in a production line or to detect the existence of parts and human for safety or human robot collaboration. A variety of sensors can be used for this purpose, ranging from simple tactile and proximity sensors to more complex vision systems. Nevertheless, they only behave like electric switches to start different routines in a robot program or interrupt the current program to handle safety conditions.

2.3.3.2 Robot control

Signals provided by the sensors are used to modify the programmed motion profile of the robot in order to correct deviations in position and orientation of the robot, the part, fixtures or all of them. The correction actions are further classified as:

- Static correction, where sensor signals initiate the corrections in the robot program coordinates before the coordinates are processed by the robot controller for motion control. Error corrections and robot motions, therefore, take place in sequence.
- Dynamic correction, where the corrections take place in parallel with the robot motions (Figure 2-20).



a) Static correction b) Dynamic correction

Figure 2-20 Sensor-based correction strategies (Warhburg, 1988)

The main difference between the two correction methods is that static correction does not require continuous feedback from sensor signals whereas dynamic correction utilizes these signals as continuous feedback in a closed-loop control strategy superimposed on the joint servo control of the robot controller. The latter thus requires signal processing of sensor information and computation of relevant control algorithms at a high cycle rate and within a specific time frame, i.e., real-time characteristics. Typically, these tasks are implemented during the interpolation cycle of the robot industrial controller (Schreiber *et.al.*, 2010) as depicted in Figure 2-21.

To the best of the author's knowledge, the sensors and corresponding dynamic control include:

- Wrist F/T sensors for force control in robotic drilling, de-burring etc.,
- Seam tracking sensors for continuous tracking control in welding,
- Vision systems for visual servoing and target tracking control in pick and place operations.

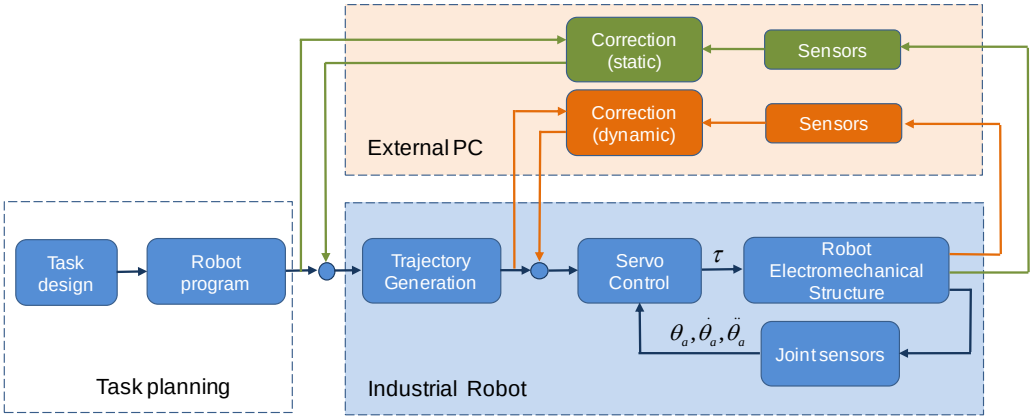


Figure 2-21 Functional block diagram of a robot system with sensor-based corrections

To give the readers an overview of how metrology assists robotic assembly, an example is given in Figure 2-22. In the figure, the robot carries a multifunctional end-effector embedded with a drill actuator, a range sensor, e.g., laser sensor or vision system, a wrist F/T sensor and is tracked by a laser tracker.

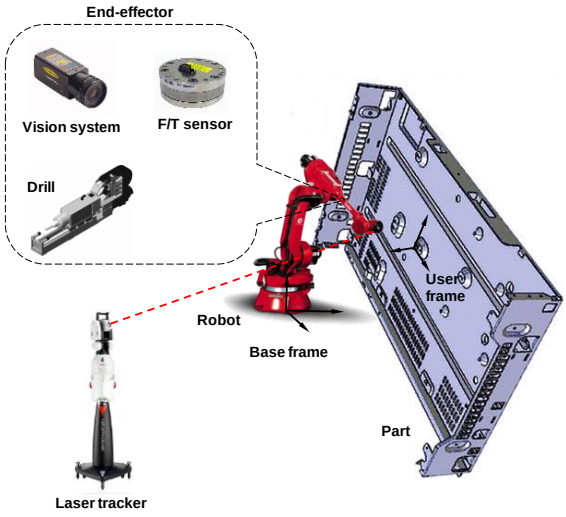


Figure 2-22 Example of a metrology assisted robot system for assembly applications

Before the drilling process can take place, the sensors will be used for *part localization*, that is, calibration of the User and Base frames. Based on prebuilt knowledge of the part (from its nominal CAD model), the robot commands the vision system and the laser tracker to measure some features (dowel holes, edges etc.) on the part and then uses these measured features to set-up the

User frame. Offsets in positions and orientations between this actual User frame and its nominal frame constructed from the part's CAD model due to misalignment or distortion of the part due to temperature fluctuations are computed, allowing for the robot to make (static) corrections to programmed drill locations accordingly. This approach is only feasible if the part contains measurable features, otherwise the laser tracker must be used to measure some reflector targets mounted on the fixtures holding the part (not shown in the figure) to construct the User frame. By having the robot capable of part localisation, the necessity for large and dedicated fixtures used for positioning the part, such as the one shown earlier in Figure 2-4, can be eliminated.

When the robot approaches a drill location, the laser tracker is employed to correct the positional error of the robot. This error is the combination of the robot's inaccurate kinematics and elasticity induced by the gravitational force on the end-effector and the pre-pressured force as presented previously in section 2.2.5. Because these forces are only static, a static correction strategy is usually performed: the robot will be positioned iteratively until the position of the end-effector, measured by the laser tracker, is well within the required $\pm$0.2mm tolerance.

During the drilling process, dynamic correction for the tool's deflections is necessary because the thrust force might change its magnitude rapidly (Kihlman, 2005). If the laser tracker is capable of measuring in real-time, its measurements can be used to correct the tool's deviation by a closed-loop position control algorithm. Otherwise, the F/T sensor and a force control algorithm must be employed to maintain the tool at the programmed position despite the effect of the cutting forces.

After the drilling has been completed, the vision system can be used again to verify the position of the drilled hole. By doing this, the robot system is employed at this stage as a shop floor CMM without having to divert the part to the laboratory for quality checking. Examples of these correction methods will be presented in part 3.2 of Chapter 3.

2.4 System integration

The metrology-assisted robot system presented above requires integration of the robot with other peripheral devices, i.e., actuators, local and global sensors. On a larger scale, this robot system is part of a work-cell that might consist of several robot systems, automated machines and vehicles, conveyors and other sensors for process control etc. The use of these proprietary *field devices* requires communication and control infrastructures, developed to link and coordinate the activities of specific devices and systems having incompatible communication interfaces and data representations.

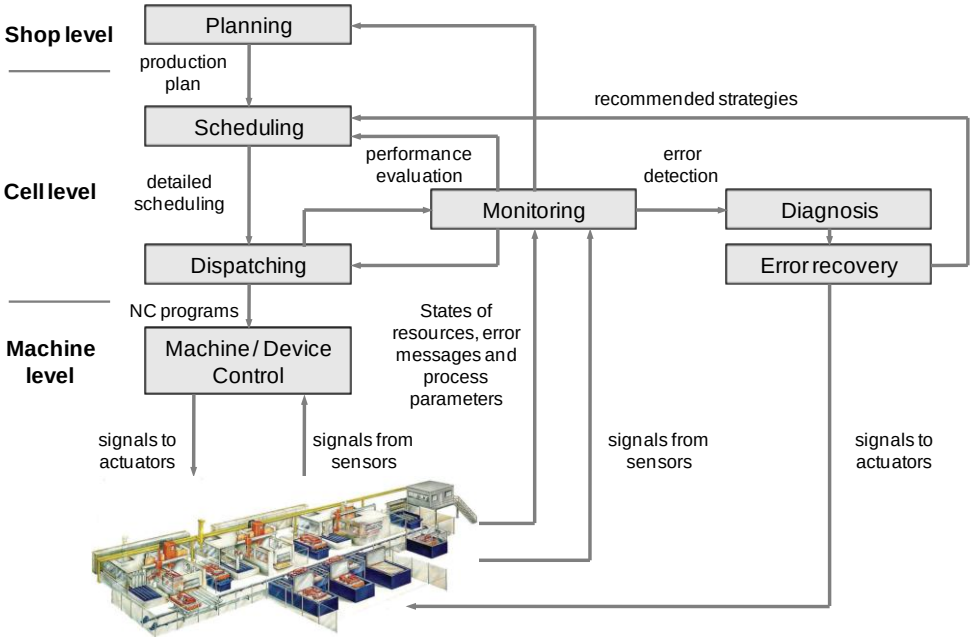


Figure 2-23 Control hierarchy in manufacturing systems (Leitão, 2009)

2.4.1 Communication architectures

An automated assembly system, or manufacturing system in general, comprises three levels of hierarchy: shop, cell and machine levels as depicted in Figure 2-23 (Dilts *et.al.*, 1991). The operations at shop level involve production planning and manufacturing resource allocation. Cell level is responsible for scheduling and dispatching the production plan, e.g., offline robot programs, issued from the upper level as well as monitoring process status reported from lower level. These activities are referred to as *process*

control. Physical manufacturing activities, e.g., assembly processes, take place at the machine level where separated robots and field devices are co-ordinated together, or *machine*/*device control*, by dedicated controllers which usually are PCs running on top of their industrial controllers. These industrial controllers are in charge of low-level *motion control* (for robots/actuators) and signal processing (for sensors) of their devices. Static correction is thus a type of device control whereas dynamic correction is a type of motion control with aid from the sensors.

The primary conduit for data exchange between relevant stations (computers/controllers) at the same and different levels leads to a corresponding system of networks depicted in Figure 2-24 (Zurawski, 2007; Hung *et.al.*, 2010):

- At shop level, the network(s) are typically used for exchanging manufacturing/process messages and various enterprise management applications. Ethernet based on the TCP/IP protocol suite represents the backbone with which the computers at this level are connected to each other and with cell controllers at cell level. At this level, the traffic is characterized by high data rates (the amount of data) whereas message delivery time is not critical.
- At cell and device levels, field devices are connected to the PC-based controllers either directly, i.e., point-to-point connection or via industrial networks to exchange data for process control (at cell level) and device control (at device level). Serial communication buses such as the RS-232/485, PCI (Peripheral Component Interconnect) and USB (Universal Serial Bus) are typically used for point-to-point connections. On the other hand, network connections are usually formed by a variety of *fieldbus* systems whose communication protocols are either built upon their own protocol suites such as the Profibus, Interbus, WorldFIP etc. or on top of the TCP/IP protocol suite such as the ProfiNet, Ethernet/IP, DeviceNet etc. Comprehensive reviews of these fieldbus technologies can be found in the book edited by (Zurawski, 2007). There is a growing tendency for these levels of networks to be based directly on the standard Ethernet

and TCP (UDP)/IP protocol suite (Vitturi, 2001). These networks are characterized by small data rates and cycle time, typically from 1-10ms (Neumann, 2007).

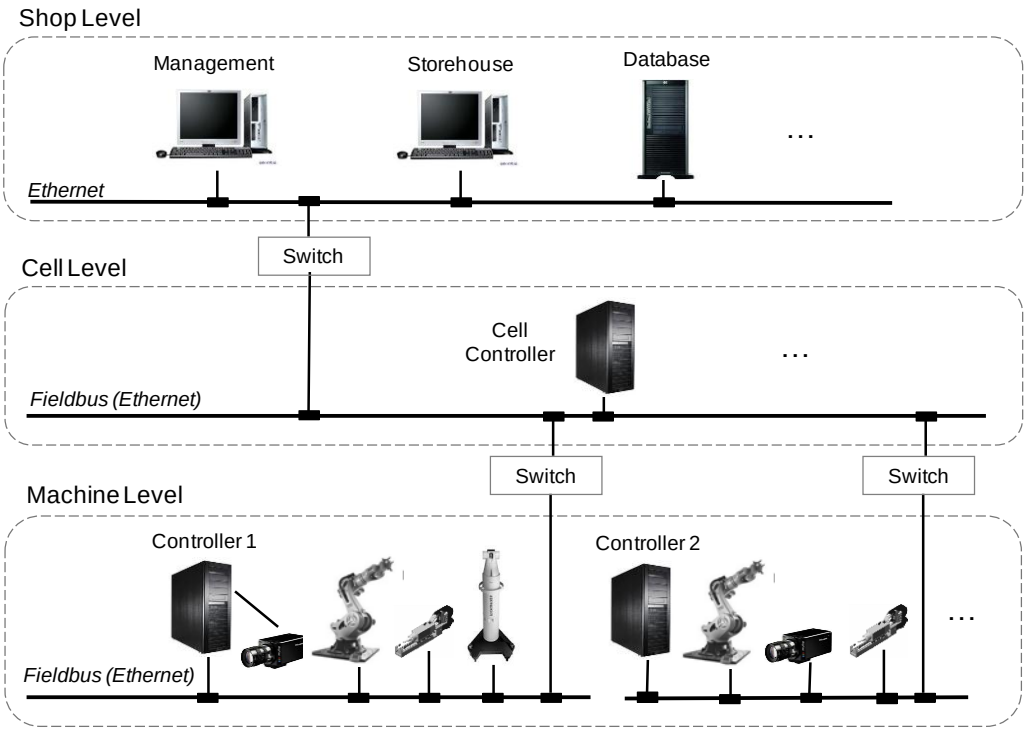


Figure 2-24 A typical network architecture in industrial automation (Zurawski, 2007)

2.4.2 Control applications

In Figure 2-24 above, the Controllers at machine level usually are PCs with control applications developed in some computer programming languages to co-ordinate separate field devices through their industrial controllers. At times, these control applications might also link with other non-physical resources such as third party software for complex processing of sensor data (e.g., image processing or numerical regression analysis). They will be described further in the following sections.

2.4.2.1 Control applications with centralized processing

A control application at machine level usually comprises the following abstraction layers (Figure 2-25):

- At the bottom is the Device Interface for communication with the controlled components (hardware/software) via some types of communication libraries provided by their vendors. The communication libraries typically are usually known as the Application Programming Interface (API) and Software Programming Kit (SDK), or simply “drivers” in the computer world.
- At the middle is the Control layer, which interprets the OLP robot program dispatched from the Cell controller along with its associated name tags into native API control commands of the robot and peripheral devices to be executed. It also performs necessary orchestrations and correction activities previously described in section 2.3.3.2.
- At the top level is the Application Interface through which the application receives input and reports process/system status to human operators and the cell controller.

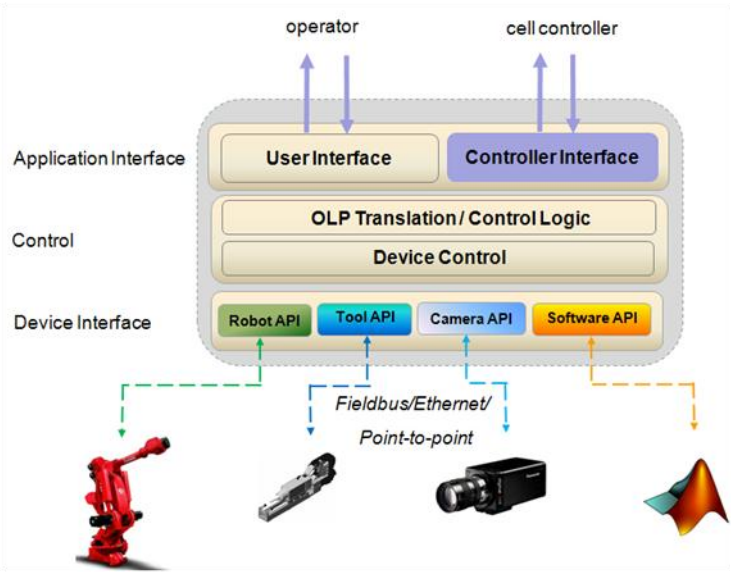


Figure 2-25 A control application with centralized processing

Such an application exhibits a centralized control unit since all the control and data processing functions are performed on a single computer. If only a few

numbers of devices are used, e.g. one robot with simple peripherals, developing a control application following this centralized approach is intuitive and relatively simple. In addition, direct communication between the application (software) and the hardware components implies fast and reliable data transfer and control. This feature essentially makes centralized control architectures ideal if the control algorithm must be guaranteed in real-time. However when numerous devices are used, this type of control application exposes several disadvantages. When it has to manage multiple concurrent processes, bottlenecks might occur at the Device Interface and Control layers due to the limited processing capability of a single computer. Increased software complexity will also be linked with inconsistency and a greater likelihood of failures during runtime, such as conflicts or deadlocks in parallel operations. On top of that, the major disadvantage of this type of application is the weak response to change since its control logic was developed to be tailored with existing hardware and was hard-coded in the software. As a consequence, any changes in the structures due to hardware replacements, upgrades or in production scale, products and control algorithm might require tedious modification of the software.

2.4.2.2 Control applications with distributed processing

The aforementioned shortcomings of a centralized control unit have led to the consideration for distributed control systems. Distributed control was originated with advances in information technology: the Remote Procedure Call (RPC) which allows separate software applications to exchange information at programmatic level to share the work load. By exploiting this capability, the centralized control application can be subdivided into several networked applications, each of which provides a *service*, representing either a physical or non-physical (software) device or a subsystem. Each of these distributed applications has its own external interface (a collection of public functions) allowing other applications to manipulate the devices represented. For example, the interface of the control application for a camera might have a function named *SnapShot* for taking a single picture (Figure 2-26). The function might be

implemented in different ways for different cameras used, based on their APIs, but is still invoked in the same manner by other applications. As a result, these applications are able to control the camera, or in order words, consume its service via the standardized interface without having to know the implementation details and hardware specifics of the camera. Any changes regarding how the camera is controlled or replacement of the camera with another one might only require modification of its control application without affecting the dependants.

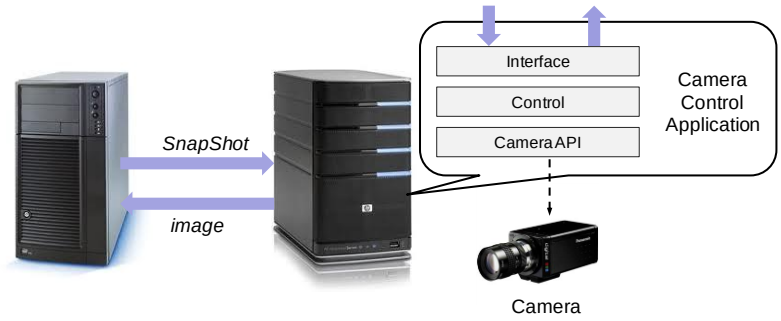


Figure 2-26 Control application of the camera allows interactions with other applications via its standardized interface

The principle of remote procedure calls between distributed applications is described in Figure 2-27. A RPC platform, also known as *communication middleware*, acts as an application layer for one application to transparently invoke a function of a remote application as if it were its local function. It allows for applications to exchange functionality without having to know their locations on the network, communication protocols, programming languages, operating systems, etc. When two applications are supposed to interact, the middleware automatically establishes the client-server relationship between them via a piece of code called a *stub* on each side. When the client application wants to invoke a method on the server application, e.g. the *SnapShot* function, it actually calls the stub on its side. The call is *serialized* into the middleware's *message* (data structure) and sent via the network to the server side via some *transport protocol*. Here, the message is *de-serialized* by the server's stub back to the original *SnapShot* function call in the camera control application, which then carries out the work and returns the captured image in the same manner.

Examples of the well-known communication middleware technologies that provide this capability are CORBA (Common Object Request Broker Architecture) of Object Management Group, DCOM (Distributed Component Object Model) of Microsoft, Web Services and the recently introduced RDS (Robotics Developer Studio) of Microsoft which targets robotic applications. These middleware platforms will be further discussed in section 3.3.2.1.

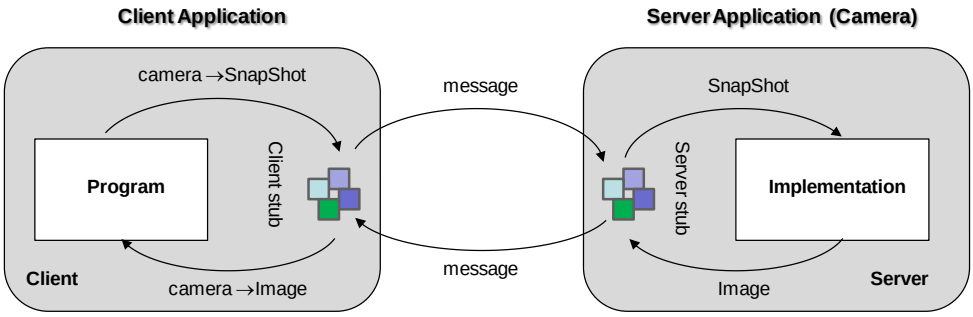


Figure 2-27 Client-server invocation via a middleware platform

A system of distributed control applications communicating via such a middleware platform is depicted in Figure 2-28. The Controller in this case, namely the *composite service*, communicates with the robot, tool, sensor and software used through their control applications, namely the *basic services*, which can reside on the same computer or distributed across the Ethernet. Each basic service controls their own device(s) and performs necessary data pre-processing in order that the data can be used by the composite service. The composite service communicates with the basic services via their application interfaces realized by the middleware used. With this architecture, the Device Interface layer of the composite service contains pointers (the client stubs) to these basic services instead of the devices' APIs. The Control Logic of its Control layer are developed based on the functions provided by the basic services, and thus are less dependent on the devices' hardware. The Application Interface contains the interface of the composite services provided to cell controllers. Up to this point, the reader can figure out the cell controller at cell level also is a composite service built on top of these control applications; the cell controllers again have interfaces which can be accessed by other computers at shop level and so forth.

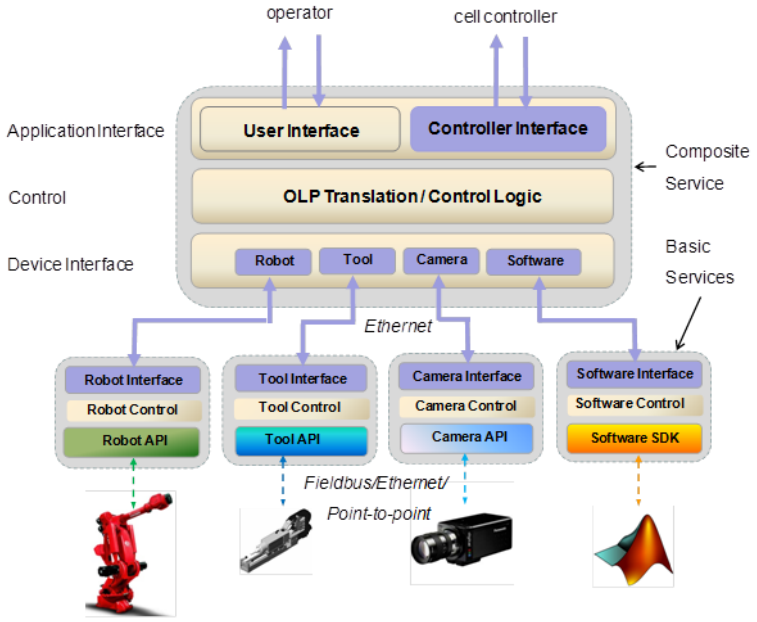


Figure 2-28 A system of distributed control applications

It is straightforward to see that the main advantages of this distributed control and service-based architecture over the centralized control architecture include:

a. Interoperability: multiple devices, regardless of their hardware, communication buses and programming languages supported by the APIs, are able to exchange information.

b. Distributed control: to ease the computation burden on the controllers at higher levels by outsourcing Device Control to corresponding basic services at lower level. This feature is particularly useful when the Device Control requires intensive CPU payload such as image processing or numerical regression analysis etc.

c. Flexibility attained through the *loosely-coupling* nature of services: they communicate via their well-defined interfaces without having to know the implementation details of the others. Therefore, changes that occur within one service, e.g. due to replacement/modification in hardware/control, might not cause cascading changes to other services that consume its functionality. This characteristic makes the system flexible and less dependent on the hardware used (Pires *et.al.*, 2009; Pereira *et.al.*, 2007; Brugali, 2007).

As a trade-off to its flexibility, the main drawback of this control architecture is the inherent latency in data transfer due to the intermediate layers (middleware and Ethernet) between the hardware and software. For this reason, sensors that communicate with robot on real-time basis, i.e., the ones used for dynamic correction, should interface directly with the robot service; even in some restricted cases, they must be integrated into robot controllers. These specific techniques will be discussed further in section 3.3.1 of Chapter 3.

3 LITERATURE REVIEW

This chapter presents a review of the published research relevant to the topics covered in this thesis. In addition, the applications of robotics in airframe assembly will be introduced to provide the reader a wider view.

3.1 Applications of robotics in airframe assembly

Within the literature, a number of historical and current applications of robotics in the aerospace industry that have had varying degrees of success are described. One of the early attempts at using robots in aircraft manufacturing is the EMAP project at British Aerospace (now BAE Systems) in the late 1980s. The system consisted of a large gantry robot performing automated drilling and fastening operations on aluminium flat and formed parts. The system, however, failed due to the inherent quality of individual components (Calder, 2011). In the mid-1990s, Boeing also failed to use a six-axis robot to join the body of its 777 jetliner (Weber, 2009). The concept of open loop control in which the accuracy of the system merely relies on that of the robot was thus to prove a failure. To overcome the inherent inaccuracy limitation of the robots, templates were first utilized as guides for the drill tools. In 1990s, Airbus employed them in the robotic assembly line of A330/340 FAL in Toulouse (Airbus, 2012). Eight Kuka robots, arranged in four sets of pairs, two above/below the wing and on either side of the aircraft, were used for riveting of the wings into the fuselage body. The robots, however, only performed drilling whereas the subsequent insertion of fasteners was still done manually. Pilot holes on the templates were detected by laser sensors mounted on the robot end-effectors so that the robots could adjust their programmed positions within the tolerances of 2.5mm (Kochan, 1991). Another early robot system dedicated for airframe assembly was the Adaptive Robotized Multifunction Assembly (ARMA) cell developed by the robotic division of Dassault Aviation in 1993 for the assembly of Rafale and Falcon panels (Figure 3-1). The cell was based around two Fanuc S420 robots working synchronously from both sides of the jig; one robot mainly performed clamping, drilling, countersinking, applying sealant and inserting rivets whereas

the other was used for clamping from the other side and detecting the installed rivets. Before the assembly process took place, the two robots manipulated their end-effectors with an array of sensors to contact some cubes located at the extremities of the jig in order to calibrate the position of the jig relatively to the robot bases. During the process, the drilling robot used a vision system with an accuracy of 0.1mm for finding pilot holes on the template to adjust its programmed points (Da Costa, 1996). It can be seen from these examples that the use of drilling templates and local sensors can overcome the inherent inaccuracy of the robots to some extent, but the full potential of jigless manufacture was not realized. For example, around 30 of these templates were required to be manually mounted in position for the robotic assembly of each A330/340 FAL airplane wings. This is a laborious process given that each template weighs about 45kg (Kochan, 1991).

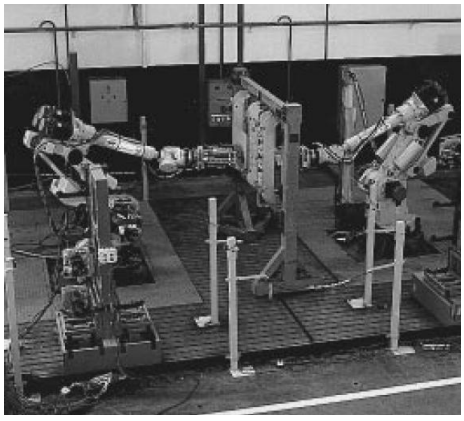


Figure 3-1 The ARMA cell (Da Costa, 1996)

In more recent applications, large volume metrology has been utilized as guiding sources for the robots used. The first robot system incorporated with metrology probably is the TI^2, whose name is derived from using a Tricept robot, a photogrammetric system from Imetric SA and the IGRIP offline programming software (Figure 3-2a). The Tricept robot, produced by Neo Robotics, is a hybrid parallel robot having a parallel tripod-like structure with a spherical wrist commonly found in serial manipulators. The accuracy of a TI^2 robot is 0.2mm, enhanced by multiple cameras tracking LED targets attached on the wrist. Boeing is the first user to use TI^2 systems in producing floor beams

for their 737, 747 and 767 airplanes (Staff, 2000; Fayad, 2002). Within the project Automated Wing Box Assembly (AWBA), a collaboration between seven UK companies including BAE and Airbus UK in the early 2000s, industrial robots are employed for rib loading and drilling/fastening of skin panels into the ribs. In the first application, one Kuka robot has to position the rib between the lower (trailing edge) and upper (leading edge) spars of a wing box. A Leica LTD500 laser tracker is used to measure the positions of the spars and guide the robot within an accuracy of ±0.5mm. In the second application, the fastening robot is equipped with a multifunctional end-effector, including a vision system, high speed spindle drilling head and stud inserter. Before skin wrapping takes place, the vision system is used to locate the position of each rib pad (Figure 3-2b). The position is then recorded in the memory so that the robot knows exactly where to drill through the skin and the pad once the skin has been placed. Drilling and fastening is done within a cycle time of 15s per hole (Rooks, 2001; Hemsteads *et.al.*, 2001). Kuka, Airbus UK and Metris (now Nikon Metrology) along with other 51 companies also participated in the recent Advanced Low Cost Aircraft Structure (ALCAS) project, a €100 million European Commission (EC) funded research program that aims to identify new composite manufacturing and assembly strategies. In the project, Kuka was in charge of developing a robotics system for the horizontal assembly of composite wings to replace the conventional vertical method. Two of the robots carried the photogrammetric K-series CMMs of Nikon Metrology for monitoring the heads of another two robots, which drilled holes from 6 to 22mm diameters and up to 100mm depth by using orbital drilling. With these enhanced systems, absolute accuracy of better than 0.1mm is easily achievable over working volumes of several meters (ALCAS, 2012; Richards, 2010; Calder, 2011). Most recently, a Volumetric Robotic Cell has been developed and currently is under final test before being put into the assembly line of nacelle systems for the newest Airbus A350 XWB in Toulouse (Goodrich, 2011).

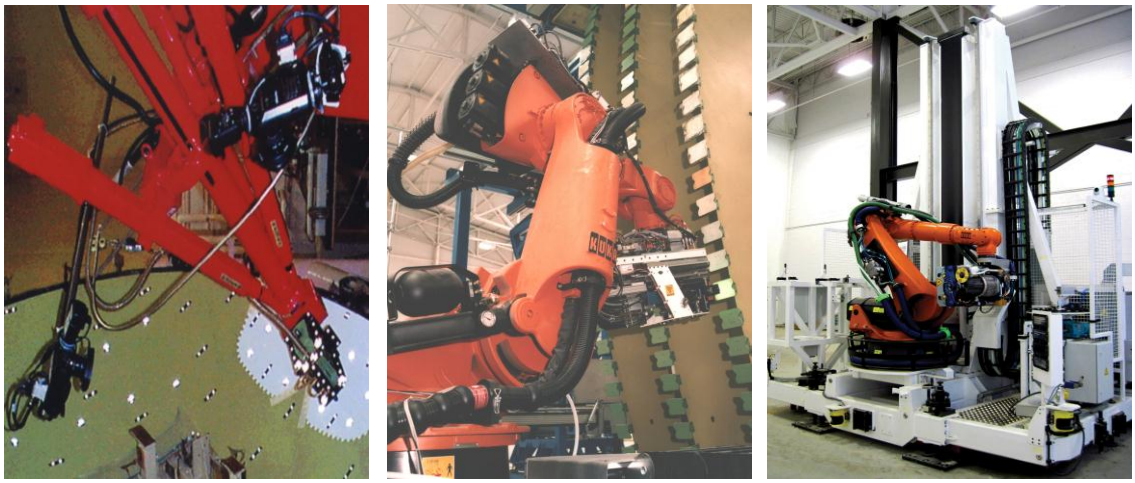


Figure 3-2 Recent applications of robots in airframe assembly. From left to right: a. The TI^2 system at Boeing (INS-News, 1998); b. Robot measuring rib pads in the AWBA project (Hemsptead *et.al.*, 2001); c. Robot for the assembly of fuselage sections of C-series aircraft at Bombardier (Arnone, 2011)

A number of robotic applications at Boeing are also found in the literature; most of them are subcontracted to automation solution suppliers. In the early 2000s, ElectroImpact developed the ONCE (One Side Cell End-Effector) system to drill, countersink and measure fastener holes in the trailing edge flaps on the Boeing F/A-18E/F Super Hornet. The end-effector of this robot system, shown previously in Figure 2-1, has multiple machining tools, hole probe and resynchronization camera to align the system to a datum target after each fixture rotation (DeVlieg, 2002). Using this multi-functional end-effector, other robot systems are developed for the assembly of 737 ailerons and 787 Dreamliner trailing edges (Atkinson *et.al.*, 2007; DeVlieg, 2008). Another company, Spirit AeroSystems in Kansas, USA, currently uses robots on several product lines of Boeing, including 787 fuselage, pylon and wing structures; 737 fuselage and thrust reverser components (Calder, 2011). Lately, Bombardier has also adopted six industrial robots in the assembly of C-series aircrafts' fuselage sections at their Saint-Lauren Manufacturing Center in Montreal. The robots are able to extend to a full height of 5.72m by using vertical lifts to reach the top and bottom of the aircraft (Figure 3-2c). Each robot can drill then

precisely rivet or hammer a fastener in 32 seconds and with the accuracy of one hundredth of an inch (0.254mm), enhanced by laser trackers (Arnone, 2011). VRSI, an automation solution supplier based in the US, has successfully applied robots for drilling inlet ducts in the F-35 Lightning II center fuselage at Northrop Grumman. The system employs DELMIA OLP software for process simulation, a Fanuc series 2000/125 robot for the drilling operations and a vision system for verifying the quality of each drilled hole. Previously, they used photogrammetric and hybrid systems for correcting the tool positions but they found only a laser tracker, provided by FARO, could maintain a high accuracy of 0.01 inch over a large volume (Costlow, 2009; Grasson, 2011).

There also exist several academic researches toward the use of robotics in airframe assembly. At the Robotics Research Group at the University of Nottingham, UK, a number of robotic applications on actual aerospace parts were carried out in order to evaluate the capabilities of metrology integrated robot systems for the assembly tasks. For example, Eastwood *et.al.* (2003) and Webb *et.al.* (2004) investigated the accuracy of the TI^2 system in drilling and milling of aerospace panel, rib and spar structures of the Bombardier Lear 45 business jet and Airbus A320. It was concluded that while the Tricept robot had sufficient repeatability and stiffness to perform machining, the obtained results relied on how accurately the transformations between the robot and the parts were determined, in these cases, by the photogrammetric cameras of the TI^2 system. Later on, a flexible robotic cell was developed (Webb *et.al.*, 2005). The cell was based around three robots working simultaneously: a Comau S2 for loading stringers to the skin panel, a Comau Smart H4 for drilling, countersinking and installing solid rivets onto the panel and the Tricept robot opposite to the H4 for creating the reaction force (Figure 3-3a). The S2 robot carried seam tracking sensors capable of detecting and measuring edge and pre-drilled holes on the stringer and on the panel, from which coordinate frames were constructed (Figure 3-3b). These frames served as the targets for the robot to pick up the stringer then attach the stringer to the panel. Since both the parts may contain distortions and misalignments, the frames were not built directly on the measured features but rather on the best-fit geometries of these

features and hence, deviations of the part were partly compensated. With this so-called adaptive assembly methodology, the robot was capable to do the assembly within the tolerance of ±0.8mm, an encouraging result considering that accuracy of the sensors used were worse than ±0.3mm (Jayaweera *et.al*., 2007). A cell control application was developed to control and coordinate the robots, sensors, end-effectors and Matlab software used during the best-fit construction (Figure 3-3c). It is a typical example of a centralized control unit with all the control and processing functions of relevant devices and processes concentrated on one computer.

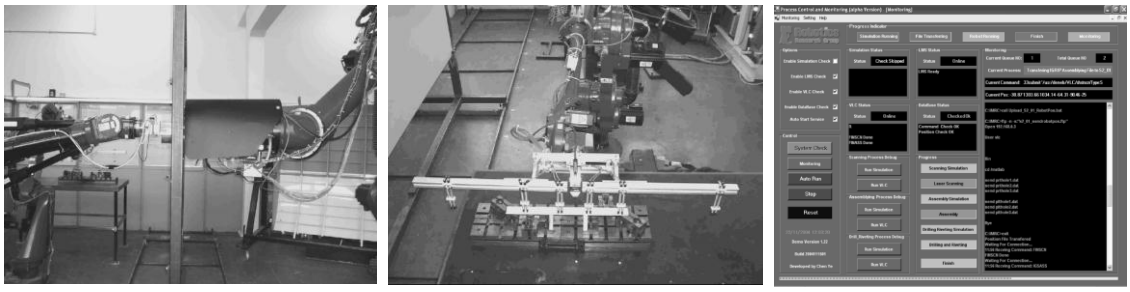


Figure 3-3 The flexible robotic cell developed by the University of Nottingham. From left to right: a. The Smart H4 and Tricept robots for drilling and fastening; b. The S2 robot for stringer loading; c. The cell control application.

To the author's opinion, one of the most outstanding academic researches in the area probably is the work presented in Henrik Kihlman's PhD thesis "Affordable Automation for Airframe Assembly" at Linköping University, Sweden (Kihlman, 2005). The affordable automation solution developed by the author covers five major areas: robotics, drilling, tooling, metrology and operation planning. In this work, an ABB IRB 4400 robot is equipped with a 6D TMAC reflector (section 2.3.1.1), allowing for the robot to be tracked by a Leica LTD800 laser tracker. The robot, with aid of the metrology system, is used to configure the location and orientation of other passive tripods and hexapods acting as flexible tooling actuators to give the part its specific localization. This is one of the main concepts of a proposed Affordable Reconfigurable Tooling (ART) framework, which is also based on steel bars bolted together by modular

box joints, rather than welded, to create its surrounding structure. Having reconfigurable components, this novel fixture can be rebuilt easily on demand when it has to be redeployed for different products (Figure 3-4).

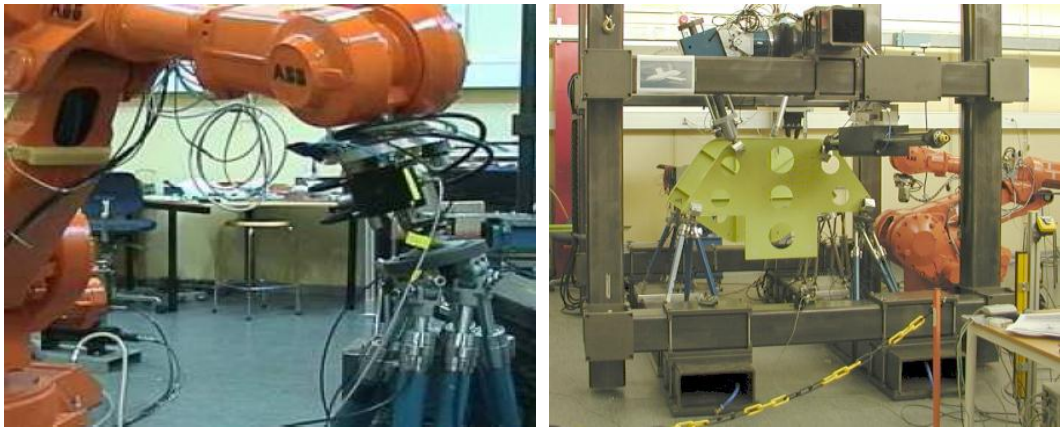


Figure 3-4 Overview of the ART concept. From left to right: a. The robot configures a flexible tooling; b. A system of tooling is used for holding an aircraft part in-position during assembly (Kihlman, 2005)

Later on, the robot is used as a drilling machine of which the tool tip is guided by the laser tracker to programmed locations. Calibrations were carried out to determine transformations between different coordinate frames: from the laser tracker base to the robot base, from the TMAC to the TCP and from the TCP to tool tip. Orbital drilling, a circular milling-like drilling technique, is employed in order to minimize the axial thrust force which tends to cause dynamic errors to the robot (section 2.2.5). The whole assembly cell is modelled and the operation processes are planned in DELMIA. In this OLP software, positions of the end-effector with different operations (configuring the tooling or drilling) and different required accuracy, either with or without metrology correction, are assigned with different name tags. The generated offline program, in the form of a readable text file, is then input to a control application which replaces these name tags with control commands of the robot, laser tracker and drill to be sent to their controllers for execution. This application is also a form of a centralized control unit (Figure 3-5).

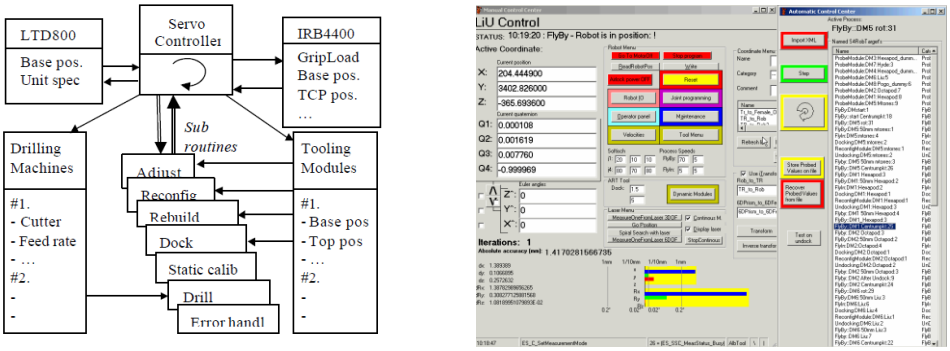


Figure 3-5 Control application of the robotic cell developed by Linköping University. From left to right: a. Functional diagram; b. Graphical interface (Kihlman, 2005).

3.2 Error compensation techniques

As previously presented in section 2.2.5, accuracy of a robotic assembly system depends on those of the robot and the part. An overview of error compensation techniques for improving system accuracy has already been introduced in section 2.3.3.2. This section will cover existing techniques in the literature.

3.2.1 Part localization

The general idea of this type of correction is that the robot uses a sensor to measure some features on the part to determine the User frame. This replaces the nominal one pre-defined in the robot program, allowing the robot to compensate for positional error of the part due to misalignment or distortion. This static correction approach has been adopted in the works of (Da Costa, 1996; Jayweera *et.al.*, 2007) presented in part 3.1 above. Another example of this method is the work of (Bone *et.al.*, 2003). The robot of this “fixtureless robotic assembly” cell carries a CCD camera used for capturing an image of the part before the assembly takes place (Figure 3-6a). The image is processed by a commercial software package which detects the edge contours of the part. The orientation components of the User frame were calculated from these edge contours (Figure 3-6b) while the position components were determined by another range sensor (not shown in the figure). Using this vision-guided method, assembly accuracy of ±2mm was achieved.

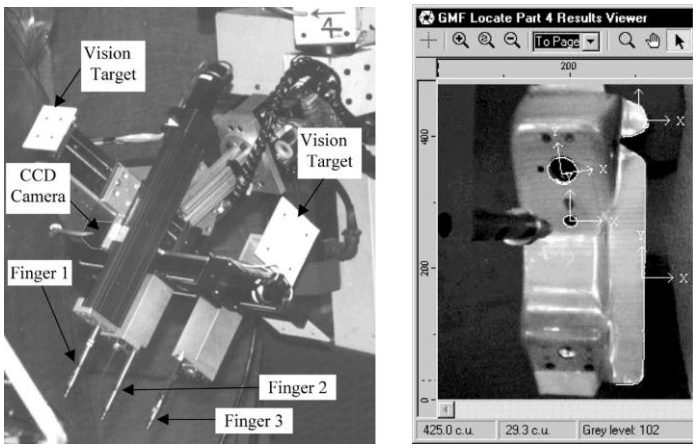


Figure 3-6 A fixtureless robotic assembly cell. From left to right: a. The robot end-effector with CCD camera; b. Detected features (Bone *et.al.*, 2003)

3.2.2 Robot positioning accuracy

Robot accuracy can be improved by model-based or sensor-based error compensation (section 2.3.3.2). Model-based error compensation employs error models to compensate for sources of inaccuracy in the robot structure. These error compensation models are developed either based on error mapping techniques, such as polynomial, bilinear, cubic spline or fuzzy interpolations (Eastwood, *et.al.*, 2010; Bai *et.al.*, 2004; Bai *et.al.*, 2005) or by systematic modelling and identification of error sources in the robot structure, which is well known as the robot calibration technique.

3.2.2.1 Robot calibration

As briefly introduced in section 2.2.5, a robot calibration procedure involves three steps: (1) modelling: developing a model relating geometric and non-geometric errors in kinematic parameters to be identified with tool pose errors, (2) measurement: measuring the tool pose errors with an external sensor and (3) identification: solving the developed model for the errors in kinematic parameters. The choice of the error model which contains all identifiable parameters is the most important step in the process. With regards to the geometric errors, various error models were already introduced. Several models

were derived from the well known Denavit - Hartenberg (DH) convention while the others were specially developed for the purpose of calibration (see e.g. (Mooring and Tang, 1984; Hayati *et.al.*, 1985; Hsu *et.al.*, 1985; Veitschegger *et.al.*, 1987; Driels *et.al.*, 1987; Stone *et.al.*, 1987; Zhuang, 1993)). Nevertheless, the four parameter DH model with Hayati's modification for handling the cases of successive parallel joint axes (Hayati *et.al.*, 1985) has been widely accepted and become the most popular convention in calibration of geometric errors owing to its 'user-friendly' form. Schröer *et.al.* (1997) further proposed that this combined model is only a subset of their DH-based "complete, minimal and continuous" kinematic models. The contribution of their research is a systematic rule for setting up DH/Hayati frames for different types and configurations of joints in open-loop robot structures, e.g., the elbow type manipulators. Once followed, the work of geometric error modelling of these manipulators becomes formulaic. It has been shown that the major sources of errors in a robot structure are joint offsets and misalignment of joint axes (Mooring *et.al.* 1989; Judd *et.al*., 1990; Bernhardt *et.al.*, 1993). By compensating for these errors, robot accuracy can be improved up to 1mm, depending on the sizes of the robots and the accuracy of measuring systems used.

For further error reduction, below 1mm, it is necessary to take into account other non-geometric effects, i.e. elastic deformations of joints and links induced by link weight (Judd *et.al.*, 1990; Schröer *et.al*., 1997; Drouet *et.al.*, 2005; Gong *et.al.*, 2000), thermal errors (Gong *et.al.*, 2000), nonlinearity and backlash in gear and drive train (Schröer *et.al*., 1997), gear run-out (eccentricity), gear orientation errors (Renders *et.al.*, 1991) etc. A comprehensive survey of developed models for these types of errors can be found in the review of (Karan *et.al.*, 1994). In contrast to geometric error modelling which has been somewhat standardized, non-geometric error modelling unfortunately varies from one researcher's point of view to another. The reasons are these errors are not ever-present in all manipulators and hard to model precisely, especially the backlash (Karan *et.al.*, 1994). Only a few agreements were made, such as link flexibility usually is less than joint flexibility: less than 20% of total flexibility,

thermal effects might cause more errors to the measuring system than the robot structure (Renders *et.al.*, 1991; DeVlieg, 2010). It is therefore necessary to solve for non-geometric errors on case by case basis. With both geometric and non-geometric errors, the best calibration result for long reach (over 2.5m) and heavy duty robots reported in the literature is in the order of 0.5mm (Schröer *et.al*., 1997). CalibWare, an optional calibration package to purchase with ABB robots, also offers similar results, with the average accuracy of 0.52mm and maximum of 1.2mm (ABB, 2010).

3.2.2.2 Sensor-based correction

In this technique, the robot is guided by a global sensor to its target locations. In Kihlman's thesis, absolute accuracy of the ABB's IRB 4400 robot is corrected by iteratively moving the robot and evaluating the errors between its programmed and actual 6D locations measured by the Leica's LTD800 laser tracker (Figure 3-7). The process is terminated, typically after 6-10 seconds, when the errors are below 0.05mm in translation and 0.0005rad (0.03°) in orientation. It was pointed out that to meet such small threshold, smaller than the repeatability of the robot used (0.07-0.1mm), the robot's resolution in translation (5μm) plays the main role. This static correction method achieved an accuracy of 0.1mm throughout the entire working range (Kihlman, 2005).

The so-called Adaptive Robot Control of Nikon Metrology is another similar technique in which photogrammetric K-series Optical CMMs are used instead of laser trackers. The method is quoted by the company as a "real-time continuous corrective adaptation" for high precision robotic drilling, milling and mould and die applications (Nikon Metrology, 2011). However, in a private conversation with the author within the Large Volume Metrology conference in 2011, R. Holden, director of the company's centre for Metrology Integrated Robotics, revealed that the technique by far has still been a "move then measure" method and thus, mostly suitable for quasi-static (e.g., drilling) applications. For milling, it is necessary to define along the path several intermediate points at which the correction will take place (Holden, 2010). Currently, researchers at the centre

are striving to make the technique a true dynamic correction within the EC project COMET (Plug-and-produce COmponents and METhods for adaptive control of industrial robots enabling cost effective, high precision manufacturing in factories of the future) (COMET Project, 2012).

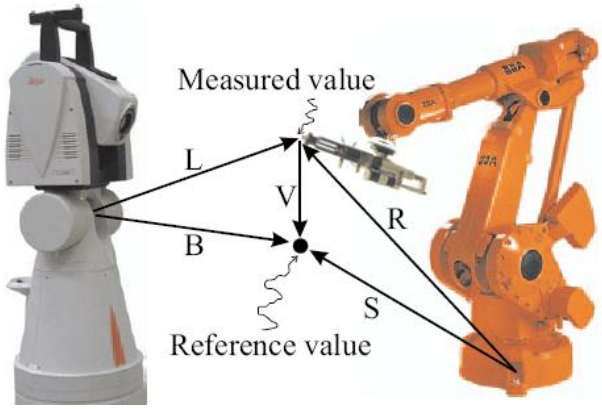


Figure 3-7 Sensor-based correction is for gradually reducing the 6D error vector V between the programmed B and measured L (Kihlman, 2005)

Dynamic positioning correction obviously is more desirable in terms of time efficiency but presents a lot of challenges in practice. To the author's knowledge, they include:

- Measurement uncertainty. For example, a laser tracker's single measurement taken in one second actually is the average of 3000 samples processed internally to cancel out the effects of noises and thermal drift (Leica, 2008). High measuring rate in dynamic correction, therefore, will result in higher uncertainty, which includes that of the optical system and oscillation of the robot when moving. In the author's experience when using a Leica AT901 laser tracker, discrepancies between the measurements of a static target taken in short periods (<10ms) and longer periods (>1s) are not always better than 0.05mm whereas oscillation of the robot might degrade the result further.
- High frequency data update between the metrology and control application to the robot controller. Despite a Leica laser tracker is able to measure up to 1KHz, it transmits the data over the network in packets of 10 measurements and hence, its practical sampling rate is only 100Hz.

In addition, dynamic correction requires direct access to the internal architecture of a robot controller, which is not always opened to all users due to proprietary and safety issues (Kihlman, 2005). This problem will be described further in section 3.3.1.

A successful case study of dynamic correction for robot positional accuracy is reported in the on-going EC project ARFLEX (Adaptive Robots for Flexible Manufacturing Systems). In the project, a system of fixed cameras is used for visual servoing of a robot with an update rate of 100Hz (Figure 3-8). Positioning accuracy of the robot system is claimed to be 0.1mm (ARFLEX Project, 2012).

Figure 3-8 The visual-servoing demonstrator of the ARFLEX project (ARFLEX, 2011)

3.2.3 Deflections in drilling

Two existing techniques for minimizing or correction of deflections of the robot structure induced by dynamic thrust force in drilling are the orbital drilling and sensor feedback drilling. Orbital drilling is a novel drilling technique, in which the drive spindle rotates eccentrically in addition to tool rotation and feed movement, leading to a circular path of the cutting tool. Compared with conventional drilling, it significantly reduces the thrust force and is possible to compensate for tool diameter deviations. Orbital drilling is demonstrated in the ALCAS project and already applied in many Airbus sites. Nevertheless, the

technique also faces many challenges, such as tool bending when drilling high thickness and vibrations (Kihlman, 2005; Deitert, 2011).

Sensor feedback drilling is a conventional drilling technique enhanced by a dynamic correction of the robot's deflections in real-time. For example, based on force feedback from a wrist F/T sensor, a force control algorithm will calculate a small change in position of the manipulator in order to generate and maintain the clamp force orthogonal to the part surface while suppressing other tangential components that cause slipping of the tool tip (Alici, 1999; Ple *et.al.*, 2011). The resulting position change is added to the reference position of the inner control loop in the robot controller, as previously depicted in Figure 2-21. By deploying such a force control scheme, Olsson *et.al.* (2010) have managed to reduce the tangential deformations from 1.6mm to below ±0.3mm within the robot workspace. Another approach is utilizing high resolution encoders mounted at the arm side of joint axes (Figure 3-9). These secondary encoders are used to measure joint deflections, the majority of deflection in the robot structure, through which deviation of the tool tip during drilling can be compensated. DeVlieg (2010) at ElectroImpact stated that with this patent pending solution and robot calibration, robot systems produced by the company are able to drill with positional accuracy better than ±0.25mm. Particularly, when the robots are guided by a laser tracker, they are able to achieve the accuracy of ±0.08mm, a remarkable result.

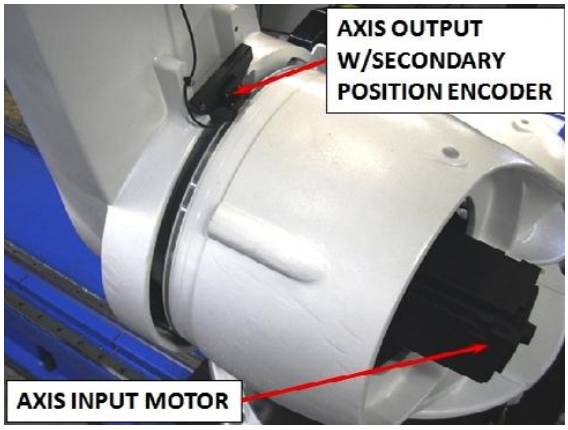


Figure 3-9 Robot axis with secondary encoder for deflection compensation (DeVlieg, 2010)

3.3 System integration

3.3.1 Direct communication and centralized control for dynamic correction

As introduced in section 2.3.3.2, dynamic correction (e.g., closed-loop position or force control) must be implemented at low level in such a way that the control set points are fed into the robot controller within its interpolation cycle, which typically is 4ms. This usually requires high speed, point-to-point connections between the sensor, control PC and robot controller as well as open access to the robot controller's internal architectures. In most force control applications (grinding, deburring, polishing and drilling etc.), the connection between the F/T sensor and the control PC is via PCI bus which is roughly ten times faster than the Ethernet whereas that between the control PC and the robot controller is either via a dedicated fieldbus or the PCI bus as well. This approach is realized from the fact that modern robot controllers, e.g., the S4 and IRC5 of ABB robots, C4G, C5G of Comau robots and KRC4 of Kuka robots, are just Intel PCs having several open PCI slots which can be used for additional periphery (ABB, 2010; KUKA, 2008). Via the PCI bus, these robot controllers allow access to the shared memory interfaces of their inner control loops, i.e., the trajectory generation (at 250Hz) or even the servo control (at 1-2kHz) via their APIs. Examples of such low level APIs are the Fast Research Interface of Kuka for their LWR (Light Weight Robot) series, the C4GOPEN of Tecnospazio *s.p.a* for Comau robots running on C4G controllers and the RCAL (Robot Controller Abstraction Layer) library of Stäubli for their RX, TX robot series. These features are exploited in many researches: the control PC and F/T sensor are connected directly to these of the PCI slots and share the same bus (Figure 3-10); the set points (joint positions/velocities) calculated by the control PC will overwrite the original values in these shared memory addresses within the 4ms time frame (Blomdel *et.al.*,2005; Pires *et.al.*, 2006; Garcia *et.al.*, 2009; Antonelli *et.al.*, 2010). The capability of accessing the shared memory interface, however, is not granted to all end-users due to proprietary and safety issues, explaining why researches on force control are mostly undertaken by Swedish,

Italian and German researchers who have close collaborations with their robot manufacturers. Nevertheless, it is a must when one wants to implement custom control at low level (Kröger *et.al.*, 2008; Pedrocchi *et.al.*, 2010).

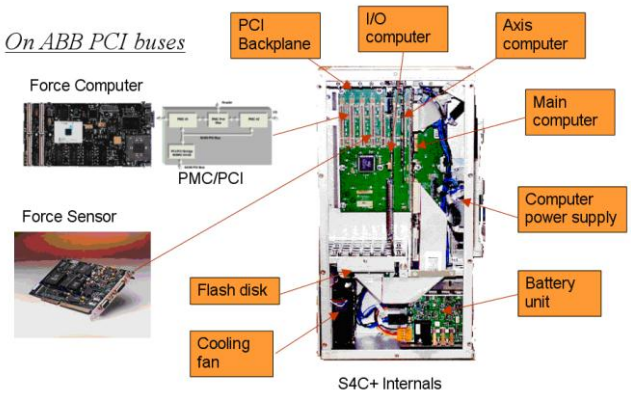


Figure 3-10 Integration between the Force Sensor, the control PC (Force Computer) and ABB S4C+ robot controller for force-control application via PCI bus (Blomdell *et.al.*, 2005).

3.3.2 Distributed control

This section firstly introduces well known middleware technologies including the ones used in distributed manufacturing systems in general as well as those developed for robotics in particular. Finally, it addresses common attributes of distributed control frameworks/systems through existing research in the area.

3.3.2.1 Communication middleware

CORBA (Common Object Request Broker Architecture) is a standard open architecture developed to integrate distributed applications by Object Management Group (OMG), a non-profit organization participated in by 700 companies and vendors. The core component of CORBA is the ORB (Object Request Broker), the middleware for integrating applications on heterogeneous operating systems (OS) and in different programming languages including C/C++, Java, COBOL and Python. To achieve language independence, CORBA requires developers to express how clients will make requests to a service using a standard and neutral language: the OMG Interface Definition

Language (IDL), a C++ syntax-like language. After the interface has been defined in IDL, an IDL compiler generates client stubs and server skeletons according to the chosen programming language and operating system (OMG, 2011). The ORB will be in charge of the communication between the client and server applications via their stub and skeleton, as depicted previously in Figure 2.27. For robotics and process automation, the real-time ORB of CORBA (RT-CORBA) is mostly used. RT-CORBA was implemented under The ACE ORB (acronym: TAO), an Open Source project founded at Washington University. TAO supports various OS platforms including Linux, Windows and Solaris (Schmidt *et.al.*, 2010). A comprehensive overview on the use of CORBA for control systems is given in (Sanz *et.al.*, 2001).

DCOM (Distributed Component Object Model) is Microsoft's solution for distributed, object-oriented applications in client-server architecture. In DCOM, a server computer contains one or more component objects, each of which may serve several services. Similar to CORBA, the structure of the component objects, their interfaces, methods, and parameters is also defined in an IDL file, which describes the contract between a client and server. To start accessing methods at interfaces of a server's component object, the client program firstly requests the Service Control Manager (SCM), a part of Windows, to create an instance of the object on the server computer. Once the remote COM object has been created, all further message exchange will be handled by the RPC stubs of the object and the client as already known. As a Microsoft proprietary technology, DCOM only runs on Windows OS and supports C++, C#, Visual Basic and Java programming languages (Rubin *et.al.*, 1999).

Web Services can be thought as a new RPC architecture introduced to overcome limitations of CORBA and DCOM. The problem with CORBA and DCOM is that each vendor uses different standards for data serialization and transmission protocol and hence, they have compatibility issues (Schmelzer *et.al.*, 2002; Hochgurtel, 2003). Web Services encapsulates the RPC using the standardized SOAP (Simple Object Access Protocol) for data serialization. As opposed to CORBA and DCOM which use binary data format and wire

protocols, SOAP uses XML (eXtensible Markup Language), a human readable document, as the neutral data format and HTTP (Hypertext Transfer Protocol) as the data transmission protocol between its distributed services. In addition, the WSDL (Web Services Description Language) used for defining interfaces of the services is also based on XML. These technologies (XML, HTTP) have already been well-defined for Internet communication, what makes it easy for Web Services to gain interoperability among distributed systems over both local and wide area networks as well as platform and programming language independence. Web Services, however, are mostly suitable for plant information management between cell with shop levels and between computers at shop level with the outside world, in which data usually are highly structured, in large amount but the data transfer time is not critical (Hu *et.al.*, 2007). At lower levels, Web Services' message exchange rate using SOAP can be considerably slower than other binary-based protocols due to the verbose XML format (Amoretti *et.al.,* 2006).

3.3.2.2 Robotic middleware

A number of middleware platforms have been developed for robotic applications, mostly by university research groups. They typically include a RPC as the core component and other value-added components (modules/libraries/classes) helpful for developing robotic applications.

RDS (Robotics Developer Studio) is the middleware for distributed robotic applications developed by Microsoft since 2004. RDS runs on Windows OS and supports .NET programming languages including C++, C# and Visual Basic. It consists of a number of software modules, including the two most important:

- DSS (Decentralized Software Services protocol): a light weight SOAP-based RPC platform. Unlike in Web Services, the SOAP in DSS uses binary serialization and TCP/IP as the transmission protocol in order to attain a higher communication rate for robotic applications.
- CCR (Concurrency and Coordination Runtime): an event-based programming model for handling concurrency and inter-task synchronization commonly encountered in robotics.

In addition, RDS also provides additional modules, such as the Visual Programming Language to create the composite service without the need for serious coding and Visual Simulation Environment for realistic on-line simulation of interactions between robots with the surrounding environment (Johns *et.al.*, 2008).

MiRPA (Middleware for Robotic and Process Automation) is a distributed real-time middleware developed by the Institute for Robotics and Process Control at TU Braunschweig, Germany. The greatest advantages of MiRPA are its very high update rate, 1kHz, and low latencies, around 10µs for local communication (when software modules reside on the same computer) and less than 100µs in a distributed system. Owing to its high performance, MiRPA is suitable for high-rate low-level control of robot manipulators, where a distributed control system powered by the MiRPA API can replace a centralized controller with point-to-point connections. It has been used for the integration of a force sensor and haptic device into a Stäubi's RX series robot, in which MIRPA is the communication layer between a control PC with a sensor-based control algorithm and the CS7 robot controller for exchanging set points within the 4ms cycle. The main drawback of the MiRPA is its reliance on the QNX, a light weight real-time operating system (RTOS), to achieve its performance. The availability of device drivers and engineering tools such as programming environment and computing software necessary for developing complex applications on this unpopular OS might be an issue. Indeed, the authors of MiRPA experienced this problem when there was no driver for the haptic device used and they had to develop it themselves (Kunbus *et.al.*, 2010).

OROCOS (Open Robot Control Software) is an Open Source C++ software framework developed by the University of Leuven, Belgium for building component-based applications in automation and robotics. OROCOS is composed of three main components: a) Real-time Toolkit which is a RPC platform based on RT-CORBA running on RTAI, a Linux-based RTOS; b) Kinematic and Dynamic Library for numerical computation of kinematics and dynamics of serial robot manipulators and c) Bayesian Filtering Library for

sensor fusion (Bruyninckx, 2001). It is stated that integration of the K-series optical CMM with a Kuka robot controller via OROCOS was successfully demonstrated at 500Hz update rate. The result of improved robot accuracy using this metrology system, however, was not given (OROCOS project, 2011).

ORiN (Open Robot/Resource interface for the Network), developed by JARA (the Japan Robot Association) in collaboration with 13 Japanese robot manufacturers, is another middleware platform for accessing information in robots, devices and equipments used in factory automation. In the ORiN context, robots from different vendors are accessed via their services, namely the RAO (Robot Access Object). ORiN is based on DCOM but uses SOAP as message transport protocol over network. ORiN operates on Windows OS and supports Microsoft Visual C++ and Basic programming languages (Mizukawa *et.al.*, 2004).

3.3.2.3 Existing distributed control frameworks/systems and their features

A large number of distributed control frameworks/systems have been reported in the literature. CORBA is the most commonly used middleware for developing distributed robotic systems, owing to its support for several OS and real-time capabilities. Song *et.al.* (2007) developed a test-bed for a robotic train maintenance system in which the robots perform disassembly of parts, replacing the worn components and re-assembling the parts back together. RT-CORBA (TAO) is used as the middleware connecting robot managers (services), main servers (cell controllers) and other client applications as depicted in Figure 3-11. In the work, the authors have pointed out several advantages of the developed distributed system including the interoperability and flexibility, thanks to the separation of interfaces from implementation. When robots are added into or removed from the system, the main server only needs to add or remove the corresponding robot managers without affecting the system's high-level conceptual service design and implementations. When the main server is ported to a new hardware server or different OS, there is no need to recompile low level robot applications in a new environment. Similar works and conclusions can be found in (Paolini *et.al.*, 1997; Jia *et.al.*, 2008; Song

et.al., 2003), where CORBA was employed as the solution for integration of multi-vendor robots and sensors having APIs provided in various programming languages and various operating systems, or for teleoperation of robotic work-cells (Tu *et.al.*, 2005). Reports on the use of other middleware technologies (e.g., DCOM, Web Services etc.) for distributed manufacturing systems, though less popular, are also found in literature. For example, DCOM was used to develop distributed robotic manufacturing cells (Pires *et.al.*, 2000) and open architecture robot controllers (Hong *et.al.*, 2001; Short *et.al.*, 2011) whereas Web Services were used for the interconnection of relevant workstations in a semiconductor manufacturing plant (Hung *et.al.*, 2010).

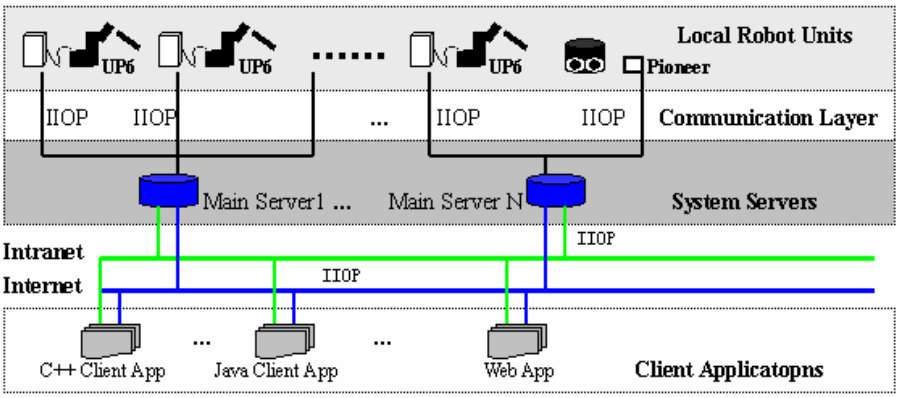


Figure 3-11 System layout of a robot system for train maintenance (Song *et.al.*, 2007)

Through research in the literature, several essential features of modular, distributed control systems from a software engineering point of view have been outlined. In the work of (Amoretti *et.al.*, 2006), the authors demonstrated a system in which a robot serves several client applications and thus, it must respond to multiple requests from the clients at the same time. Commands for querying the robot's status, e.g., its instant position, can be performed in parallel whereas other commands involving motions must be performed successively. Therefore, **concurrency** (multi-tasking) in server operations is needed, however, a synchronization mechanism among the threads must be used for the latter case: a client must acquire a software lock from the robot in order to gain its exclusive control while the others must wait until the lock is released.

This is the traditional **locking mechanism** to resolve simultaneous requests to the shared resource commonly found in concurrent programming. In addition, **asynchronous communication** (non-blocking) between client and server is also desirable. In contrast to synchronous communication which halts operation of the client until it receives response from the server, asynchronous communication allows for the client to perform other tasks while waiting for the response and resume its execution once the message arrives. Colon *et.al.*, (2005) also specified that distributed control systems would support not only **request/response** but **publish/subscribe** communication mechanisms. Request/response is the typical bidirectional communication used in client/server architecture: the client firstly invokes a function at the server then the server, as a result, returns the data back to the client, either synchronously or asynchronously. On the other hand, the publish/subscribe is a unidirectional communication: the publisher (e.g., a touch probe) notifies the data to a group of subscribers (e.g., a robot) either automatically upon an event (e.g., touching a surface) or on demand when a subscriber asks for updates. As opposed to the tightly-coupled and one-to-one request/response model, the publish/subscribe model exhibits a loosely coupled mechanism: the publisher does not need to be aware of the subscriber presence and it can be used for one-to-many and many-to-many (peer-to-peer) communications (Lee, 2007). In CORBA, the aforementioned concurrency, locking mechanism, asynchronous and publish/subscribe communications are provided by separate modules Concurrency Control Service and Asynchronous Method Invocation.

Real-time capability, i.e., meeting deadlines for data transmission, is another desirable feature of distributed control systems and for this reason, the RT-CORBA (TAO) is usually selected as the middleware used. Interestingly, all the frameworks/systems cited above, though claimed to be real-time capable, only demonstrated applications of which the real-time requirements are not critical, e.g., sensor-based robot control in static mode or robotics in factory automation. In fact, achieving true real-time determinism is difficult since it demands not only a RTOS but a real-time transmission protocol and is complicated by the requirements of high sampling rate and low latency in dynamic correction. Many

middleware platforms, including the RT-CORBA, rest upon a Linux-based RTOS (e.g., RTAI or XENOMAI) to achieve this capability. However, Kröger *et.al.* (2008) have pointed out that these monolithic systems cannot guarantee the worse case latencies due to the problem of priority inversion in inter-node communications. In addition, CORBA and CORBA-based robotic middleware such as the OROCOS is built upon TCP/IP, which is rather more suited for transmission of long messages over long distances than high rate and short messages either (Pan, 2011). It is suggested that in order to implement distributed control at low level, middleware built on a microkernel-based OS (e.g., VxWorks, QNX) and UDP transmission protocol, such as the MiRPA, is a better choice (Kröger *et.al.*, 2008; Bäuml *et.al.*, 2008). However, whether it is convenient to develop complex robotic applications based on these middleware and OS, is still questionable, not to mention that UDP is an unreliable protocol (no packet loss recovery). Implementing real-time control via middleware is thus a challenge. That explains why in most low level control applications found in the literature, resorting to centralized control and point to point robot sensor integration (section 3.3.1) is still the dominating approach.

In order to achieve dynamic reconfigurability at runtime, distributed control systems should provide the "**Plug and Produce**" capability. The phrase Plug and Produce (PnP) is inspired from the concept of Plug and Play technology in Windows OS where a device (e.g., a printer, webcam) can be freely connected to or removed from a computer without requiring manual configuration. A PnP automation system, therefore, will allow for a machine/component to be brought into or withdrawn from production instantly without having to redesign (reprogram) the existing infrastructure, which causes disruption to the manufacturing process. However, this behaviour is quite difficult to achieve in practice, since without human knowledge, the system itself would not be able to know what functionality the new device offers and how to actually process its data (Pitzek *et.al.*, 2007). Indeed, existing researches toward PnP automation so far (e.g. (Deter, 2001; Naumann, 2007; Ahn *et.al.*, 2009; Pires *et.al.*, 2009)) are only able to solve the plug-ability of system components, that is, a new device joining the system is able to be automatically detected and advertises its

interface to other devices so that they can potentially use it. This is typically achieved by adopting a communication platform capable of automatic discovery of its heterogeneous services, such as the Universal Plug and Play (UPnP Forum, 2012). Another approach toward this feature is described in the new International Standard IEC 61499 (Hanisch *et.al.*, 2007; Vyatkin, 2009). The standard defines a homogeneous architectural design for function blocks having inputs and outputs which can be interconnected, or "plugged" together, to form a more complex software component or system (Figure 3-12). Nevertheless, programming (in a computer programming language) is still required in both approaches to define the execution semantics as well as to perform data conversion/transformation between different software components. True PnP capability, without user intervention, thus still remains an appealing vision. PnP automation currently is the research theme of several on-going EC-funded projects, including the SMERobot and POPJIM - Plug and Produce Joint Interface Modules (SMERobot Project, 2012; POPJIM Project, 2012).

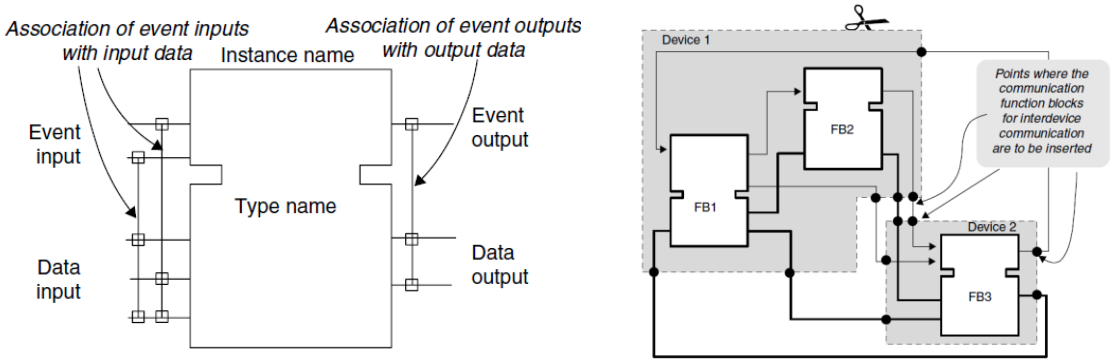


Figure 3-12 Brief description on the IEC 61499 standard. From left to right: a. A function block with standardized external interface; b. A distributed control application built on these functional blocks (Hanisch *et.al.*, 2007)

3.4 Discussions

The literature review described in this chapter has shown that there is great potential for a wider utilization of industrial robots in airframe assembly, especially when they are combined with metrology. Many of the existing robot systems augmented with guidance from an external sensor and localized

correction have been proved to meet the high dimensional accuracy requirements. However, their control systems usually exhibit a centralised architecture with strongly-coupled hardware/software which limits their use to a dedicated operation or product once delivered to the floor. Since aerospace manufacturing has low production rate but diversity of subassembly components, it is desirable to improve the flexibility of these robot systems such that they can be redeployed rapidly for different operations and product variants.

In order to achieve flexibility, the control architectures of these robot systems must be organized in modular, distributed manner. In such a system, manufacturing resources (robots, actuators, sensors, 3rd party software etc.) are controlled by separated software components, namely services, which can reside on different computers and are linked together by a middleware platform. Since the services are loosely-coupled via their interfaces, modification made to one component would not cause cascading changes to the whole system.

To provide the system with a further degree of flexibility, that is, dynamic reconfigurability at runtime, PnP integration should be supported. Ideally, it would allow the system to be reconfigured (e.g., components to be added to/withdrawn from the existing infrastructure) for different manufacturing processes without user intervention. However, it has been pointed out in section 3.3.2.3 that software modification is still required to define control activities between the components even though they have been made pluggable to each other. System programming still requires expert knowledge and hence, might cause significant delay to manufacturing activities. Improvements should be made so that reprogramming in such cases is easier and quicker.

Also outlined in section 3.3.2.3, a distributed robot control system should support both concurrency and synchronization in processing, asynchronous communication in request/response and publish/subscribe manners. On top of that, it must be able to facilitate several error correction and verification stages required in airframe assembly processes (i.e., measurement, part localisation, robot positioning and force control during drilling). Among them, it appears that

distributed control systems, due to limitations in current middleware technology, are not well suited for force control which requires high speed and low latency communication. For the components supposedly used in low level control and requiring strict real-time characteristics, resorting to direct communication is still necessary, rather than via a middleware platform. The other components, on the other hand, could be controlled by separate and pluggable services, making the system reconfigurable.

In addition to inflexibility, another disadvantage of current metrology-integrated robot systems in airframe assembly is cost ineffectiveness since one expensive piece of large volume metrology (e.g., a laser tracker) is used only for one inexpensive robot, which usually costs several times less. This is primarily due to the way the sensor is strongly-coupled in the robot control application and the feature of the laser tracker which tends to follow one target. Since airframe assembly mostly involves drilling or handling operations which require only static accuracy, the laser tracker can possibly serve more than one robot to reduce the cost of investment. This approach is feasible by exploiting the advantage of the ADM (section 2.3.1.1) to unlock the laser beam selectively from one robot and point to another.

The work presented in this thesis attempts to fill the gaps discussed above. To improve the flexibility of robots systems used in airframe assembly, an application framework for developing distributed, service-based control systems in a PnP manner is introduced (Figure 3-13). Reprogramming is still needed when the control system is reconfigured for a new manufacturing process to define the execution semantics between services but it will be done in robot programming languages by technicians on the floor, who only need to write robot programs, instead of computer programs, to develop new applications. Complex and time-consuming system programming is not required.

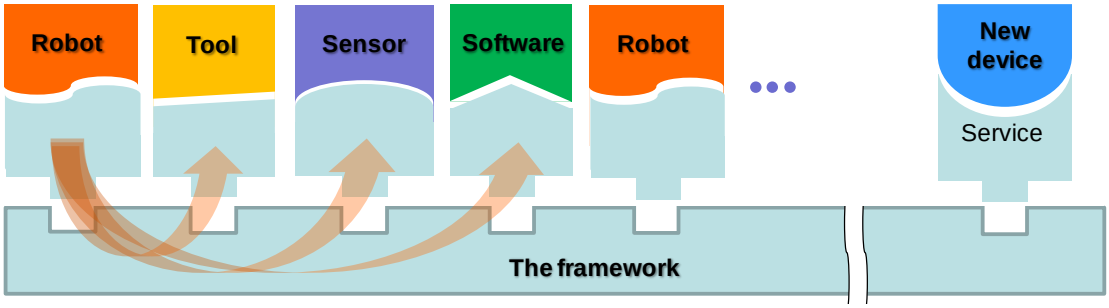


Figure 3-13 Concept of the proposed framework for PnP integration

To reduce the cost of investment for a global metrology source (e.g., a laser tracker), this thesis proposes a two-stage (model-based and sensor-based) error compensation scheme that promotes the use of one laser tracker for several robots. The main purposes of utilizing an error compensation model for each robot in the first stage as subordinate to the laser tracker are twofold:

- To narrow down the error bandwidth of the robot thus reducing the time needed for sensor guided correction.
- To provide the laser tracker the position of the reflector mounted on a robot so that it can point the laser beam toward (Figure 3-14). This information is also helpful to reconnect the laser beam in case it is accidently interrupted due to the presence of fixtures and other supporting structures in the workspace, thus making the laser tracker less prone to its line-of-sight problem.

The proposed framework will be adopted to develop a distributed control system that automates the calibration and the aforementioned hybrid error compensation processes. Whenever a robot is reconfigured with a new end-effector or when it is relocated in the work-cell, the user only needs to run a robot program that performs the calibration and builds up the error compensation model automatically. Thereafter, the robots are able to improve the positioning accuracy with their own models and share the laser tracker to guide their tools into work. There is no need for a central cell controller that coordinates the exclusive use of the shared metrology system. Multiple robots will be able to send their requests for positional correction to the metrology

system simultaneously. The metrology system will collate these requests and through its task queue, perform the measurements in sequence.

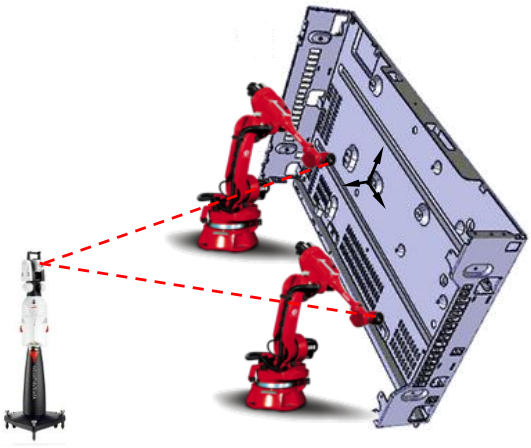


Figure 3-14 The purpose of model-based error compensation is improving robot accuracy and allowing one laser tracker to serve multiple robots

4 METHODOLOGY

This chapter presents the reader with the methodology used to achieve the objectives of this thesis. To address the first objective, an application framework for the integration of manufacturing resources (robots, sensors, end-effectors etc.) in the work-cell in a PnP manner is introduced. For the second objective, it is proposed that each robot should be calibrated beforehand to reduce its error band to an acceptable level, allowing for one global metrology source to be used for several robots. Typically, calibration is performed before the robots are put into production. This thesis, however, will demonstrate the use of the framework to automate the calibration and error compensation processes in-line with production activities. The following sections will describe the research approaches used. Section 4.1 firstly outlines features of the framework then describes the techniques to retain these features. Section 4.2 presents the robot calibration technique for elbow type manipulators, the challenges and the author's solution for modelling errors in the parallelogram linkage type manipulators. Further details on the developed framework, error modelling and compensation will be presented in Chapters 5 and 6. Figure 4-1 summarizes the work performed in this thesis.

4.1 The application framework for flexible system integration in robotics

4.1.1 Features of the framework

It is proposed that the framework would support the following capabilities which have been outlined in the literature review:

a. PnP integration: the framework permits removal of existing components / addition of new components without the need for shutting down the system and manual configuration. This characteristic encompasses the interoperability between the components, the modifiability and extensibility of the framework. It includes the "pluggable" ability, meaning that new components can be detected and hot-wired with others at run-time and the

"playable" ability, meaning that the functionalities of these components can be exploited without requiring software programming by system integrators.

b. Reusability: the framework provides a design template to develop the services for future components.

Other features from a software engineering standpoint are also provided:

c. Concurrent (multi-tasking) processing: processes within a service are distributed in independent threads to boost the performance whenever it is applicable.

d. Lock-free synchronization: services of shared resources (e.g., the laser tracker) use message queues, instead of the error-prone locking mechanism, to synchronize the tasks sent to it.

e. Asynchronous and publish/subscribe communications: services in the framework communicate with each other in non-blocking, publish/subscribe manners.

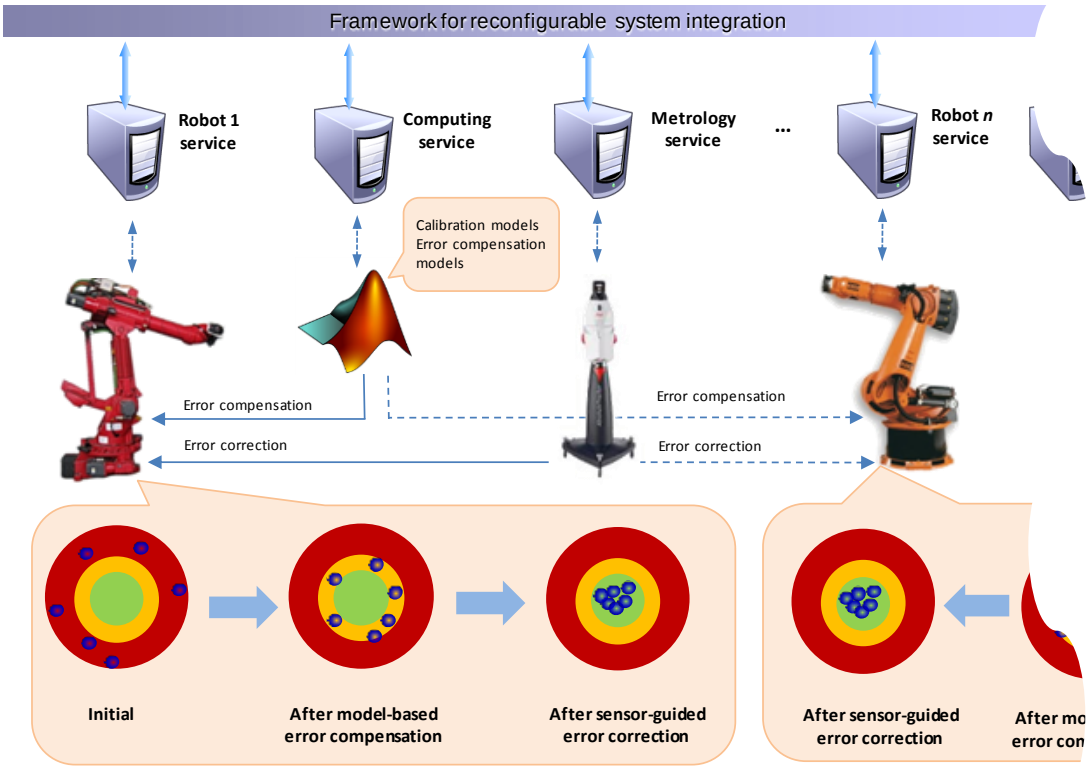


Figure 4-1 Overview of the framework and its applications for robot calibration and error compensation

4.1.2 Selecting the middleware

In this work, RDS from Microsoft has been chosen as the middleware for the framework due to the following advantages:

- RDS supports concurrent programming. Except for CORBA which also provides its own concurrency handlers, other communication middleware requires the programmer to rely on the OS kernel-supported methods for handling multi-tasking. Development and particularly debugging of software systems with many parallel processes using these methods (e.g., locks, semaphores) has historically been very difficult, especially for inter-process communication and synchronization. With CCR's novel concepts, such as *Port* (message queue), *Receiver* (message handler) and *Arbiter* (coordinator applied on the received message at a Port, allowing for different Receivers to be selected), complex concurrency problems can be solved in simple and robust codes (Figure 4-2).
- RDS supports communication in asynchronous and publish/subscribe manners which are necessary to implement loosely-coupled interfaces of services. It is worth noting that not all communication and robotic middleware provide such mechanisms, e.g., the DCOM and its variants (Namoshe *et.al.*, 2008; Mohamed *et.al.*, 2008).
- The availability of device drivers, familiar programming environments and supporting software on Windows OS. Indeed, when the Leica laser tracker AT901-MR was brought to the lab facility in early 2009, its SDK was provided only for the Windows platform. Having a friendly programming and run-time environment is also an important factor, given that the framework might be extended by other programmers and used by technicians on the floor. In the author's experience, it was much easier to absorb RDS concepts rather than those of DCOM or CORBA, which are intended to use for business integration. In this work, services are created using different programming environments (C#, C++, Matlab) and yet they are able to communicate with each other.

- RDS is freely provided for non-commercial use and frequently upgraded. Microsoft also holds a forum where programmers exchange their expertise with RDS's developers (MRDS Forums, 2012).

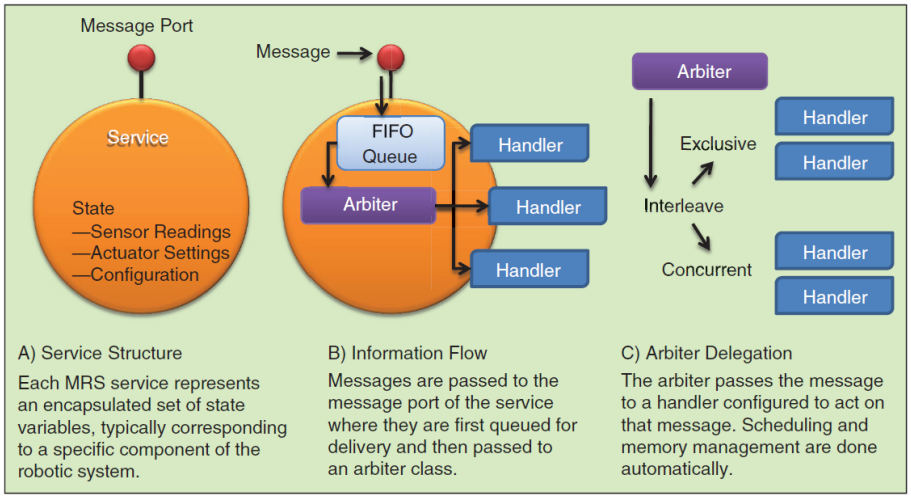


Figure 4-2 RDS service structure and concurrent message handling (Jackson, 2007)

As presented in section 3.3.2.3, there is always a trade-off between flexibility and real-time capability in every middleware platform and apparently, none is able to afford both the requirements. RDS is not an exception: it is not an ideal platform for implementing real-time systems with high rate and low latency communication. The RDS operates on top of the Windows.NET framework having too complicated memory management to guarantee real-time determinism. Therefore, to implement low level sensor guided robot motion control, it is suggested to keep the real-time code running in an unmanaged environment and then write an RDS service that interacts with the real-time code and the rest of the robot system (Jackson, 2007). In addition to the software part, point-to-point communication between the hardware parts (e.g., the robot and sensor) might also be needed, instead of via the RDS, to meet the demands for high data exchange rate in such situations. The author, however, does not rule out the possibility of using RDS for low level control. Experimental evaluation presented in part 5.3 will validate the communication rate of RDS for dynamic correction. RDS service structure and how it handles concurrency will be described in further details in part 5.1 of Chapter 5 and Appendix E.

4.1.3 Approach to PnP integration

Though exhibiting some degree of flexibility, the conventional distributed control architecture shown earlier in Figure 2-28 of Chapter 2 does not support PnP. Replacing an existing or adding a new basic service (BS) does require modification of the composite service (CS) because:

a. The new BS interface is unknown to the CS. This problem is due to the diversity of naming conventions, data types and message exchange patterns provided by heterogeneous basic services. The programmer must create an instance (stub) of the new BS in the CS in order that its input and output are transparent to the CS. This process must take place at development time.
b. The Control Logic is hard-coded in the CS and thus, must be modified to make use of the new BS's functionality.

Solutions to the problems will be presented in the following sections.

4.1.3.1 The "pluggable" Generic Device abstract service

In this work, the solution to the problem (a) stated above is providing services with a unique architectural design similar to what has been conceptually drawn in the IEC 61499 standard (section 3.3.2.3). Assuming that all components are derived from a virtual "generic device", their services thereby can be sub-classed from a common abstract service, namely the *Generic Device* service. When services are sub-classed from an abstract service, they inherit the interface of the abstract service (see section 5.1.1.4 for further details). As a result, any service in the framework will share the same interface of the *Generic Device* service, namely the *Generic Interface*, for receiving inbound messages from other services. It also has a dynamic array of the *Generic Device* service instances for sending outbound messages to any other services (Figure 4-3). Since all services appear to be identical from their viewpoints, connecting a new service to another service or detaching an existing one out of it only requires shrinking or growing the array by one instance, something that can be handled by the service itself automatically without the need for reprogramming.

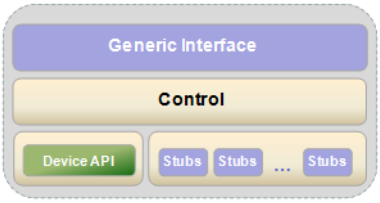


Figure 4-3 Any service in the framework has both female adapter (the Generic Interface) and male adapters (a dynamic array of server stubs) allowing arbitrary incoming and outgoing connections with other services

With the given structure of a service, it is possible to construct the connection topology for a robot system as depicted in Figure 4-4. In the figure, the robot service connects with the tool, sensor and computing services; the sensor service might also connect with the computing service that processes its measurements, e.g., image processing, before returning the results to the robot service. There is no need to create a CS that glues the services together; each service in the framework is a CS itself whose connections with others can be established and destroyed at runtime. Notice that a robot service might include not only the robot API but those of the sensors that connect directly and communicate with it on a real-time basis for dynamic correction. Unlike other resources integrated via the framework, these modules are not detachable from the robot service and are controlled by the real-time code as presented earlier.

The *Generic Interface* consists of a number of functions that facilitate the exchange of messages between the services. In order that a service connects with another one on the network, it invokes a function named *Subscribe(address)* provided the IP address of the remote service. For example, the robot service in Figure 4-4 must subscribe to the tool, sensor and computing service by calling this command. Thereafter, the robot service is able to send requests to them and receive their status and data feedback in publish/subscribe manner. To send a request, the robot service invokes a function named *CreateProcess(process)* to initialize a process at the remote service. The parameter *process* passed to the function is a data structure containing the command to be executed along with its optional parameters and input data, if required. The idea is depicted in Figure 4-5, in which the robot

service uses the *CreateProcess* function to activate the commands *SnapShot* of the camera service and *Drill* of the tool service. After the camera and tool's controllers have executed these commands, their services will return the feedback including the command results and data, if any, to the robot service via *ProcessUpdate*(*process*) event notifications. The *ProcessUpdate* notification might be sent once or several times, depending on the type of the executed command. An example of the latter case is when the camera takes a movie which results in a time series of images. It is also worth noting from Figure 4-5 that the *CreateProcess* and *ProcessUpdate* messages are able to envelope different types and sizes of data between the services. Obviously, the recipient services must also have functions for handling these messages in proper ways. Service structure and message handlers will be explained in detail in part 5.2 of Chapter 5.

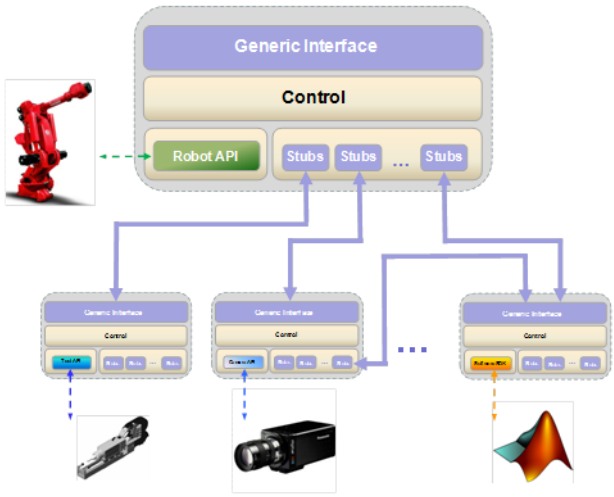


Figure 4-4 Connection topology of services in the framework

4.1.3.2 The "playable" robots

Conventionally, the robot service must interpret the dispatched offline program line by line into equivalent robot APIs motion commands. It must also replace associated name tags in the program with relevant *CreateProcess* function calls to the sensor and tool services. In this way, the Control Logic is hard-coded in the robot service. As a result, the robot service must be reprogrammed if a new

device is used, even when the service of this device has already been made pluggable.

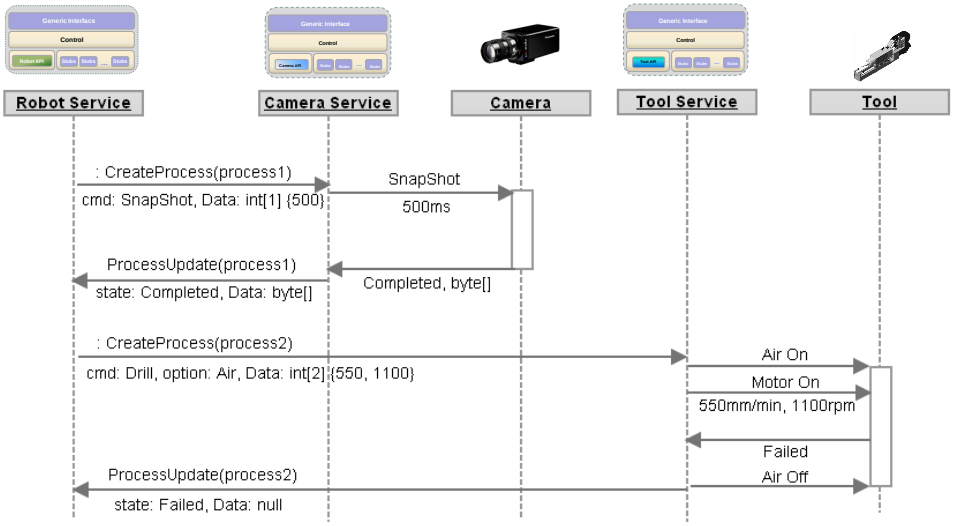


Figure 4-5 The robot service can invoke different commands on other services via one standard CreateProcess operation.

In this work, a simple solution to the problem above was found by realizing that robots are reprogrammable devices. Indeed, all industrial robots are equipped with high-level programming languages (e.g., Comau robots use PDL, ABB robots use RAPID, Kuka robots use KRL languages etc.). These programming languages are provided with a variety of condition handlers and mathematical functions enough for handling complicated control flow and data processing. Therefore, instead of being treated like dumb devices: the robots receive and execute control commands sent from their computer programs (services) in the slave/master relationship, their roles are reversed: the robots execute the provided offline programs and send instructions to their services to control the peripherals. The robots can also delegate complex tasks, e.g., image processing, regression analysis to the computing service and retrieve the final results. The main advantage of this reversal approach is that the Control Logic resides in an editable robot program instead of being hard-coded in computer program, thus can be easily modified. The following example will demonstrate how the robot generates the activities given in Figure 4-5 in Comau robots' PDL, a PASCAL-like language.

```
PROGRAM EXAMPLE

…

-- Subroutines

ROUTINE SNAPSHOT (exposure_time: integer): boolean

BEGIN
  ext_cmd_finished:=false
  ext_cmd_result:= false
  WRITE pc_client ('ACTIVATE CAMERA #SnapShot##', exposure_time)
  WAIT FOR ext_cmd_finished
  RETURN ext_cmd_result
END SNAPSHOT

ROUTINE DRILL (air: boolean, feed: integer, speed: integer): boolean
VAR air_on:string[3]
BEGIN
  ext_cmd_finished:=false
  ext_cmd_result:= false
  IF air=true THEN air_on:= 'Air'
    ELSE air_on:= ''
  ENDIF
  WRITE pc_client ('ACTIVATE TOOL #Drill#',air_on,'#', feed, ',',speed)
  WAIT FOR ext_cmd_finished

  RETURN ext_cmd_result
END DRILL

-- Main Program

BEGIN

  …

  MOVE TO pnt0001
  IF SNAPSHOT(500)=FALSE THEN
     DEACTIVATE
  ENDIF

  MOVE TO pnt0002

  IF DRILL(TRUE,550,1100)= FALSE THEN
     DEACTIVATE
  ENDIF

END EXAMPLE
```

The process takes place as follows:

- In the main PDL program, the robot moves to two points. At one point, it activates the camera by calling the routine SNAPSHOT which sends the string “ACTIVATE CAMERA #SnapShot##500”, at the other point, it activates the tool by calling the routine DRILL which sends the string “ACTIVATE TOOL #Drill#Air#550,1100” to the robot service. After sending each string, the robot halts its execution and waits until the variable *ext_cmd_finished* is set to *true*.
- When receiving these strings, the robot service interprets them to the *CreateProcess* function calls to remotely activate the commands *SnapShot* at the CAMERA service and *Drill* at the TOOL service. After receiving the

ProcessUpdate notifications from these two services, the robot service returns the results in the variable *ext_cmd_result* then sets the variable *ext_cmd_finished* of the PDL robot program to *true*, which resumes the robot execution. Based on the retuned value of the variable *ext_cmd_result*, the robot can perform proper actions (in the example, it simply deactivates itself from running if there are errors in the sensor and tool services). Handling data returned by the sensor would be done in a similar manner where the feedback data are converted into data types supported by PDL language and processed by the robot program (not shown in the code snippet above).

For low-level dynamic correction (e.g., force control), the robot program might send a different string and relinquish its control to the robot service. The robot service then executes its real-time code written using the robot and sensor APIs for dynamic correction (e.g., force control). When the correction has finished, the control will be returned to the robot program.

This approach differs from the conventional PC-based control as follows:

- The Control Logic resides in editable, text-based robot programs. Therefore, a new device introduced to the system only requires writing new routines in a robot programming language while all computer programs remain unchanged. This is a relatively simple job compared with reprogramming the services in C/C++, and hence, can be done by technicians on the floor, without concerns about network, threads, synchronizations and the like. Since the technicians can develop new applications by themselves without the need for a specialist from outsourced companies, production downtime and costs are reduced.
- There is no need to translate the robot program into robot API motion commands. When generating the robot program in some OLP software, the technicians also assign specific name tags that correspond to the operations of the robot. After these name tags are replaced by the corresponding routine calls (e.g., SNAPSHOT, DRILL) using a text editor, the robot program can be downloaded directly to the robot controller for execution without the need for any further translation.

- Less error prone. In the code snippet above, after the robot sends out a string command, it waits until the handshake variable *ext_cmd_finished* is set by the service when the communications with the peripheral devices have completed. Any errors induced by noise or device malfunction that corrupt services' activities will only halt the robot execution at this point rather than causing fatal failures. In addition, since the robot program is executed by the robot controller instead of the PC, the technicians are able to test the program using the teach pendant on the floor, thus reducing the likeliness of failures.

4.1.4 Approach to lock-free task synchronization

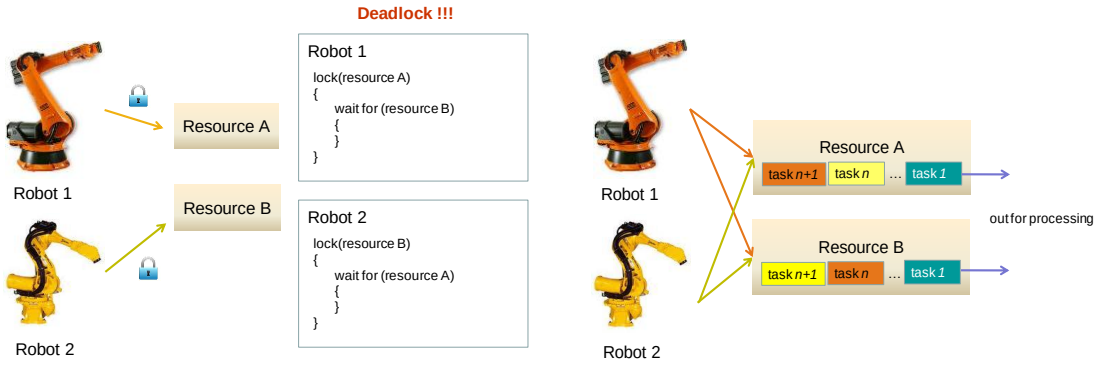


Figure 4-6 Task synchronization. From left to right: a. Deadlock situation when using traditional locks; b. To avoid, services in the framework use internal task queues

As presented in section 3.3.2.3, a locking mechanism is usually employed to resolve mutually exclusive access to shared resources (e.g., global metrology, conveyor). This mechanism, however, may cause the so-called deadlock situation in which each robot holds a lock and waits for the other to be released and thus, all end up waiting forever. In the framework, the necessity of locks is eliminated. Each service in the framework utilizes a First-In-First-Out (FIFO) task queue built upon the CCR's Port structure for sequencing the tasks (Figure 4-6). It also has a scheduling algorithm that will rearrange the tasks in the queue so that they will be processed in the right order. The scheduling mechanism will be described in section 5.2.3.5.

4.2 Robot calibration and error compensation

Robot calibration technique, which has been well-established for open-loop serial manipulators (e.g., the elbow type robots), will be introduced in section 4.2.1. Section 4.2.2 presents the author's novel approach to resolve remaining challenges in the calibration of serial manipulators having a parallelogram linkage.

4.2.1 Kinematic calibration for open-loop serial manipulators

4.2.1.1 Error modelling

As briefly introduced in section 3.2.2.1, the most important step in a robot calibration process is error modelling: deriving a mathematical formulation mapping the unknown error sources in the robot structure with the measurable tool pose errors. Error modelling usually starts from the kinematic model of the robot then perturbs the nominal kinematic parameters with the unknown error sources, which will result in the tool pose errors.

Suppose an open-chain manipulator has n+1 links numbered from 0 to n serially connected together via n actuated joints, numbered from 1 to n. Denote $x = (p_x, p_y, p_z, \phi_x, \phi_y, \phi_z)$ the (6×1) - vector of positions and orientations of the end-effector (the TCP frame) in the base frame. It is possible to write the forward kinematic model of the robot given in the Appendix A in the form:

$$x = f(q, g) \tag{4.1}$$

where the function f is derived from equations (A.1) and (A.2); $q = (q_1, q_2, ..., q_n)$ is $(n \times 1)$ - vector of command joint variables; $g = (\theta, d, a, \alpha)$ is $(4n \times 1)$ - vector of nominal DH parameters of the manipulator, in which θ_i, d_i, a_i, α_i respectively are joint angles, link offsets, link lengths and twist angles associated with link i. Note that in calibration, the units are represented in metres and radians.

Considering geometric error parameters: if g is perturbed with the error $\Delta g = (\Delta\theta, \Delta d, \Delta a, \Delta\alpha)$ to be identified, x will deviate from its value an amount $\Delta x = (\Delta p_x, \Delta p_y, \Delta p_z, \Delta\phi_x, \Delta\phi_y, \Delta\phi_z)$ as:

$$x + \Delta x = f(q, g + \Delta g) \tag{4.2}$$

Assuming Δg is small, the linear approximation of Δx can be obtained as:

$$\Delta x = H(g)\Delta g \tag{4.3}$$

where:

$$H(g) = \frac{\partial f}{\partial g} = \left[\frac{\partial f}{\partial \theta} \quad \frac{\partial f}{\partial d} \quad \frac{\partial f}{\partial a} \quad \frac{\partial f}{\partial \alpha}\right] = \left[H_\theta \quad H_d \quad H_a \quad H_\alpha\right] \tag{4.4}$$

is a $(6 \times 4n)$- matrix relating Δx with Δg and is called the identification Jacobian matrix. Each column of $H(g)$ represents the sensitivity of the tool pose error Δx with regards to a particular parameter in Δg, for example, $H_{\theta 2}$ is the Jacobian of parameter $\Delta\theta_2$ of link 2 and so forth. In practice, $H(g)$ is usually derived through the so-called differential homogeneous transformations (Paul, 1981), rather than differentiating equation (4.2) directly to avoid complication. Detailed derivation of $H(g)$ using this method is given in Appendix B.

The system of six equations (4.3) represents the desirable mathematical formulation (error model) between the unknown error parameters Δg and the measurable tool pose error Δx:

$$\Delta x = x_M - x \tag{4.5}$$

where x_M is the measured tool pose by an external sensor.

4.2.1.2 Identification

The actual number of equations k $(k \le 6)$ in equation (4.3) that can be used to identify Δg depends on how many components of x_M (and thus Δx) are observed by the sensor in equation (4.5). For example, if the sensor only measures position components of the end-effector then each measurement provides k=3 equations and if it measures both positions and orientations of the

end-effector, k=6. Since $k<<r$: the number of identifiable geometric error parameters in Δg ($r \leq 4n$), equation (4.3) is underdetermined, thus a large number of measurements of Δx are required to solve it in least square sense:

$$\begin{aligned} \Delta x_1 &= H(g_1)\Delta g \\ &\vdots \\ \Delta x_m &= H(g_m)\Delta g \end{aligned} \tag{4.6}$$

where m is the number of measurements taken with different robot configurations (i.e., with different sets of q) such that $mk \geq r$.

The system of mk equations (4.6) can be stacked into matrix form as:

$$\Delta \mathbf{x} = \mathbf{H}(g)\Delta g \tag{4.7}$$

where $\Delta \mathbf{x}$ is a ($mk \times 1$) – concatenated vector of the measurements $\Delta x_i, i = 1...m$, $\mathbf{H}(g)$ is the ($mk \times r$) – regression matrix.

If the matrix $\mathbf{H}(g)$ is full rank, the ordinary least square solution of (4.7) is:

$$\Delta g = (\mathbf{H}(g)^{\mathrm{T}}\mathbf{H}(g))^{-1}\mathbf{H}(g)^{\mathrm{T}}\Delta \mathbf{x} \tag{4.8}$$

Since the identification Jacobian contains a linear approximation, the process (4.1-8) must be applied iteratively with the new update $g = g+\Delta g$ until Δg becomes sufficiently small. The calibration result (the magnitude of the residual error $\Delta \mathbf{x}$) depends mostly on whether $H(g)$ has been accurately and sufficiently modelled (with geometric and non-geometric errors) and the accuracy of the sensor used.

When $\mathbf{H}(g)$ is rank deficient (e.g. due to the presence of unidentifiable, poorly identifiable or linearly dependent parameters in g), it will cause a problem when inverting $(\mathbf{H}(g)^{\mathrm{T}}\mathbf{H}(g))^{-1}$ in equation (4.8). In such cases, numerical tools through manipulation of $\mathbf{H}(g)$, e.g. using singular value decomposition (SVD), are usually used to eliminate parameter redundancies in the model. For SVD, $\mathbf{H}(g)$ is decomposed as:

$$\mathbf{H}=\mathrm{U}\Sigma\mathrm{V}^{\mathrm{T}} \tag{4.9}$$

where $\sum$ is the ($mk\ x\ r$) – diagonal matrix in decreasing order of singular values $\sigma_1 > \sigma_2 > \ldots > \sigma_r$. Poorly identifiable parameters are indicated by zero or very small singular values. It is heuristically suggested that the condition number of a well-conditioned regression matrix should be less than 100 (Bernhardt, 1993):

$$\kappa(H) = \frac{\sigma_1}{\sigma_r} < 100 \tag{4.10}$$

If condition number is above 100, elements of column v_r of matrix V are examined. If there is an element j of v_r that is much larger than the others, the corresponding error parameter Δg_j is a candidate for elimination (Siciliano *et.al.*, 2008). This process of pinpoint and elimination of parameters is repeated until the condition (4.10) is met, then it is possible to solve g from equation (4.8).

4.2.1.3 Error compensation

After the calibration process has completed, the kinematic model f with identified parameters g is able to predict the actual tool pose more accurately than the one used by robot controller. Ideally, g should replace for the nominal values defined in the robot controller but in most cases, modification is not allowable. Error compensation thereby is usually done by means of software as follows.

Given the programmed end-effector location x^d, calculate the joint solution q using the nominal inverse kinematic model. The deviation Δx between the desired x^d and the actual pose x predicted by the identified forward kinematic model f can be compensated by small joint increments δq as:

$$\delta q = J(q)^{-1} \Delta x \tag{4.11}$$

where $J(q)$ is the well-known manipulator Jacobian (Spong *et.al.*, 2004). The compensated joint values q^d:

$$q^d = q + \delta q \tag{4.12}$$

will be downloaded to the robot controller and replace the nominal q to correct Δx before the motion take place (static correction). The idea of this error

compensation scheme can be thought of as 'providing the robot a false target so that when it reaches there, it actually is closer to the desired one (Figure 4-7).

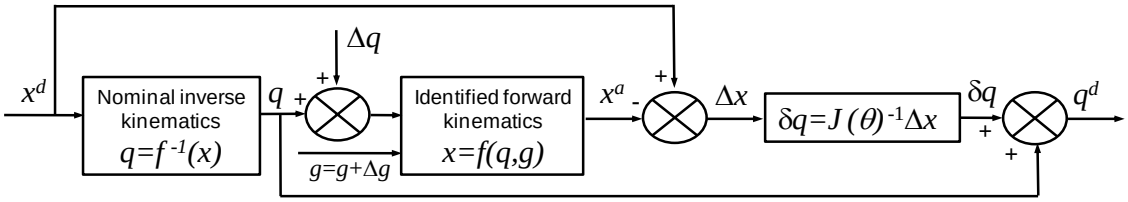


Figure 4-7 Error compensation using the calibration model (Khalil *et.al.*, 2004)

4.2.1.4 The "standardized" modified DH model

As introduced in section 3.2.2.1 of the literature review chapter, the modified DH model suggested by (Hayati *et.al.*, 1985) is the most commonly used in kinematic calibration. The authors have found that the original DH convention suffers limitation when modelling parallelism of a consecutive pair of parallel axes. When two adjacent joint axes are parallel, e.g., joint 2 and 3 of elbow type manipulators, small tool error Δx may result in unrealistic identified Δd_3 (Figure 4-8). The reason is because Δd_3 is linearly dependent with Δd_2. To overcome, an additional term $rot(y,\beta_i)$ is post-multiplied to the original DH model (equation (A.2)), resulting:

$$T_i^{i-1} = rot(z,\theta_i)tran(z,d_i)tran(x,a_i)rot(x,\alpha_i)rot(y,\beta_i) \tag{4.13}$$

in which the identifiable kinematic parameters follow the rule:

- $\Delta\theta_i, \Delta a_i, \Delta\alpha_i, \Delta\beta_i$ if joint *i* is a rotary joint and $z_{i-1} // z_i$
- $\Delta\theta_i, \Delta d_i, \Delta a_i, \Delta\alpha_i$ if joint *i* is a rotary joint and $z_{i-1} \perp z_i$
- $\Delta\theta_i, \Delta\alpha_i$ if joint *i* is a prismatic joint.

It has also been proved that the maximum number of identifiable geometric parameters for a robot having R rotary joints and P prismatic joint is 4N+2P+6 where the last number 6 is for parameters of two additional transformations relating the sensor frames and the robot frames in cases the sensor cannot

measure position of the robot end-effector directly but a target fixed on it (Veitsschegger *et.al.*, 1988; Schröer *et.al.*, 1997).

Using this convention, comprehensive formulations of the identification Jacobian coefficients in equation (4.4) are given as (Benett *et.al.*, 1995; Siciliano *et.al.*, 2008):

$$H_{\theta i} = \begin{bmatrix} z_{i-1} \times p_{i-1} \\ z_{i-1} \end{bmatrix},\ H_{di} = \begin{bmatrix} z_{i-1} \\ 0 \end{bmatrix},\ H_{ai} = \begin{bmatrix} x_i \\ 0 \end{bmatrix} \qquad (4.14)$$

$$H_{\alpha i} = \begin{bmatrix} x_i \times p_i \\ x_i \end{bmatrix},\ H_{\beta i} = \begin{bmatrix} y_i \times p_i \\ y_i \end{bmatrix}$$

where x_i, y_i, z_i are directional vectors and p_i is the position of link frame F_i expressed in the base frame F_0. Details on this derivation are described in Appendix B. From what has been described thus far, it can be seen that all issues associated with modelling, identification and compensation for geometric parameters of serial-link manipulators have been well-defined and treated in the literature. Therefore, it is no longer a challenge to adopt the technique to improve robot accuracy to some level.

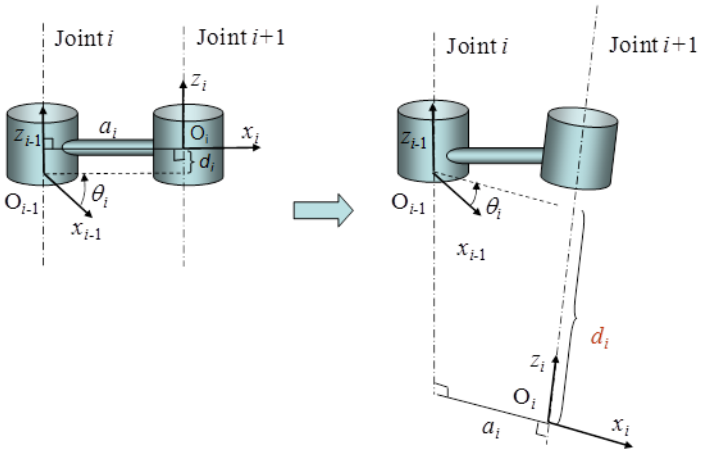


Figure 4-8 Small deviation from the ideal parallelism (left) may cause unrealistic identified value of d_i (right)

4.2.2 Kinematic calibration for serial manipulators having a parallelogram linkage

4.2.2.1 Geometric error modelling

It is worth noting that the above error modelling convention, however, cannot be employed directly on serial manipulators that contain closed-loop chains, i.e. those with the parallelogram linkage. In these robots, passive joints are driven by actuated joints through the parallelogram mechanism, implying that they are dependent (unidentifiable) parameters via some constraints and are degraded by the errors of these components. Therefore, an additional error model of the loop must be derived from the constraint equations and then merged with the global open chain's error model described in equation (4.3). Deriving the loop's error model is usually complicated and introduces further parameter redundancy which will cause the regression matrix $\mathbf{H}$ in equation (4.7) rank-deficiency. For simplicity, many researchers ignored the parallelogram structure and thus, regarded the robots as standard serial ones. However, it will be shown later in Chapter 8 that identification accuracy can be drastically improved if the loop's errors are taken into account. In their paper, Schröer *et.al.* (1997) modelled a degenerated parallelogram structure as a planar four bar linkage, of which the position constraints were mathematically solved for its passive joint angles with respect to actuated joint angles and actual link lengths. A similar approach was taken in the work of (Marie *et.al.*, 2008), where this solution was further differentiated to obtain the loop's error model. Though such derivations are necessary, difficulties may arise because solutions of the loop constraints usually are highly nonlinear and hard to be differentiated. The reader can look up in Appendix C to see how complex it would be to differentiate the position solution of the four bar linkage. In contrast, the calibration model suggested by (Alici *et.al.*, 2005) is too simple because the essential relation between errors in passive joint angles and other loop parameters was not provided. Ananthanarayanan *et.al.* (1992) suggested an experimental method to investigate link length errors of a parallelogram mechanism. As this method relies on moving the arm in a specific trajectory, the calibration results are prone

to unaccounted effects, i.e. the compliance due to robot's gravitational loading. It is thus desirable to have a simpler yet more accurate model for manipulators of this type.

In this work, an improved kinematic model for parallelogram linkage type manipulators is developed. To avoid the complications mentioned above, the loop's error model is derived by differentiating the loop's position constraint equations, instead of solving the equations first then differentiating the solution (To *et.al.*, 2012). After being merged with the resulting loop's error model, the manipulator's global model becomes similar to that of an open-loop robot, which makes it possible to exploit the well-defined techniques presented thus far in section 4.2.1. The analytic form of the Jacobian matrix is also given, based on which remained redundant parameters due to kinematic design of the parallelogram structure can be eliminated without having to use the trivial numerical technique described in equations (4.9-10). Geometric error modelling for a parallelogram linkage type manipulator will be presented in detail in section 6.2.1 of Chapter 6.

4.2.2.2 Non-geometric error modelling

To further improve robot accuracy, non-geometric errors must be considered. As presented in section 2.2.5 of Chapter 2, among several non-geometric sources which are not always evident, compliance (elastic deflection) due to gravitational loading contributes significantly to inaccuracy and hence, was chosen to be modelled and compensated. For a large robot, the deflection is induced not only by applied payload (e.g., the end-effector's weight) but its link masses. Figure 4-9 describes an experiment in this work discovering the deflections caused by link masses of the robot used: even when the robot is unloaded, rotating the forearm alone around joint 3's axis caused severe deviation in the positions of a laser tracker's SMR target fixed on the upper arm. This can be explained as during the rotation of the forearm, its weight $\mathrm{F_g}$ created a variable moment $\mathrm{M_g}$ at joint 2 that deflected the upper arm.

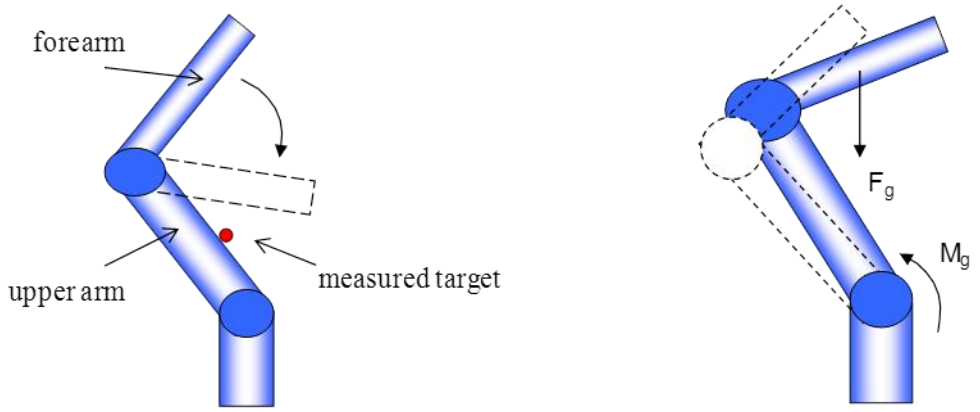


Figure 4-9 Rotating the forearm causes deflection of the upper arm (left) as a result of the moment M_g created by the mass F_g of the forearm (right).

In this work, deflection modelling followed the assumption which has been used in many other researches for elbow type manipulators that compliances mostly occur at the joints whilst those at the links are negligible. A joint is modelled as a torsional spring with constant joint stiffness κ (Nm/rad). Therefore, if τ_i is the generated torque at a joint i to counteract with the applied external moment (e.g., M_g in Figure 4-9), joint deflection $\Delta\theta_i^d$ from its unloaded position is calculated as:

$$\Delta\theta_i^d = c_i \tau_i \tag{4.15}$$

where $c_i = 1/\kappa_i$ is joint compliance (rad/Nm). This is the constant to be identified for each joint.

When joint deflection is considered, joint errors $\Delta\theta_i$ in the parameter g in equation (4.3) include not only the constant offsets but the parts due to deflections:

$$\Delta\theta_i = \Delta\theta_i^{off} + \Delta\theta_i^d = \Delta\theta_i^{off} + \Delta\theta_i^s + \Delta\theta_i^e \tag{4.16}$$

for i=1..n. In (4.16), $\Delta\theta_i^{off}$ is constant joint offset (geometric error) while $\Delta\theta_i^s$ and $\Delta\theta_i^e$ are variable joint deflections (non-geometric errors) caused by structural loading and external applied payload, respectively. Substituting (4.15, 4.16) into (4.3) then rearranging the resulting equations, one may identify the compliance

coefficients c_i of the joints, provided that the applied moment τ_i is accurately modelled or measured. The following calculations will be done in this work:

- For the deflection $\Delta\theta_i^s$ induced by link mass, τ_i is calculated from the static equilibrium at joint i. The computation of τ_i for parallelogram linkage type manipulators in this thesis is modified from the method presented in (Judd *et.al.*, 1990; Gong *et.al.*, 2000) for elbow-type manipulators. This model will be presented in section 6.3.1 of Chapter 6.
- For the deflection $\Delta\theta_i^e$ induced by applied payload, τ_i is calculated from the well-known static force-torque relation (Paul, 1981; Spong *et.al.*, 2004):

$$\tau = J(q)^T W \tag{4.17}$$

where $\tau = (\tau_1, \cdots, \tau_i, \cdots, \tau_n)$ is the $(n \times 1)$ – vector of torque generated at *n* actuated joints to counteract with the generalized force (force and moment) W applied at the end-effector. This equation usually requires an F/T sensor to measure precisely W which, in this case, is the generalized force created by the weight of the end-effector. This work will present a compensation model for the deflection without the need for an F/T sensor. This model will be presented in section 6.3.2 of Chapter 6.

It can be seen from section 4.2.1 that kinematic calibration for elbow type manipulators has been "standardized" in the literature, especially for geometric errors. This thesis further presents a relevant work for parallelogram linkage type manipulators, taking into account both geometric errors and joint compliance. It is thereby possible to adopt the calibration technique to most popular kinematic designs of industrial robots to improve their absolute accuracy to some level. The global metrology (e.g., a laser tracker) will only be needed to correct small residual errors and thus, can serve more than one robot in a multi-robot work-cell. Further details on how to automate the calibration and the two stage (model-based and sensor-based) error compensation process using the proposed framework will be described in Chapter 7 and 8.

5 DEVELOPMENT OF THE FRAMEWORK

This chapter presents the proposed application framework for flexible system integration in robotics. Section 5.1 firstly introduces Microsoft Robotic Developer Studio (RDS), the middleware platform that the framework is developed upon. Background information provided in this section is essential for section 5.2 which describes the framework's service architectures designed for PnP integration capability. Section 5.3 describes the framework's performance in terms of message exchange rate and latency. Code snippets in C# along with comments will be given in this chapter.

5.1 Robotics Developer Studio: the middleware

As introduced previously in sections 3.3.2.2, the RDS consists of two most important modules: Decentralized Software Service Protocol (DSS) and Concurrency and Coordination Runtime (CCR). The DSS is an ordinary communication middleware allowing multiple services to interoperate via the network whereas the CCR, operating at lower level, allows multiple tasks within each service to run concurrently. These two modules will be presented in the following sections.

5.1.1 Decentralized Software Service Protocol

5.1.1.1 DSS service

In the context of middleware technology, a service generally consists of:

- Interface: the description of what operations the service performs (the types of messages the service can receive). In DSS, interface is also referred as service contract.
- Implementation: actual handlers of the interface.
- State: a collection of state variables describing the content of the service.

For example, the interface (contract) of the camera service depicted earlier in Figure 2-26 of Chapter 2 may consist of the operations *Connect* and *Disconnect* for setting up communication with the camera controller, *SnapShot*, *MovieShot*,

and *StopMovie* for start/stop grabbing images. The service state may contain variables describing the current image, image dimension, frame rate and other parameters such as connection status, camera status (busy or idle) etc. When the service receives a message, e.g. a *SnapShot* call, the corresponding handling function in the service implementation will be triggered to perform its action, e.g., to command the camera API to take an image, then write the image to the state and return it to the caller.

The specific DSS service model, shown diagrammatically in Figure 5-1, can be described as follows. *Service URI* (Universal Resource Identifier) and *Contract Identifier* are just system numbers used to identify the service instance and its contract with other resources on the network. The service may have a *Subscription Manager* for managing a list of its subscribers (the services subscribing to it). It may also have one or more *Partners* (other services that it subscribe to). Messages sent from the service subscribers and notifications sent from the service partners will arrive at the *Main Port* and *Notification Ports*, which basically are FIFO message queues. *Service Handlers* and *Notification Handlers* are functions of the service implementation that process these messages and notifications out of their queues.

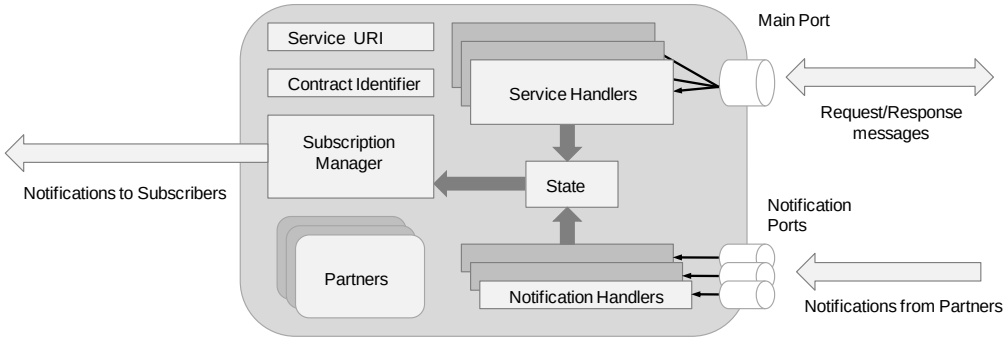


Figure 5-1 DSS service architecture (Microsoft, 2008)

In DSS, the components (state, interface and implementation) of the above-mentioned camera service are encapsulated into three classes: *CameraState*, *CameraOperations* and *CameraService* as depicted in Figure 5-2. The class *CameraService* is the main body of the service, from which the other two classes are instantiated with the objects *state* and *mainPort*, respectively. It also

has a variable named *submgrPort* of type *SubscriptionManagerPort* through which notifications to the service subscribers will be sent. DSS services also use a *manifest* at start-up that describes its execution context. Manifest is a XML file that lists service partners and their addresses on the network.

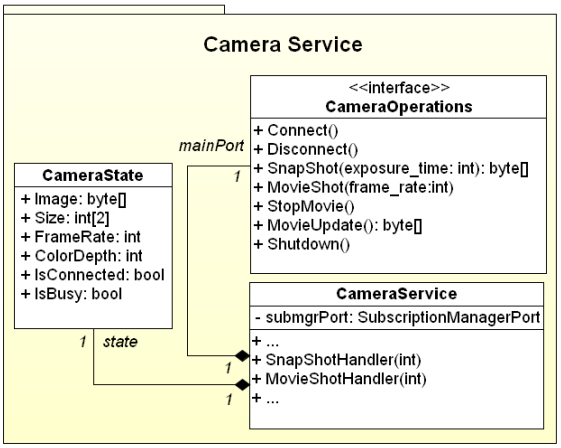


Figure 5-2 Class diagram of a DSSP service

5.1.1.2 Message exchange patterns

Within the context of a service handler, a DSS service can send messages to other services in two manners:

- Request/Response (two-way) messaging pattern: a single request message sent from a sender to a receiver, followed by a single response sent from the receiver to the sender of the request.
- Publish/Subscribe (one-way) messaging pattern: a single message, in the form of an event notification, sent from a publisher to subscribers.

The choice of message exchange patterns depends on the type of operations: some operations only allow the programmer to use the one-way message exchange pattern while others can support both. For example, the handling function for the *SnapShot* request (*SnapShotHandler)* might either send the captured image in the response or in a separate notification. On the other hand, the handling function for the *MovieShot* request (*MovieShotHandler)* must use notifications because it is not possible to send a time series of captured images in a single response message. Multiple *MovieUpdate* notifications which

embody the images will be sent (through the *submgrPort*) until the camera service receives a *StopMovie* request (Figure 5-3).

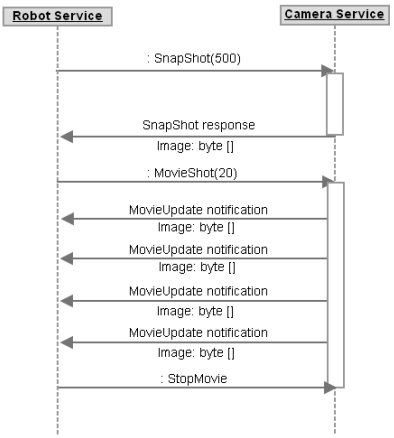


Figure 5-3 Example of the message exchange of the camera service

5.1.1.3 Procedure of service integration

Suppose a robot service is to use the camera service. The following procedure must be followed while programming the robot service in order that it is able to integrate with the camera service:

- When the camera service is compiled, its service contract is embodied into a Proxy dynamic link library (DLL) file. The robot service needs to reference with this Proxy DLL so that the camera's functions *SnapShot*, *MovieShot* etc. become transparent to it.
- In the robot service's interface (the class *RobotOperations*), declare the types of notifications from the camera service that the robot service wants to receive, i.e., the *Shutdown* and *MovieUpdate* notifications

```
using camera = Camera.Proxy;                    // Using the camera Proxy DLL
[ServicePort()]
public class RobotOperations : PortSet<
        DssDefaultLookup,
        DssDefaultDrop,
        Replace,
        Get,
        Subscribe,
        camera.Shutdown,                        // Notifications received from the camera
        camera.MovieShot,
        … >
{
}
```

- At the start of the robot service implementation (the class *RobotService*), invoke the standard operation *Subscribe* to subscribe the robot service to the camera service.

```
using camera = Camera.Proxy;                    // Using the camera Proxy DLL
class RobotService : DsspServiceBase
{
    // The robot service declares the camera service as its partner
    [Partner("Camera", Contract = camera.Contract.Identifier, CreationPolicy =
    PartnerCreationPolicy.UseExisting)]

    // Defines instances of the camera service
    camera.CameraOperations camPort = new camera.CameraOperations();
    camera.CameraOperations camNotify = new camera.CameraOperations();

    protected override void Start()
    {
        base.Start();

        // Subscribing to the camera service
        camPort.Subscribe(camNotify);
        ...
    }
}
```

Herein, the robot service declares the camera service as its partner (publisher) in the bracket [*Partner...*]. Two instances (client stubs) of the camera interface *CameraOperations*: *camPort* and *camNotify*, one for sending outbound and another for receiving inbound messages (notifications) with the camera service, are created. The robot service then subscribes to the camera service by using the standard operation *Subscribe*.

- Finally, provide the address of the camera service on the network in the robot service manifest (configuration) file as follows:

```
<?xml version="1.0" ?>
<Manifest
    xmlns="http://schemas.microsoft.com/xw/2004/10/manifest.html"
    xmlns:dssp="http://schemas.microsoft.com/xw/2004/10/dssp.html"
    xmlns:Robot = "http://schemas.cranfield.ac.uk/2010/12/Robot.html">
    <CreateServiceList>
        <ServiceRecordType>
          <dssp:Contract> http://schemas.cranfield.ac.uk/2010/12/robot.html
          </dssp:Contract>
          <dssp:PartnerList>
            <dssp:Partner>
              <dssp:Service>http://192.168.0.2:50000/Camera</dssp:Service>
              <dssp:Name>Robot:Camera</dssp:Name>
            </dssp:Partner>
          </dssp:PartnerList>
        </ServiceRecordType>
    </CreateServiceList>
</Manifest>
```

Herein, the robot service indicates that its partner, enclosed in the <*dssp:Partner*> </*dssp:Partner*> XML mark-up, is the camera service supposedly running at the at IP address 192.168.0.2, port 50000. By doing this, the client stubs *camPort*, *camNotify* of the class *RobotService* become liaised with the server stub of the actual camera service running at the given node on the network.

Having followed these steps, it is now possible to call the camera's *SnapShot* operation in request/response manner within a service handler of the class *RobotService* as follows:

```
// Invoke the SnapShot operation of the camera service
Activate(Arbiter.Choice(camPort.SnapShot(500),
        delegate (byte[] image)
        {
            // If the call is succeeded, the response will be the captured image
            …
        },
        delegate (Fault fault)
        {
            // Otherwise, an error will be catched and processed here
            …
        })
);
```

In the code snippet above, the camera's *SnapShot* function is called via the *camPort* instance provided the exposure time of 500ms as the parameter. After sending the message to the camera service, the robot service waits for the response, which could either be the captured image if successful or a *Fault* message otherwise. The CCR's command *Arbiter.Choice* will trigger the corresponding delegate handler depending on the types of response messages.

The robot service can also simply invoke the camera's *MovieShot* function as:

```
camPort.MovieShot(20);
```

where the parameter 20 is the frame rate per second. Because *MovieShot* uses the one-way messaging pattern, the captured images will be sent in separate *MovieUpdate* notifications and the robot service must implement a separate handler for these notifications, e.g., the *MovieUpdateHandler* below:

```
public void MovieUpdateHandler(camera.MovieUpdate notification)
{
        // Retrieve the image out of the notification message
        byte[] image = notification.Body.Image;

        // Processing the image
        …
}
```

The function *MovieUpdateHandler* will be triggered every time a *MovieUpdate* notification arrives at the *camNotify* instance until the robot service invokes the camera's function *StopMovie*.

5.1.1.4 Abstract service

In DSS, developers can define abstract services to represent actuators and sensors that have common characteristics. For example, the camera service discussed thus far may serve as an abstract service for many types of IP cameras ranging from webcams to machine vision systems because all of them operate in the same way regardless of their make: they're all able to be remotely connected / disconnected and take a single / a series of images. Likewise, contact sensors and proximity sensors, though different, may share an abstract service representing binary sensors that trigger signals whenever they detect objects within their ranges. The reason for using abstract services is, therefore, to reduce the diversity of service interfaces.

An abstract service, also referred to as generic contract in DSS, is analogous to the concept of an abstract class in object-oriented programming, except there is no implementation inheritance: an abstract service consists of solely a state and an interface but no implementation. As a result, services derived from an abstract service must implement service handlers on their own based on their APIs. They can reuse the state and interface of the abstract service as-is or extend them with additional variables and functionalities. Figure 5-4 depicts two such services, the *Webcam* and *MotorisedCamera* services, in which the extended *MotorisedCamera* service may have its own parameters for describing and operations for adjusting its camera tilted angles in addition to the generic state and operations inherited from the *Camera* abstract service.

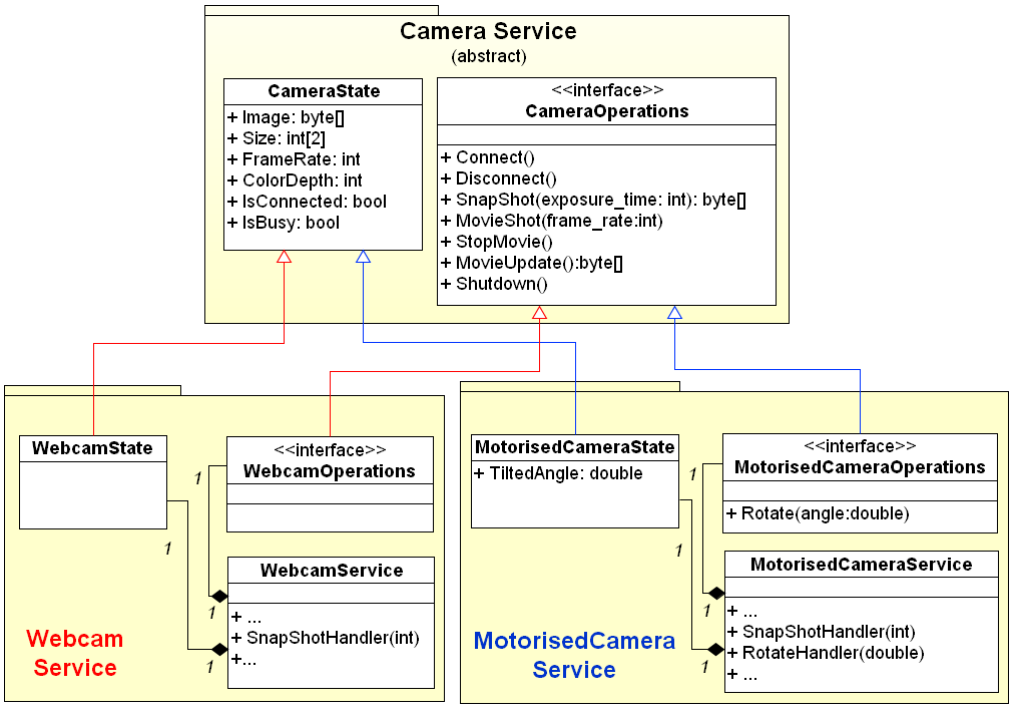


Figure 5-4 Class diagram of services derived from the abstract Camera service.

Abstract services offer many advantages whenever they are applicable. Consider a robot using two similar cameras A and B from different vendors. Without using abstract services, the robot service might have to reference with the Proxy DLLs of the two camera services, subscribe to both of them and finally end up dealing with two different conventions of *SnapShot* operations and *MovieShot* notifications at compile-time. The situation becomes even worse when it has to use a new camera C, which will require the robot service to be reprogrammed following the procedure described in section 5.1.1.2. When using the camera abstract service on the other hand, the cameras look all the same from the robot service's viewpoint and hence, it can be composed without having to know what specific cameras it is using. If it is programmed with dynamic arrays of the camera service instances (*camPort*, *camNotify*) which can grow or shrink their sizes at run-time, the robot service can connect with an arbitrary number of cameras at run-time without requiring modification. Actually, this is the general idea to achieve the plug-ability for the framework services proposed in this thesis.

5.1.2 Concurrency and Coordination Runtime

The CCR is a managed library that provides classes and methods for concurrent and asynchronous I/O programming. The CCR architecture is depicted in Figure 5-5. Unlike ordinary event-driven programming techniques which rely mainly on the event subscription and registered callback functions, the CCR derives its own abstraction layer formed by two novel concepts: *Port* and *Arbiter*. The *Port* is simply a FIFO queue for event messages sent either internally between the service's components or externally from another service. Messages posted to the ports remain there until they are consumed by corresponding receivers. The *Arbiter* enables complex logics to be applied on the receivers, such as a *Join* between two ports (two messages must arrive at them, which effectively is a logical AND) or a *Choice* between them (a message arrives at either port, creating a logical OR). This indirection allows selecting appropriate tasks (handlers) in a much simpler way, compared with the ordinary event-driven programming to achieve the same effects. Selected tasks are then scheduled to a *DispatcherQueue* and finally passed to the *Dispatcher*. The *Dispatcher* manages a pool of threads; the number of threads depends on the number of CPU/core. The threads, assigned with different priorities, will pick up the ready tasks for execution, creating a fully multi-tasking environment.

Detailed descriptions on the most important features of the CCR, the *Port* and *Arbiter* classes, are given in the Appendix E. In addition, the *Iterator*, a C# 2.0 feature that is used in a creative way by the CCR, is also introduced. The reader is recommended to read these descriptions to grasp the idea of how the CCR handles concurrency and asynchronous communications. Further information regarding the *DispatcherQueue* and *Dispatcher* can be found in (Microsoft, 2008; Johns *et.al.*, 2008).

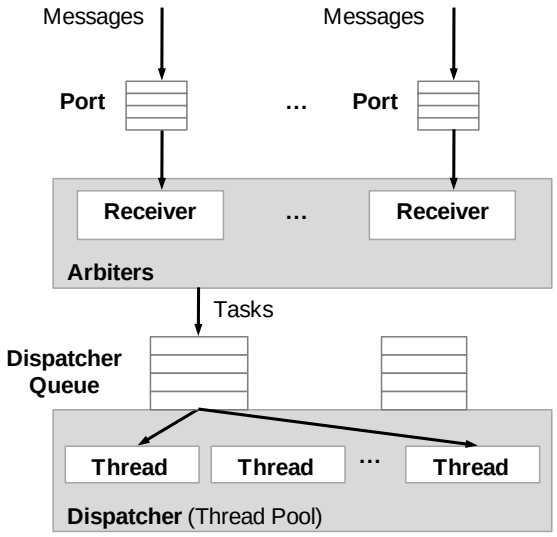


Figure 5-5 Architecture of the Concurrency and Coordination Runtime

5.2 The framework

As presented in section 5.1.1.4, abstraction is the key enabling technique for PnP integration. If there were abstract services for all types of robots, sensors, tools etc., the services of these components would be totally pluggable, i.e., removing or adding a service of these types from/to an existing system will not require any modification. However, defining standardized interfaces for a wide range of devices would result in unnecessary complications and is somewhat impractical for one person's job. Therefore, instead of having various abstract service interfaces with various I/O messages, all services in this proposed framework are derived from a single *GenericDevice* abstract service and mainly use two messages, the *CreateProcess* request and *ProcessUpdate* notification, for communication (see section 4.1.3.1). The data embodied in these messages, however, are structured in such a way that they are able to convey sufficient information as the former does. These predefined data structure (classes/enumerations) will be presented in section 5.2.1. Architecture of the *GenericDevice* abstract service and its implementation (service handlers) will be described in sections 5.2.2 and 5.2.3, respectively.

5.2.1 Predefined data structures

A full list of the data structures defined in this framework is provided in the Appendix F; this section will discuss the most important ones: *Command*, *Process* and *ProcessUpdateNotification* (Figure 5-6).

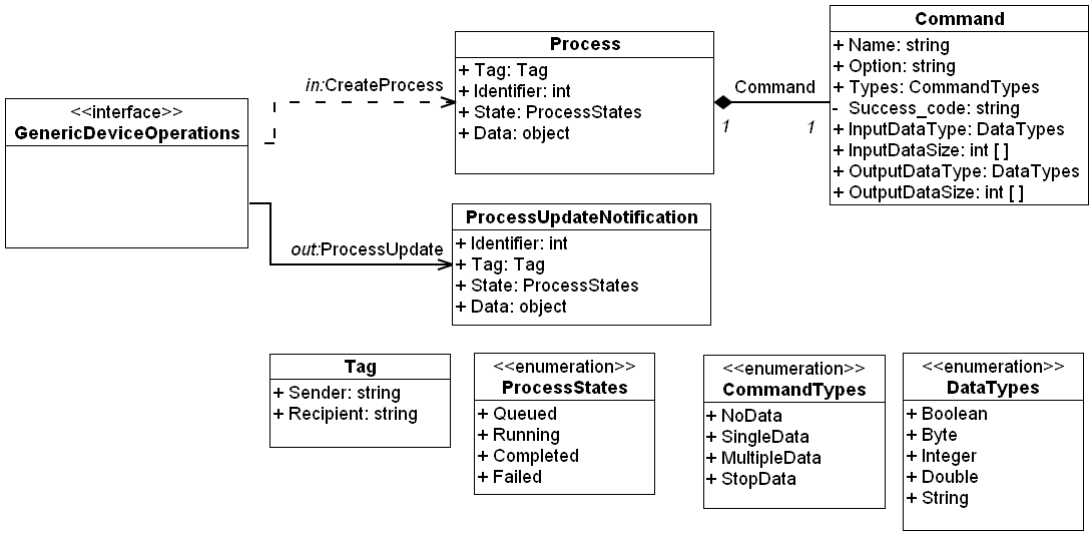


Figure 5-6 The main predefined classes/enumerations in the framework

5.2.1.1 Command

The class *Command* is used to describe a control command of a device. It contains the fields describing the command name, command type, optional parameters, sizes and types of input and output data. The command type (the field *Type*) in the class needs a few more explanations. Regardless of various naming conventions, the control commands can be generalized into four types as follows:

(a) *NoData*: This type of command does not involve a measurement. Examples of such commands are the *Connect*, *Disconnect* of the camera service in section 5.1.1, *Drill* of the tool service in Figure 4-5 of Chapter 4 which do not produce data feedback. The sender of the command is supposed to receive only a command feedback from the recipient specifying results of the command execution (whether it has been processed, completed successfully or failed to complete).

(b) *SingleData*: This type of command involves a single measurement. The sender is supposed to receive both command and data feedbacks in a single notification messsage. An example of such commands is the *SnapShot* of the camera service.

(c) *MultipleData*: This type of command involves multiple measurements. The sender is supposed to receive command feedback and a time series of data feedback in separate notification messages. An example of such commands is the *MovieShot* of the camera service.

(d) *StopData*: This type of command will deactivate commands of type *MultipleData*. An example of such commands is the *StopMovie* of the camera service.

5.2.1.2 Process

The class *Process* represents the dynamic instance of a command. It contains an identifier, names of the sender and recipient (in the field *Tag*), the command that the sender requests the recipient to perform, the command's current I/O data and status. The *Process* is used as the input parameter type of the *CreateProcess* request message.

5.2.1.3 ProcessUpdateNotification

The class *ProcessUpdateNotification* is a reduced form of the class *Process* (without the field *Command*) and is used as the output parameter type of the *ProcessUpdate* notification message. Command and data feedbacks (if any) from the recipient to the sender are enclosed in the fields *State* and *Data*.

The communication between services using the *CreateProcess* request and *ProcessUpdate* notification messages was already depicted in Figure 4-5 of Chapter 4 and are summarized as follows. When the robot service connects to the camera service, it retrieves the list of commands supported by the camera. When receiving a string initiated with "ACTIVATE" (section 4.1.3.2 of Chapter 4) from the current robot program, the robot service searches in its partners list for the given device name, then searches the command list of this device for the

given command name. A *process* is then created with the corresponding command along with its option and input data then is sent to the camera service using the *CreateProcess* message. Based on the command type (the field *Command*.*Type*), the camera knows how to dispatch it to the camera controller and setup receivers for command and data feedback. Likewise, the robot service also knows how to setup corresponding receivers on its side to intercept the command and data feedback enclosed in the *ProcessUpdate* notifications sent from the camera service and returns the results to the robot program. Notice that input and output data (the field *Data*) in the *CreateProcess* and *ProcessUpdate* messages are expressed as a generic *object* but are able to be converted to the right types and sizes thanks to the descriptions given in the fields (*Command*.*InputDataType*, *Command*.*InputDataSize*) and (*Command*. *OutputDataType*, *Command*.*OutputDataSize*), respectively. As can be seen, by having a comprehensive *Command* structure, different types of commands and data can be properly handled by the robot service without prior knowledge of the camera service. The following sections will describe the service architecture and implementation that necessitate this mechanism.

5.2.2 Service architecture

In this section, architecture of the *GenericDevice* service, the base of all services in the framework, will be described. The *GenericDevice* service is a DSS abstract service, i.e., it contains solely a state (the class *GenericDeviceState*) and an interface (the class *GenericDeviceOperations*). A camera service, derived from the *GenericDevice* service thereby consists of three classes:

1) *CameraState* sub-classed from the *GenericDeviceState.*
2) *CameraOperations* sub-classed from the *GenericDeviceOperations.*
3) *CameraService* which contains implementations of the operations defined in the class *CameraOperations* (Figure 5-7).

The classes *GenericDeviceState* and *GenericDeviceOperations* will be described in sections 5.2.2.1 and 5.2.2.2, respectively. Though the service

GenericDevice does not contain its implementation, a design template for it will also be given in section 5.2.3 through the class *CameraService* so that all other services sub-classed from it can reuse this template systematically.

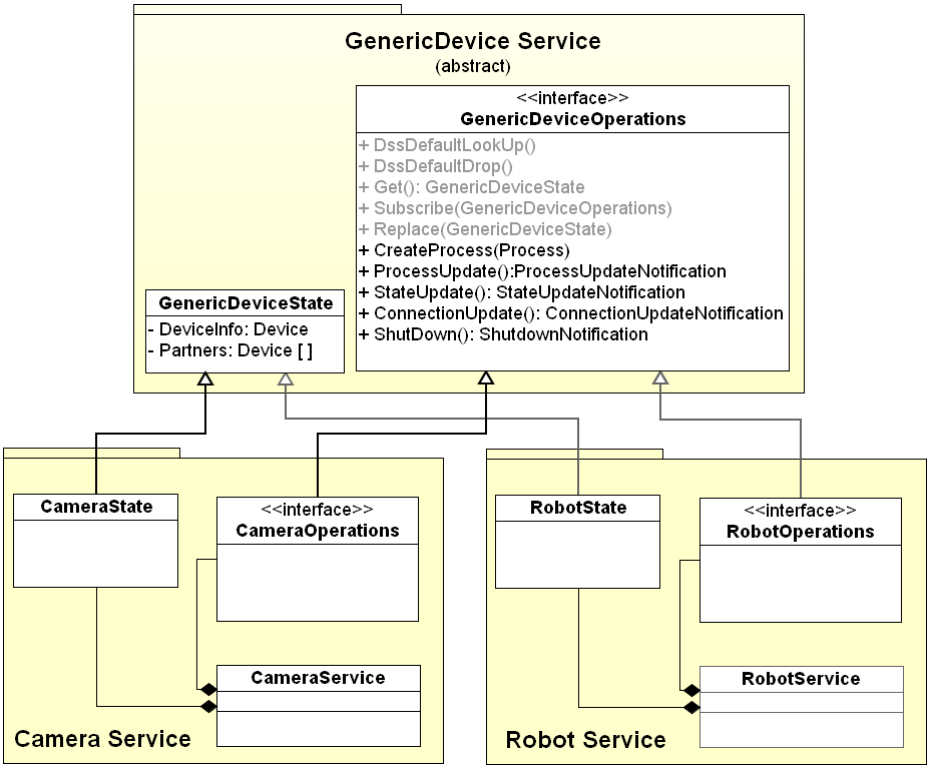


Figure 5-7 Class diagram of the *GenericDevice* abstract service and two examples, the *Camera* and *Robot* services, derived from it.

5.2.2.1 The generic state

The class *GenericDeviceState* and its aggregation are depicted in Figure 5-8. It contains essential information about the device the service represented and a list of the service's partners (Table 5-1). For example, the field *DeviceInfo* of the class *RobotState* will contain information about the robot while the field *Partners* will contain information on the devices the robot is using (tools, sensors etc.). The *DeviceInfo* is initialized by the service when it is activated while the list *Partners* will be populated when it is subscribed to other services at run-time. *DeviceInfo* and *Partners* are of class *Device* which includes the device name, vendor and its current statuses etc. As introduced earlier, the most important part of the class *Device* is a list of commands that the device can performs. The

commands are of type *Command* which includes the name, type of command, types and size of its input and output data. Using this contract information, services are possible to dispatch a command and handle its feedbacks properly.

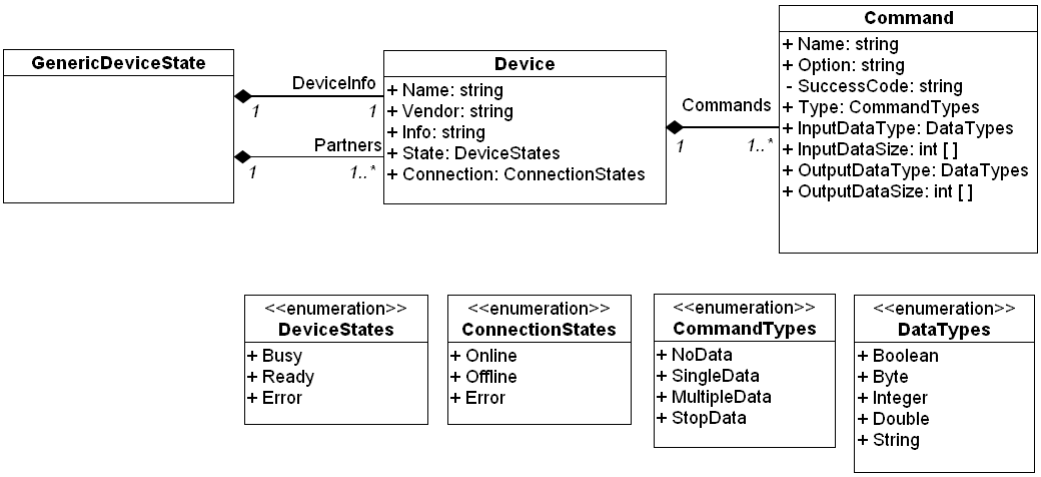


Figure 5-8 Class diagram of the class *GenericDeviceState*.

Table 5-1 Members of the class *GenericDeviceState*

Member	Type	Description
DeviceInfo	Device	Information about the device that the service represents.
Partners	Device [ ]	List of the service's partners.

5.2.2.2 The generic interface

The class *GenericDeviceOperations* and its associations are depicted in Figure 5-9. It contains standard operations, listed in Table 5-2, designed for the communication of PnP services in the framework.

Among these operations, the first five are default to every DSS service providing their basic functionality. The definitions and implementations of these operations can be found in RDS documentations (Microsoft, 2008). For example, a service can invoke the method *Get* to retrieve the whole state of another service. The last five operations involve interactions between the service with others in publish/subscribe manner: the *CreateProcess* is the request (input) message the subscribers send to the service whereas the

ProcessUpdate, *StateUpdate*, *ConnectionUpdate* and *Shutdown* are notification (output) messages the service sends to its subscribers. The input and return parameters of these messages, encapsulated in the classes *Process*, *ProcessUpdateNotification*, *StateUpdateNotification*, *ConnectionUpdateNotification* and *ShutdownNotification* are described in further details in Appendix E9-12.

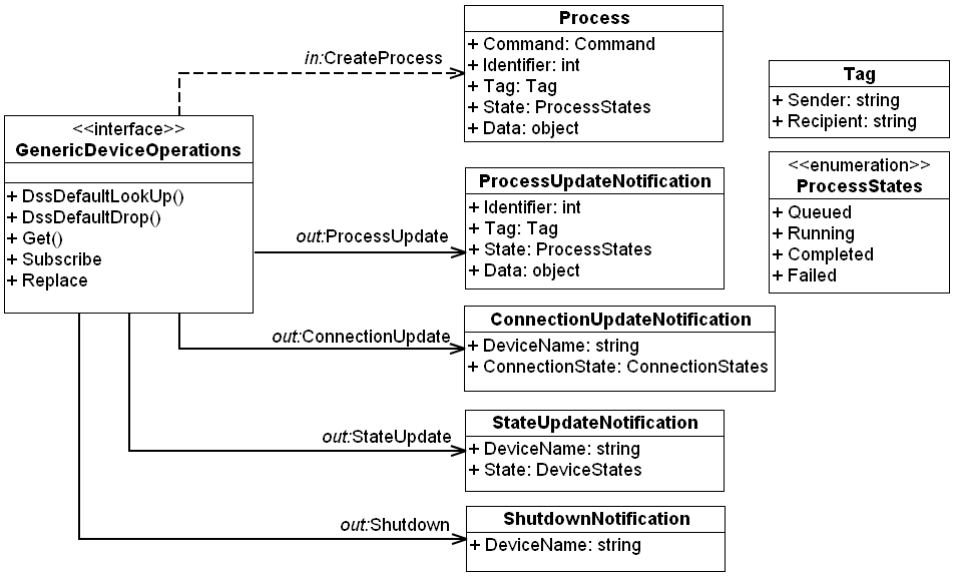


Figure 5-9 Class diagram of the class *GenericDeviceOperations*

Table 5-2 Standard methods of the class *GenericDeviceOperations*

Method	Description
DsspDefaultLookup DsspDefaultDrop Get Subscribe Replace	Default operations of DSS services.
CreateProcess	Allow service subscribers to generate a process at the service.
ProcessUpdate	Notify service subscribers that the executed process has updated new status or data.
StateUpdate	Notify service subscribers that the device the service represents has changed status.
ConnectionUpdate	Notify service subscribers that the connection between the service and its device has changed its status.
Shutdown	Notify service subscribers that the service has been shutdown

5.2.3 Service implementation

As introduced earlier, the service *GenericDevice* does not have an implementation (i.e., the class *GenericDeviceService*). A design template for it, however, will be provided via the class *CameraService*. Any service representing a physical device can reuse the template to shorten the development time because they have the same structure.

Figure 5-10 displays the functional block diagram for the class *CameraService*. Conceptually, it consists of two layers: the DSS layer operating on top of the *Camera Interface* layer as follows.

- The DSS layer is responsible for processing inbound and sending outbound messages. It defines a bunch of CCR's *Ports*, including a FIFO task queue for sequential processing (see section 4.1.4 of Chapter 4), and their corresponding receivers (handlers). Descriptions on these ports and receivers will be introduced in sections 5.2.3.1 and 5.2.3.2, among which the most important handler, the *ProcessHandler*, will be described in detail in section 5.2.3.4. In addition to these components, the DSS layer also contains a dynamic array of instances of the *GenericDevice* service (the lists *genericPort* for sending outbound request and *genericNotify* for receiving inbound notifications with service partners) as well as a *submgrPort* through which the service sends notifications to its subscribers.
- The *Camera Interface* layer is responsible for the communication between the DSS layer and the camera controller and is built upon the specific API of the camera used. The layer is composed of two main modules, namely the *CommandDispatcher* for dispatching commands and *FeedbackReceiver* for receiving status/data feedbacks from the camera controller. The feedbacks are classified and forwarded to corresponding ports of the DSS layer, where they are processed by the receivers. Services derived from the *GenericDevice* abstract service are only different from each other by this layer.

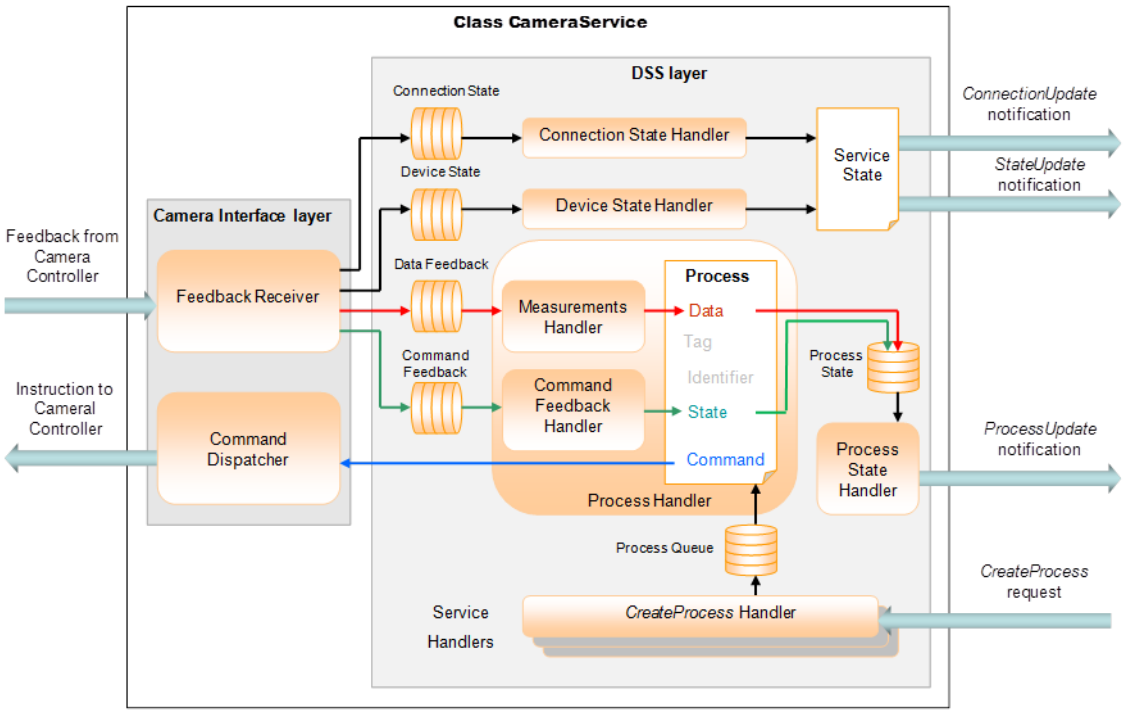


Figure 5-10 Functional blocks of the class *CameraService*

The class diagram of the above components of the class *CameraService* is depicted in Figure 5-11.

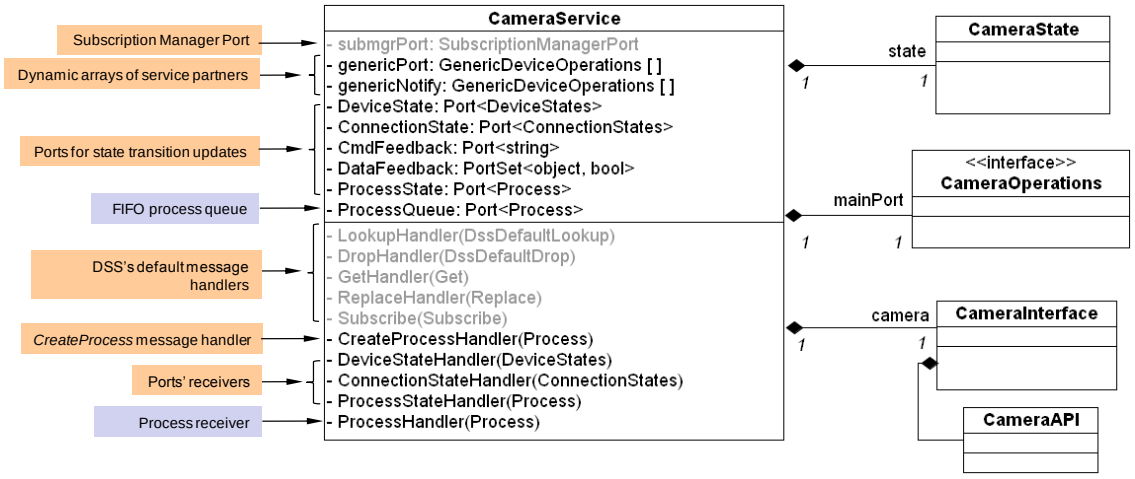


Figure 5-11 Class diagram of the class *CameraService*

5.2.3.1 The ports

The class *CameraService* defines six internal CCR's *Ports*/*PortSets* (Appendix E.1) listed in Table 5-3, among which the first five are for conveying feedback

information from the camera industrial controller to their corresponding receivers and the last one serves as a FIFO process queue. Notice the *DataFeedBack* is a *PortSet* instead of *Port* which accepts two types of messages: measurement data and booleans. The reason will be explained in section 5.2.3.4.

Table 5-3 Defined ports and receivers of the class CameraService

Port/Portset	Data Type	Receiver	Receiver type
DeviceState	DeviceStates	*DeviceStateHandler*	Persisted / Exclusive
ConnectionState	ConnectionStates	*ConnectionStateHandler*	Persisted / Exclusive
CmdFeedback	string	*CommandFeedbackHandler*, created by the *ProcessHandler*	Non-persisted/ Concurrent
DataFeedback	object, boolean	*MeasurementsHandler*, created by the *ProcessHandler*	Non-persisted/ Concurrent
ProcessState	Process	*ProcessFeedbackHandler*	Persisted / Concurrent
ProcessQueue	Process	*ProcessHandler*	Persisted / Concurrent

5.2.3.2 The receivers

Registered to the ports are their receivers (Appendix E.2.1) that trigger actions whenever there are data (messages) sent to the ports at run-time. Most of them are used simply for updating the feedbacks from the camera controller to other services. For example, a change in the device status will be sent to the *DeviceState* port. As a result, the *DeviceStateHandler* function is triggered, which firstly writes down the new status to the service state, then sends the *StateUpdate* notification to the service's subscribers as shown below.

```
public void DeviceStateHandler(DeviceStates state_upd)
{
    // Update new status into the service state
    state.DeviceInfo.State = state_upd;
    // Send the notification to subscribers via the submgrPort
    StateUpdate notification = new StateUpdate(state.DeviceInfo.Name,state_upd);
    base.SendNotification(submgrPort, notification);
}
```

5.2.3.3 Service start-up

```
using submgr = Microsoft.Dss.Services.SubscriptionManager;
class CameraService : DsspServiceBase
{
    // Declare the main service port
    CameraOperations mainPort = new CameraOperations();

    // Declare the service state
    CameraState state = new CameraState();

    // Declare the Camera Interface API
    CameraInterface camera = new CameraInterface();

    // Declare the Ports
    Port<DeviceStates>          DeviceState     = camera.DeviceState;
    Port<ConnectionStates>      ConnectionState = camera.ConnectionState;
    Port<string>                CmdFeedback     = camera.CmdFeedback;
    PortSet<object, bool>       DataFeedback    = camera.DataFeedback;
    Port<Process>               ProcessState    = new Port<Process>();
    Port<Process>               ProcessQueue    = new Port<Process>();

    // Declare the Subsciption Manager Port (for service subscribers)
    [SubscriptionManagerPartner]
    submgr.SubscriptionManagerPort submgrPort = new submgr.SubscriptionManagerPort();

    // Declare the dynamic arrays of GenericDevice service instances (for service partners)
    List<GenericDeviceOperations> genericPort = new List<GenericDeviceOperations>();
    List<GenericDeviceOperations> genericNotify = new List<GenericDeviceOperations>();

    protected override void Start()
    {
        base.Start();

        // Setup the Interleave
        MainPortInterleave.CombineWith (
           new TeardownReceiverGroup(
              Arbiter.Receive <DsspDefaultDrop> (false, mainPort, DropHandler)
           ),
           new ExclusiveReceiverGroup(
              Arbiter.Receive <Subscribe> (true, mainPort, SubscribeHandler),
              Arbiter.Receive <Replace> (true, mainPort, ReplaceHandler),
              Arbiter.Receive <DeviceStates>(true, DeviceState, DeviceStateHandler),
              Arbiter.Receive <ConnectionStates> (true, ConnectionState,
                                                  ConnectionStateHandler),
           ),
           new ConcurrentReceiverGroup(
              Arbiter.Receive <Get> (true, mainPort, GetHandler),
              Arbiter.Receive <DsspDefaultLookup> (true, mainPort, LookupHandler),
              Arbiter.Receive <CreateProcess> (true, mainPort, CreateProcessHandler),
              Arbiter.Receive <ProcessStates>(true, ProcessState, ProcessStateHandler)
           )
        );

        StateInitialize();              // Initialize the state
        SpawnIterator(ProcessHandler); // Activate the ProcessHandler that monitors the ProcessQueue
    }
    …
}
```

At start-up, the service instantiates the following components:

- The main service port (the instance *mainPort* of class *CameraOperations*),
- The state (the instance *state* of class *CameraState*),
- The camera interface (the instance *camera* of class *CameraInterface*),
- The *Ports* listed in Table 5-3,
- The Subscription Manager Port (the instance *submgrPort* of class *SubscriptionManagerPort*),
- The dynamic arrays of GenericDevice interface (the arrays *genericPort* and *genericNotify* of class *GenericDeviceOperations*).

In the main entry of the class (the function *Start*), an *Interleave* arbiter (Appendix E.2) is defined to categorize persistent receivers of the main service port and those of the *Ports* listed in Table 5-3 into corresponding concurrent and exclusive groups. Finally, the function invokes another function *StateInitialize*, shown below, to initialize the list of commands that the service supports then activates the *ProcessHandler*, the main receiver that monitors and processes the task queue.

```
private void StateInitialize()
{
    // Initialize the DeviceInfo structure
    state.DeviceInfo = new Device();
    state.DeviceInfo.Name = "CAMERA";
    state.DeviceInfo.Info = "Specs: 640x480, 8bit Grayscale, Framerate: 80fps";
    state.DeviceInfo.Vendor = "Cognex";

    // Initialize the list of supported commands
    state.DeviceInfo.Commands = new List<Command>();

    // Command 1 – SNAPSHOT
    Command command = new Command();
    command.Name = "SnapShot";
    command.Option = "";
    command.SuccessCode = "S";
    command.Type = CommandTypes.SingleData;              // Single output data
    command.InputDataType = DataTypes.Integer;           // Input data is an integer
    command.InputDataSize = new Int32[1] { 1 };          // which is the exposure time in msec
    command.OutputDataType = DataTypes.Byte;             // Output data is an image
    command.OutputDataSize = new Int32[2] { 640, 480 }; // which is a 2D-array of byte
    state.DeviceInfo.Commands.Add(command);              // Add the command to the list

    // Command 2 – MOVIE SHOT
    …
}
```

The function *StateInitialize* firstly initializes the *DeviceInfo* structure of the service state then the list of supported commands by the camera. The first command is the *SnapShot* which is of type *SingleData*, accepts one input parameter of type Integer (the exposure time) and outputs a gray-scale (8-bit) image having resolution of 640 pixels in width, 480 pixels in height. Other commands are also declared in the same manner and added to the list.

5.2.3.4 Inbound message handling

The main responsibility of the camera service is processing the *CreateProcess* messages requested by service subscribers and respond to them feedback information (status, data) from the camera controller. This activity is performed by the receiver *ProcessHandler* which constantly monitors the availability of processes at the port *ProcessQueue*. This is the most important function in the class *CameraService* and will be described in the following.

Suppose the camera service receives a *CreateProcess* message requested by another service. When the message arrives at the main service port, it is intercepted by the *CreateProcessHandler* function, which retrieves the *Process* instance embodied in the message and en-queues it to the port *ProcessQueue* as shown below (Figure 5-10).

```
public void CreateProcessHandler(CreateProcess request)
{
    // Post the received process into the ProcessQueue port
    ProcessQueue.Post(request.Body);

    // Send a acknowledgement receipt to the sender
    request.ResponsePort.Post(DefaultUpdateResponseType.Instance);
}
```

Normally, when the port *ProcessQueue* has no item, the receiver *ProcessHandler* is halted by the CCR's command *yield return* (Appendix E.3). When there is a process available and no other process is being executed, *ProcessHandler* resumes its execution immediately and retrieves the new process out of the port.

The execution of the function *ProcessHandler* will continue as follows.

1. If the command attached in the process involves measurement, the function sets up the *MeasurementsHandler* for measurement data arriving at the *DataFeedback* port (from the camera controller).

```
public IEnumerator<ITask> ProcessHandler()
{
    Process process = null;
    while (true)
    {
        // Apply a receiver at the port ProcessQueue for a process arrived at the port

        yield return Arbiter.Receive(false, ProcessQueue,
            delegate(Process pr)
            {
                // Re-scheduling: if the previous process involves multiple
                // measurements, the received process should be the one
                // that terminates it

                if (process.Command.Type == CommandTypes.MultipleData)
                    if (pr.Command.Type == CommandTypes.StopData &&
                        process.Tag.Sender == pr.Tag.Sender)
                        process = pr;
                    else
                    {
                        // Othewise, it will be posted back to the ProcessQueue

                        ProcessQueue.Post(pr);
                        continue;
                    }
                else process = pr;
            });

        // If the process involves measurement, set up the Measurement Handler
        // for incomming data from the camera controller

        if (process.Command.Type == CommandTypes.SingleData ||
            process.Command.Type == CommandTypes.MultipleData)
        {
            SpawnIterator<Process>(process, MeasurementsHandler);
        }

        // Send the command embodied in the process to the camera controller for
        // execution

        camera.CommandDispatcher(process.Command.Name, process.Command.Option,
                                 process.Data);

        // Set the process state as Running and post to the port ProcessState

        process.State = ProcessStates.Running;
        ProcessState.Post(process);

        // Set the device state as Busy and post to the port DeviceState

        DeviceState.Post(DeviceStates.Busy);
```

(continue on next page…)

(continued from last page…)

```
        // Set up the CommandFeedBack handler for command feedback from the
        controller

        yield return Arbiter.Receive(false, CmdFeedback,
            delegate(string feedback)
            {
                // When the feedback arrives:
                // If the command has been successfully completed,
                if (feedback == process.Command.SuccessCode)
                {
                    // Set the process state as Completed and post to the port

                    process.State = ProcessStates.Completed;
                    ProcessState.Post(process);

                    // Set the device state as Ready and post to the port

                    if (process.Command.Type != CommandTypes.MultipleData)
                    {
                        DeviceState.Post(DeviceStates.Ready);
                    }

                    // If the process is to stop multiple measurements and it has been
                    // completed, dispose the Measurements Handler

                    if (process.Command.Type == CommandTypes.StopData)
                    {
                        DataFeedBack.Post(false);
                    }
                }

                // If the command has failed to completed,

                else
                {
                    // Set the process state as Failed and post to the port

                    process.State = ProcessStates.Failed;
                    ProcessState.Post(process);

                    // Dispose the Measurements Handler, if any

                    DataFeedBack.Post(true);
                }
            });
    }
}
```

2. The command is forwarded to the function *CommandDispatcher* of the class *CameraInterface* for dispatching to the camera controller. The process status is then set as *Running* and the sensor status is set as *Busy* and posted to their ports.

3. The function creates the *CommandFeedbackHandler* receiver at the *CmdFeedBack* port for command feedback from the camera controller.

When the feedback message arrives, the service updates the process and sensor states accordingly.

4. If the command has been terminated with either *Completed* or *Failed* status, the function *ProcessHandler* is looped back to its start waiting for the next process.

- Notes:

1. Although the execution of the *ProcessHandler* looks sequential, it is actually segmented into multi-threads running concurrently. By posting the device and process statuses to their ports, the *ProcessHandler* leaves the tasks of updating the information to the ports' receivers and continues its execution without having to wait until the updates complete. In the same manner, the two receivers for measurement and command feedbacks were created and run side by side. Therefore, it does not restrict which one must arrive first at their ports.

2. The purpose of defining *DataFeedBack* as a *PortSet* instead of a *Port* is for disposing the *MeasurementsHandler*, shown below, when it is no longer needed. Without disposal, many instances of the *MeasurementsHandler* might listen to the *DataFeedBack* port at the same time and incoming measurement data could be assigned to a wrong process. Thanks to CCR's *Arbiter*.*Choice*, various conditions for deletion can be simply handled:
   a. If the command involves single data measurement (i.e., *SnapShot*), the receiver is dismissed automatically after the first data arrives at the port *DataFeedBack*.
   b. If the command involves multiple measurements (i.e., *MovieShot*), the receiver remains until receiving a boolean *false* sent by the code *DataFeedBack*.*Post(false)* which is in turn triggered by the command that stops the measurement (i.e., *StopMovie*) from the *ProcessHandler*.
   c. If the command has failed to complete (due to improper parameter settings, for example), the receiver is dismissed after receiving a boolean *true* sent by the code *DataFeedBack*.*Post(true)* from the *ProcessHandler*.

```
private IEnumerator<ITask> MeasurementsHandler(Process process)
{
    bool bContinue = true;
    while (bContinue == true)
    {
        // Apply a Choice at the portset DataFeedback
        yield return Arbiter.Choice(DataFeedback,
            delegate(object data)       // <---- If the data is received
            {
                // Update it to the the process's Data field and post to the port
                process.Data = data;
                _ProcessState.Post(process);

                // If the process involves a single measurement, the handler will be terminated
                if (process.Command.Type == CommandTypes.SingleData)
                    bContinue = false;
            },
            delegate (bool running)     // <---- If a boolean is received
            {
                // Depends on the boolean value and process state, the handler will be terminated
                bContinue = running;
                if (process.State == ProcessStates.Failed)
                    bContinue = false;
            }
        );
    }
 }
```

3. Finally, the receiver at the *ProcessState* port, the *ProcessStateHandler,* will send the *ProcessUpdate* notification to the subscribers whenever it receives an update forwarded from the *ProcessHandler* and *MeasurementsHandler* as shown in the code snippet below.

```
public void ProcessStateHandler(Process process)
{
    // Send the notification to subscribers via the submgrPort
    ProcessUpdate notification = new ProcessUpdate ( process.Identifer,
                                                     process.Tag,
                                                     process.State,
                                                     process.Data);
    base.SendNotification(submgrPort, notification);
}
```

The activities of the *ProcessHandler* and *MeasurementsHandler* are summarized in Figure 5-12.

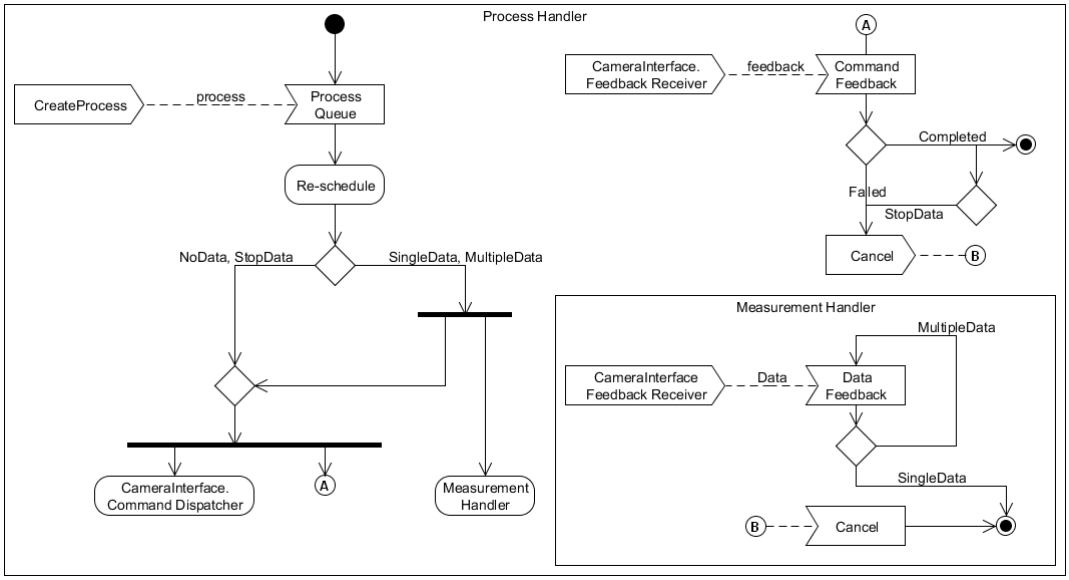


Figure 5-12 Activity diagram of the function *ProcessHandler* of the service *CameraService*

5.2.3.5 Task re-scheduling

Assuming at a specific time, the *Camera* service may receive simultaneous *CreateProcess* requests from other services, the default processing order of these requests (first in first served) may cause logical failures. Let's take an example when there are three services, *robotA*, *robotB* and *robotC*, subscribing to the *Camera* service. If all the services only request commands involving no data or single data feedback, the order that *Camera* dispatches these commands is not critical because they are self-terminated processes. However, when one of the services requests a command involving multiple data feedback (e.g., the *MovieShot* command), failures might happen if the next command is not the one that terminates it (e.g., the *StopMovie* command that sent by the same robot service). In such a case, items in the port *ProcessQueue* will be reordered until properly sequenced as depicted in Figure 5-13. Process re-scheduling is handled in the first delegate of the function *ProcessHandler*.

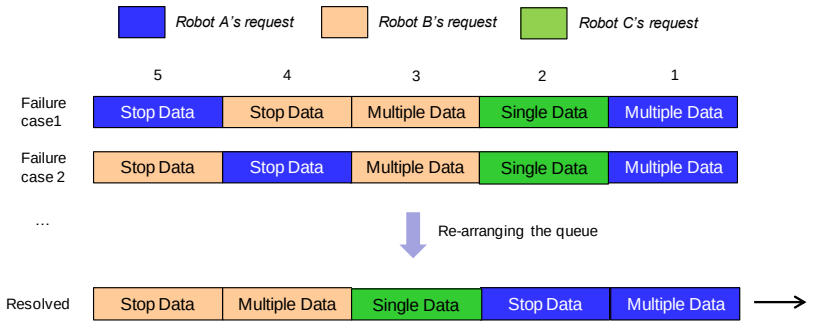


Figure 5-13 Scheduling method for the process queue

5.2.3.6 Outbound message handling

Section 5.2.3.4 has described how the camera service handles the *CreateProcess* request from other services and sends the *ProcessUpdate* notifications to them. On the other hand, a robot service must also know how to generate the request to other services and handle their notifications. The following procedure takes place automatically by the robot service to connect and communicate with an arbitrary service (e.g., the camera service) at runtime:

- When the robot service subscribes to the camera service, it retrieves the *DeviceInfo* structure of the camera (by using the request *Get*) and adds this information structure to the list *state*.*Partners*. Its dynamic arrays *genericPort* and *genericNotify* are also increased one instance, which is the client stub of the remote camera service. It also creates an *Interleave* that categorizes the receivers of the camera's notifications messages that it is interested in.
- When the robot service receives a string (e.g., "ACTIVATE CAMERA #SnapShot##, 500") from the robot program, it searches in the list *state*.*Partners* for the device given the name (e.g., "CAMERA") then searches in the list *DeviceInfo*.*Commands* of the corresponding device for the command given the name (e.g., "SnapShot"). The index number of the device in the list *state*.*Partners* is obtained and through the corresponding instance in the list *genericPort*, the *CreateProcess* message will be sent to the camera service on the network. The subsequent process will then be carried out as depicted in Figure 5-14 in a similar manner as what has been

previously done in the method *ProcessHandler* shown Figure 5-12, thanks to the concise structure of the classes *Process* and *Command*. Therefore, the programmer can benefit from duplicating the code of the *ProcessHandler* for creating the method *ExternalProcessHandler* of the class *RobotService* without much effort.

- When the robot service receives a *Shutdown* notification notifying that the camera service has left the network, it deletes the corresponding entries in the list *state*.*Partner,* the arrays *genericPort* and *genericNotify* as well as disposes the *Interleave* for this camera.

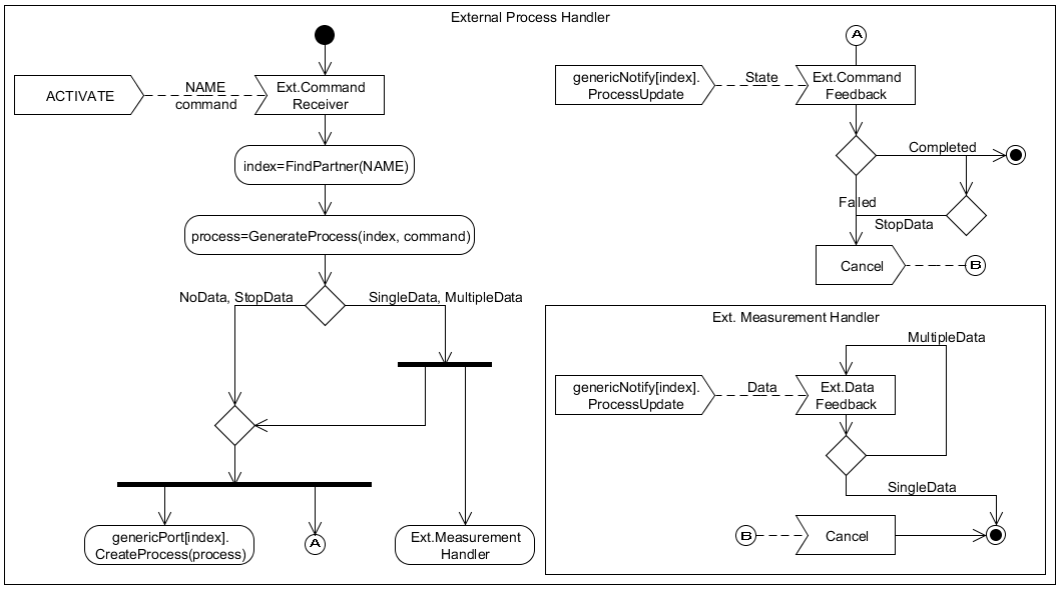
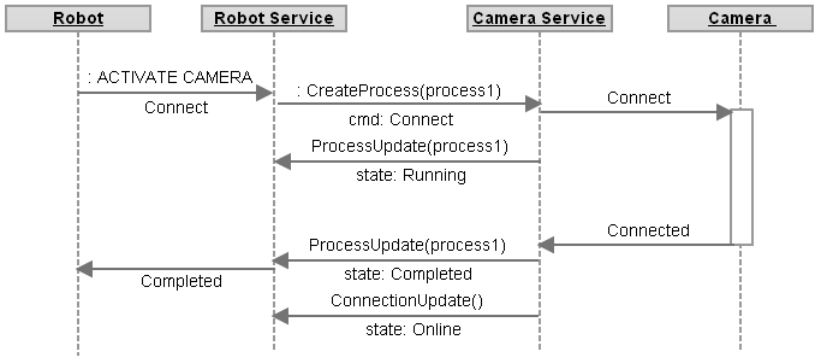
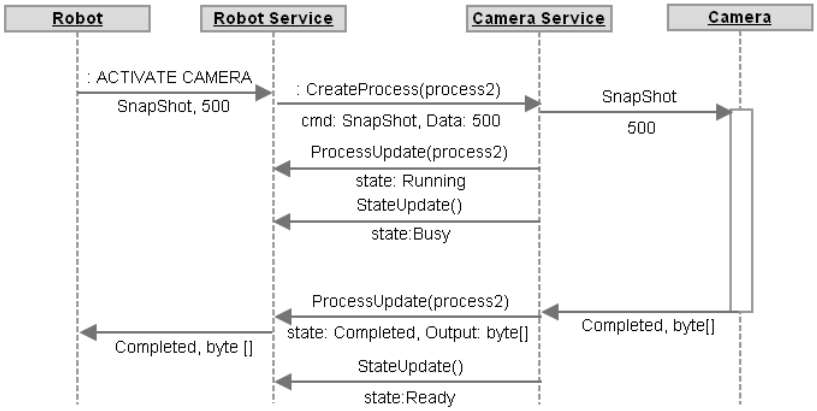


Figure 5-14 Activity diagram of the function *ExternalProcessHandler* of the service *RobotService*.

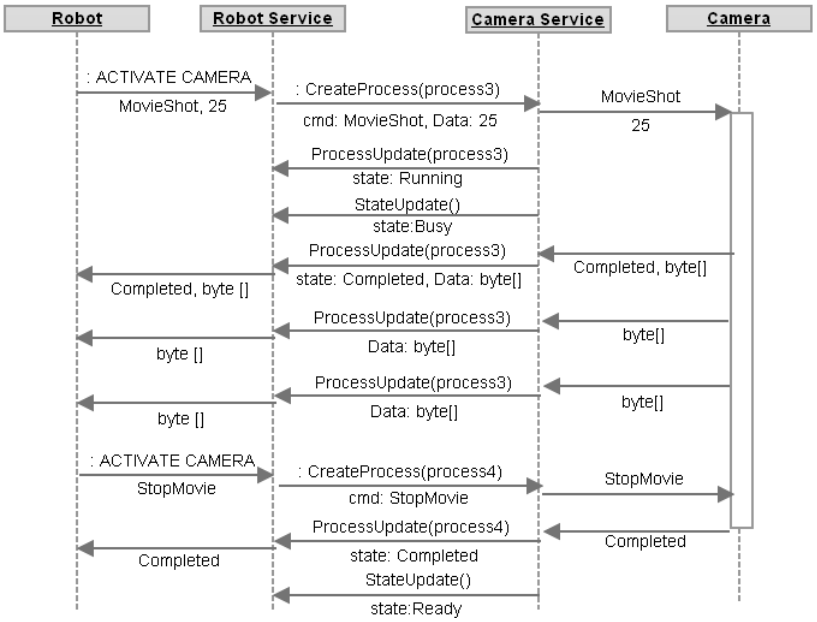
Figure 5-15 summarizes interactions between a robot and a sensor (e.g., a camera) on the exchange of the four command types listed in section 5.2.1. From what has been discussed thus far, it can be seen that services in the framework are able to connect/disconnect and establish communication channels with other services automatically without knowing their interfaces in advance. The interconnection of services, therefore, can be reconfigured without having to modify manually following the procedure described in section 5.1.1.3 at compile-time.



a. Commands involve no measurement



b. Commands involve a single measurement



c. Commands involve multiple measurements

Figure 5-15 Interactions between a robot and a camera through their services on the exchange of different command types

5.3 Performance evaluation

5.3.1 Experiments

Performances of the framework in terms of message exchange rate (throughput) and latency have been assessed in order to validate the communication rate at which the framework can be used for dynamic correction. The exchange of messages is performed between two services developed using the design template in the previous section: one is the publisher (e.g., a camera service) and the other is the consumer (e.g., a robot service).

In the test of message exchange rate, the publisher pushes 10000 *ProcessUpdate* notifications as fast as possible toward the consumer. The test was carried out with different payloads (the amount of data attached in the *ProcessUpdateNotification*.*Data*) (Table 5-4). The message throughput r (msg/s) is calculated as the average number of the messages sent within 1 second:

$$r = \frac{10000}{t_{total}} \tag{5.1}$$

where t_{total} (s) is the time for the consumer to receive the total 10000 *ProcessUpdate* messages.

In the test of message exchange latency, the consumer sends a *CreateProcess* message with attached data (in the field *Process.Data*) to the publisher, followed by a *ProcessUpdate* notification with the same data (in the field *ProcessUpdateNotification*.*Data*) returned by the publisher. Since there are two messages (round-trip) exchanged in each cycle, the latency t_l (ms) is calculated as half of the difference between the time the *CreateProcess* message is sent and the time the *ProcessUpdate* message is received by the consumer. The process was repeated for 2000 messages and tested with various amounts of data as in the first test. Average, maximum, minimum and standard deviation (jitter) values of the latency are evaluated.

Table 5-4 Tested payloads

Data	1	2	3	4	5	6	7
Payload (Byte)	4 (1-Double)	32 (6-Doubles)	1K (256-Doubles)	4K	64K	0.3M (640x480)	0.9M (640x480x3)
Typical sensor	1D point laser	6D force sensor laser tracker	1D line scan laser			8-bit grayscale 640x480 camera	24-bit colour 640x480 camera

Each test above was implemented in two cases: when the services were running on the same computer and on two separate computers connected over a 100MBit Ethernet through a network switch (Figure 5-16). The computers used are Intel Core 2 Duo 2.4GHz on Windows XP. Timing is measured using a Windows OS' multimedia timer having resolution in the order of microseconds.

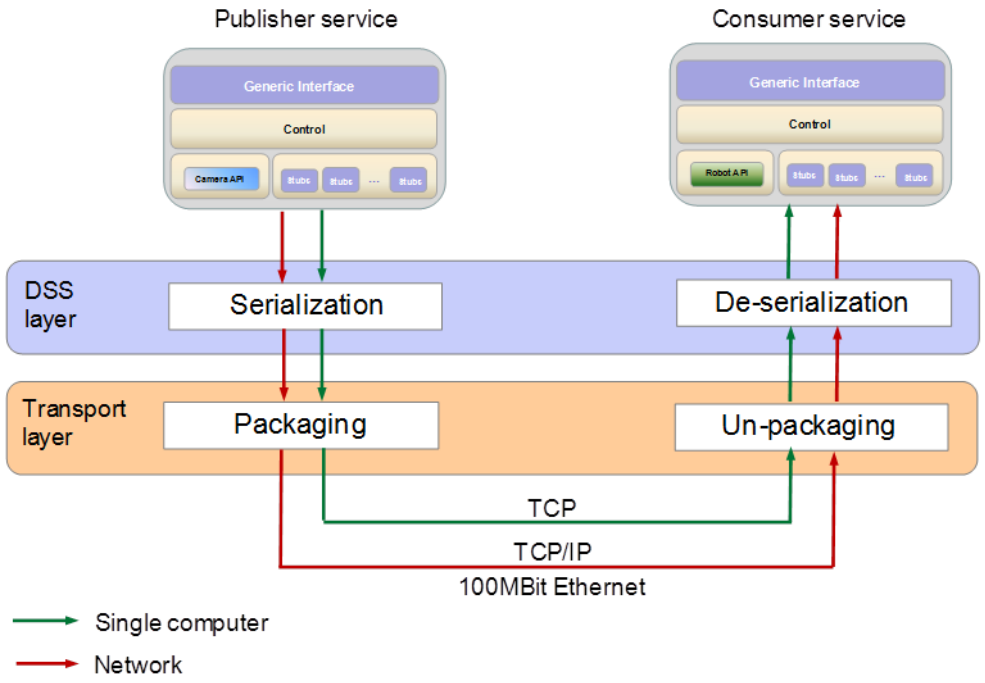


Figure 5-16 The *ProcessUpdate* notification message's flow in the tests

5.3.2 Results and discussions

The results of the message exchange between the two services running on the same computer and networked computers are given in Table 5-5 and

Table **5-6**, respectively. The results are summarized in the following sections.

Table 5-5 Message throughput and latency between two local services

Data	Payload (Byte)	Throughput r (msg/s)	Latency t_l (ms)			
			Avg.	Max.	Min.	Std.Dev.
1	4	>10000	0.23	3.09	0.21	0.05
2	32	>10000	0.30	3.33	0.22	0.06
3	1K	8333	0.32	3.45	0.26	0.31
4	4K	6250	2.01	5.51	1.26	0.91
5	64K	725	4.23	8.82	3.00	1.34
6	0.3M	80	8.80	11.39	5.38	1.57
7	0.9M	46	23.69	28.31	18.33	1.90

Table 5-6 Message throughput and latency between two networked services

Data	Payload (Byte)	Throughput r (msg/s)	Latency t_l (ms)			
			Avg.	Max.	Min.	Std.Dev.
1	4	>5000	0.62	6.46	0.42	0.60
2	32	>5000	0.54	1.75	0.42	0.14
3	1K	3500	0.84	2.24	0.67	0.15
4	4K	1385	1.36	4.36	1.21	0.42
5	64K	90	13.53	17.18	12.67	1.32
6	0.3M	19	60.62	63.07	57.26	1.33
7	0.9M	6	178.34	184.12	174.07	2.10

5.3.2.1 Throughput

It can be seen from the tables that the framework is able to transmit more than 1000 msg/s when the payload size is smaller than 64KB. This is considerably sufficient for most dynamic correction applications which are usually implemented at update rates smaller than 1KHz (typically at 250Hz) and with small sensor data (e.g., those of force sensors, laser trackers). However, the framework is not entirely suitable for the transmission of large data, e.g., camera images, especially via the network. To handle such a situation in practice, the camera service might need to perform image processing locally

then transfer only the results via the network to the robot service rather than the raw image (in a similar way to a seam-tracking sensor).

5.3.2.2 Latency

As depicted in Figure 5-16, the latency t_l in the arrival of one message is the combination of:

- the time for serializing the data into SOAP message by the DSS middleware and packaging the message into TCP's segments.
- the time for transmitting the message using TCP (when the services reside on one computer) and TCP/IP (when the services reside on networked computers).
- the time for unpacking the TCP's segments to the SOAP message and de-serializing the message by the DSS middleware.

Among them, the times used for serializing/de-serializing and transmitting the data over the Ethernet (through the network switch) are the dominating sources of latency when the payload size is large.

The average latency of the framework is less than 1ms when the payload size is smaller than 4KB. This might also be sufficient for dynamic correction as long as the force or position control algorithms can be calculated within the remaining 3ms for the 250Hz update rate. However, it is not sufficient under the worst case latency. The maximum latency in both test cases is sometimes a lot higher than the averages plus 3 times the standard deviations, meaning that these outliers are originated by some abnormal activities in the infrastructure. Indeed, when the services reside on one computer, the peak latency might be caused by a Windows' system process having higher priority or when the computer CPU is under high stress (because it must perform both the queuing and de-queuing of messages of the two services). When the services reside on networked computers, the peak latency might also be caused by the TCP's error correction mechanism (retransmission of lost segments); however, these late data might be as bad as the lost ones in the control point of view.

The latency may affect the throughput drastically since the TCP utilizes a flow control algorithm that adjusts the network bandwidth based on detected latency in order to guarantee the delivery of their transmitted segments. The UDP, on the other hand, is relatively immune to latency; however, it does not detect message losses (Parziale *et.al.*, 2006).

From what has been discussed in this part, it can be concluded that the safe update frequency of the framework for dynamic correction considering the worst case latency (6ms) and data smaller than 4KB is around 100Hz. This is apparently not sufficient for force control applications, but is for visual servoing and seam-tracking applications whose vision systems used usually have less communicate rates. A higher update rate might also be possible; however, it must be provided with some sort of error correction strategy (e.g., extrapolation of the sensor data) when the data are not delivered within the determined time frame. The reasons that limit the communication rate of the framework are originated from both the non real-time characteristics of Windows OS and features of the TCP communication.

6 CALIBRATION AND ERROR COMPENSATION FOR SERIAL ROBOTS HAVING A PARALELLOGRAM LINKAGE

This chapter presents the theoretical work on calibration and error compensation for serial robots having closed-chain mechanisms, particularly ones with a parallelogram linkage. The robot examined is a Comau Smart H4, a long reach and heavy duty industrial manipulator. However, the same approach could be applied to other robots with a similar structure. Error modelling for geometric parameters and non-geometric parameters, specifically, joint deflection, is introduced in sections 6.2 and 6.3. Algorithms for identification and compensation for errors in the robot structure will be presented in sections 6.4 and 6.5.

6.1 Robot forward kinematic model

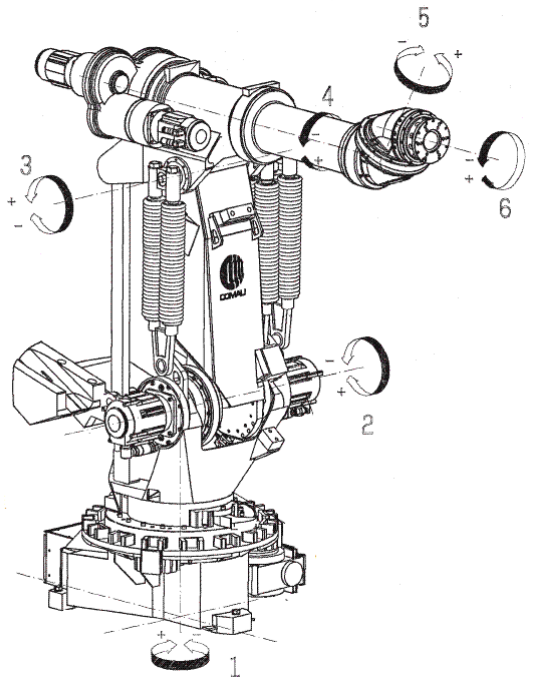


Figure 6-1 The Comau Smart H4 robot (Comau Robotics, 1998)

The Comau Smart H4 is a 6dof parallelogram linkage type manipulator with a load capacity of 200kg, maximum horizontal and vertical reaches of 2.318 and 2.720 meters, respectively. Joint 2 of the robot is a double revolute joint actuated by two co-axial motors, one for driving the upper arm and another for the forearm via the parallelogram structure (Figure 6-1). The repeatability of the robot quoted by Comau is ±0.3mm.

The nominal forward kinematic model of the Smart H4 manipulator is computed following the method described in (Siciliano *et.al*, 2007). Joint 3 of the robot is virtually cut open, allowing link frames to be assigned with the DH convention (Figure 6-2). Joint 3', 4' are passive joints driven by actuated joints 2 and 2' via the parallelogram structure formed by links 2', 3', 4' and 2. Notice there are two frames at the cut joint 3: frame F_2 describes the relation between links 2 and 3 whereas frame $F_{4'}$ between links 4' and link 3. Nominal DH parameters of the robot are given in Table 6-1.

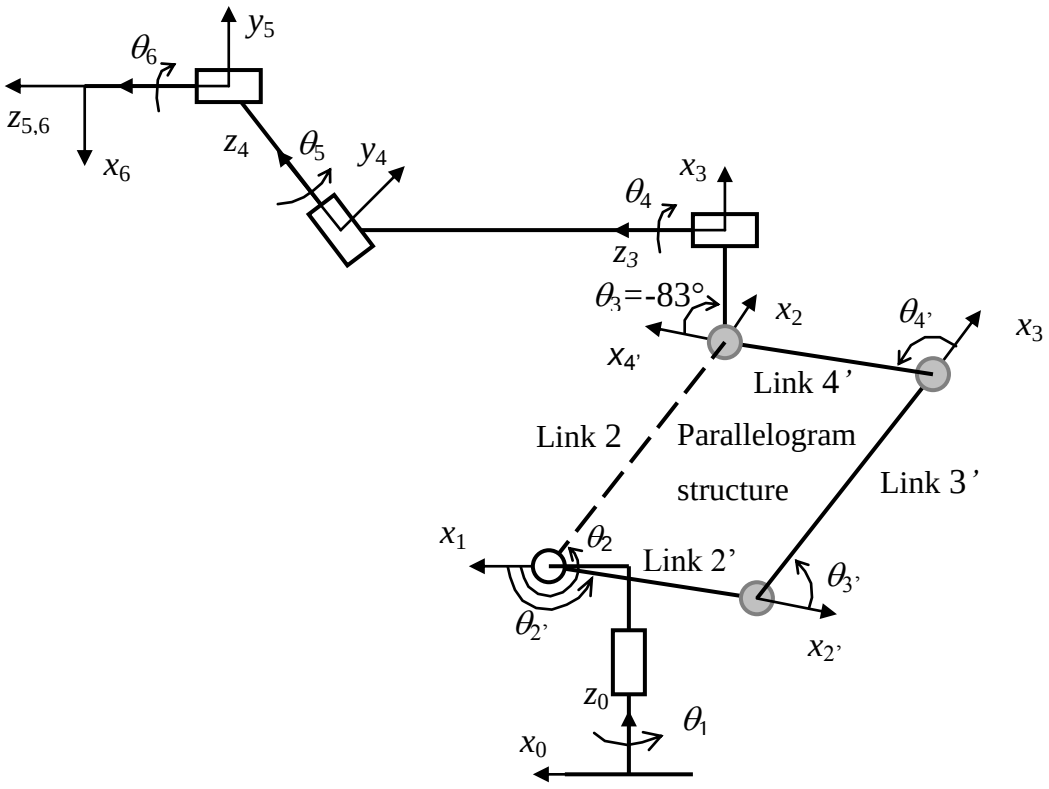


Figure 6-2 Schematic diagram of the Smart H4 robot with DH frame assignments (passive joints are marked in gray colour).

Location of the end frame F_6 with respect to (w.r.t.) the base frame F_0 is represented as:

$$T_6^0 = T_1^0 T_3^1 T_4^3 T_5^4 T_6^5 \tag{6.1}$$

where T_3^1 can be expressed either with the branch (12'3'4'3) :

$$T_3^1 = T_{2'}^1 . T_{3'}^{2'} . T_{4'}^{3'} . T_3^{4'} \tag{6.2}$$

or equivalently, with the branch (123):

$$T_3^1 = T_2^1 . T_3^2 \tag{6.3}$$

Table 6-1 Nominal DH parameters of the Smart H4 robot

Link	θ_i (°)	d_i (m)	a_i (m)	α_i (°)
l_1	θ_1	1	0.2	-90
l_2^*	θ_2	0	1.05	0
$l_{2'}$	$\theta_{2'}$	0	0.45	0
$l_{3'}$	$\theta_{3'}$	0	1.05	0
$l_{4'}$	$\theta_{4'}$	0	0.45	0
l_3	-83	0	0.25	-90
l_4	θ_4	1.1395	0	-60
l_5	θ_5	1.1588	0	60
l_6	θ_6	0.2176	0	0

(* : link does not belong to the chain 12'…6)

In this work, equation (6.1) was computed using (6.2) in order to account for parameters in the parallelogram mechanism. The transformation matrices $T_{2'}^1, T_{3'}^{2'}, T_{4'}^{3'}, T_3^{4'}$ in equation (6.2) are with joint angles ($\theta_{2'}$, $\theta_{3'}$, $\theta_{4'}$, θ_3=-83°) where the passive joint variables $\theta_{3'}$, $\theta_{4'}$ must be computed w.r.t. θ_2 and $\theta_{2'}$ via resolutions of the closure constraints in the position and orientation between frame 4' and frame 2. As illustrated in Figure 6-3, z_4' must align with z_2 and the origin $O_{4'}$ must align with O_2 ($d_{4'2}$=0), thus these constraints are:

- Orientation constraint : $z_{4'}^1(q') = z_2^1(q)$ (6.4)

- Position constraint : $$p_{4'}^{1}(q') - p_{2}^{1}(q) = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \tag{6.5}$$

In order to solve the constraints, the position and orientation of frame $F_{4'}$ and F_2 w.r.t. the common frame F_1 are firstly expressed as:

$$T_{4'}^{1}(q') = T_{2'}^{1} T_{3'}^{2'} T_{4'}^{3'} = \begin{bmatrix} c_{2'3'4'} & -s_{2'3'4'} & 0 & a_{2'}c_{2'} + a_{3'}c_{2'3'} + a_{4'}c_{2'3'4'} \\ s_{2'3'4'} & c_{2'3'4'} & 0 & a_{2'}s_{2'} + a_{3'}s_{2'3'} + a_{4'}s_{2'3'4'} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.6}$$

where $q' = [\theta_{2'}, \theta_{3'}, \theta_{4'}]$ and

$$T_{2}^{1}(q) = \begin{bmatrix} c_2 & -s_2 & 0 & a_2 c_2 \\ s_2 & c_2 & 0 & a_2 s_2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.7}$$

where $q = \theta_2$.

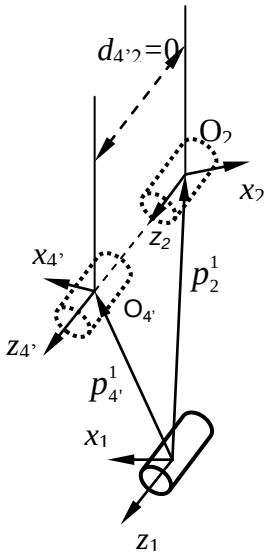


Figure 6-3 The closure constraints between frame 4' and frame 2 at cut joint 3

One might see that the orientation constraint (6.4) is satisfied regardless of q and q'. The position constraint (6.5) is extracted as:

$$\begin{aligned} a_{2'}c\theta_{2'} + a_{3'}c\theta_{2'3'} + a_{4'}c\theta_{2'3'4'} - a_2c\theta_2 = 0 \\ a_{2'}s\theta_{2'} + a_{3'}s\theta_{2'3'} + a_{4'}s\theta_{2'3'4'} - a_2s\theta_2 = 0 \end{aligned} \tag{6.8}$$

where $c\theta_i$, $s\theta_i$ and θ_{ij} short for $cos\theta_i$, $sin\theta_i$ and $\theta_i+\theta_j$, respectively.

Since $a_{4'}=a_{2'}$ and $a_{3'}=a_2$, (6.8) becomes:

$$\begin{aligned} a_{2'}(c\theta_{2'} + c\theta_{2'3'4'}) + a_2(c\theta_{2'3'} - c\theta_2) = 0 \\ a_{2'}(s\theta_{2'} + s\theta_{2'3'4'}) + a_2(s\theta_{2'3'} - s\theta_2) = 0 \end{aligned} \tag{6.9}$$

which leads to the following solutions, given arbitrary choice of a_2 and $a_{2'}$:

$$\begin{aligned} \theta_{3'} = \theta_2 - \theta_{2'} \\ \theta_{4'} = \pi - \theta_2 + \theta_{2'} \end{aligned} \tag{6.10}$$

The forward kinematic model of open chain (12'…3…6) is thus solved.

6.2 Modelling of geometric errors

6.2.1 Modelling of errors in the robot's internal parameters

Firstly, assuming the open chain (12'...3…6) composed of 8 links is driven by all actuated revolute joints. As introduced in section 4.2.1.1 of Chapter 4, the initial error model of the manipulator is:

$$\Delta x = H(g)\Delta g \tag{6.11}$$

In the above equation, $\Delta x = (\Delta p_x, \Delta p_y, \Delta p_z, \Delta\phi_x, \Delta\phi_y, \Delta\phi_z)$ is a (6×1) - vector made up three differential positions and three differential orientations of the end-effector, Δg is a (32×1) - concatenated vector of geometric errors: $\Delta g = (\Delta\theta, \Delta d, \Delta a, \Delta\alpha, \Delta\beta)$ in which $\Delta\theta, \Delta a, \Delta\alpha$ are (8×1) - vectors such that $\Delta\theta = (\Delta\theta_1, \Delta\theta_{2'}, ..., \Delta\theta_6)$ and so forth, $\Delta d = (\Delta d_1, \Delta d_3, ..., \Delta d_6)$ is a (5×1) - vector and $\Delta\beta = (\Delta\beta_{2'}, \Delta\beta_{3'}, \Delta\beta_{4'})$ is a (3×1) - vector. Recall that $\Delta\beta$ is the additional Hayati parameters to handle consecutively parallel axes between joints 2', 3' and 4' (section 4.2.1.4). $H(g)$ is the (6×32) – identification Jacobian matrix relating

Δx and Δg. The columns $J_{\theta i}, J_{di}, J_{ai}, J_{\alpha i}, J_{\beta i}$, $i=1...n,$ for a n - link serial manipulator are given as:

$$H_{\theta i} = \begin{bmatrix} z_{i-1} \times p_{i-1} \\ z_{i-1} \end{bmatrix},\ H_{di} = \begin{bmatrix} z_{i-1} \\ 0 \end{bmatrix},\ H_{ai} = \begin{bmatrix} x_i \\ 0 \end{bmatrix},$$
$$H_{\alpha i} = \begin{bmatrix} x_i \times p_i \\ x_i \end{bmatrix},\ H_{\beta i} = \begin{bmatrix} y_i \times p_i \\ y_i \end{bmatrix} \tag{6.12}$$

where x_i, y_i, z_i are directional vectors and p_i is the position of frame F_i in the base frame F_0. Clearly, this model is not yet complete since the passive joint angles $\Delta\theta_{3'}, \Delta\theta_{4'}$ are not independent and thus, an internal relationship between them with errors in the parallelogram linkage's parameters must be derived.

Considering when the parallelogram structure has errors in its parameters (Figure 6-4), it degenerates into a four bar linkage. As a result, actual passive joint angles $\theta_{3'}, \theta_{4'}$ deviate from their nominal values $\widehat{\theta}_{3'}, \widehat{\theta}_{4'}$ computed in (6.10) as:

$$\begin{aligned} \theta_{3'} &= \widehat{\theta}_{3'} + \Delta\theta_{3'} \\ \theta_{4'} &= \widehat{\theta}_{4'} + \Delta\theta_{4'} \end{aligned} \tag{6.13}$$

Indeed, they relate to other parameters of the linkage as:

$$\begin{aligned} \theta_{3'} &= f(\theta, a) \\ \theta_{4'} &= h(\theta, a) \end{aligned} \tag{6.14}$$

where $\theta = (\theta_2, \theta_{2'})$; $a = (a_2, a_{2'}, a_{3'}, a_{4'})$; f and h are position functions of a general four bar linkage (Appendix C).

Assuming $\Delta\theta$ and Δa are small, errors in passive joint angles $\Delta\theta_{3'}, \Delta\theta_{4'}$ can be derived from the linearization of (6.14):

$$\begin{aligned} \Delta\theta_{3'} &= \left.\frac{\partial f}{\partial \theta}\right|_{\Delta a=0} \Delta\theta + \left.\frac{\partial f}{\partial a}\right|_{\Delta\theta=0} \Delta a \\ \Delta\theta_{4'} &= \left.\frac{\partial h}{\partial \theta}\right|_{\Delta a=0} \Delta\theta + \left.\frac{\partial h}{\partial a}\right|_{\Delta\theta=0} \Delta a \end{aligned} \tag{6.15}$$

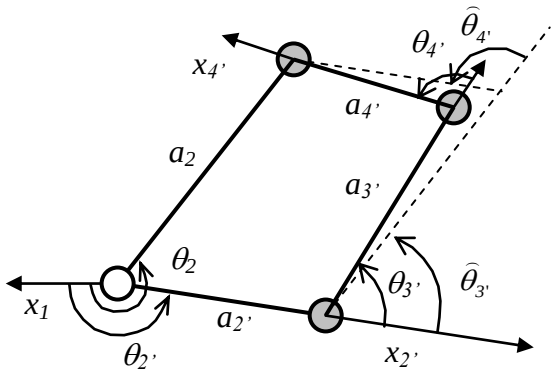


Figure 6-4 A degenerated parallelogram with uneven link lengths.

This approach, however, result in highly nonlinear and complicated equations. In this work, the results of equation (6.15) can be obtained without actually linearizing f and h.

The first items on the right hand side of (6.15) are passive joint angle errors due to joint offsets $\Delta\theta_2, \Delta\theta_{2'}$ only and thus can be obtained from derivatives of equation (6.10):

$$\begin{aligned} \Delta\theta_{3'} &= \Delta\theta_2 - \Delta\theta_{2'} \\ \Delta\theta_{4'} &= -\Delta\theta_2 + \Delta\theta_{2'} \end{aligned} \tag{6.16}$$

Likewise, the second items on the right hand side of (6.15) are passive joint angle errors due to deviations of parallelogram link lengths only. Considering when $a_i = \hat{a}_i + \Delta a_i$, the position constraint (6.9) becomes:

$$\begin{aligned} &(\hat{a}_{2'} + \Delta a_{2'})c\hat{\theta}_{2'} + (\hat{a}_{3'} + \Delta a_{3'})c(\hat{\theta}_{2'3'} + \Delta\theta_{3'}) + (\hat{a}_{4'} + \Delta a_{4'})c(\hat{\theta}_{2'3'4'} + \Delta\theta_{3'} + \Delta\theta_{4'}) \\ &\qquad - (\hat{a}_2 + \Delta a_2)c\hat{\theta}_2 = 0 \\ &(\hat{a}_{2'} + \Delta a_{2'})s\hat{\theta}_{2'} + (\hat{a}_{3'} + \Delta a_{3'})s(\hat{\theta}_{2'3'} + \Delta\theta_{3'}) + (\hat{a}_{4'} + \Delta a_{4'})s(\hat{\theta}_{2'3'4'} + \Delta\theta_{3'} + \Delta\theta_{4'}) \\ &\qquad - (\hat{a}_2 + \Delta a_2)s\hat{\theta}_2 = 0 \end{aligned} \tag{6.17}$$

where:

$$\begin{aligned} &\hat{a}_{2'} = \hat{a}_{3'}, \hat{a}_{4'} = \hat{a}_2 \\ &\hat{\theta}_{2'3'} = \hat{\theta}_{2'} + (\hat{\theta}_2 - \hat{\theta}_{2'}) = \hat{\theta}_2 \\ &\hat{\theta}_{2'3'4'} = \hat{\theta}_{2'} + \hat{\theta}_{3'} + \pi - \hat{\theta}_{3'} = \pi + \hat{\theta}_{2'} \end{aligned}$$

Simplifying (6.17) by ignoring high order terms and using the linearization forms of *cosine* and *sine* functions:

$$c(\hat{\theta}_i + \Delta\theta_i) = c\hat{\theta}_i - s\hat{\theta}_i\Delta\theta_i$$
$$s(\hat{\theta}_i + \Delta\theta_i) = s\hat{\theta}_i + c\hat{\theta}_i\Delta\theta_i$$

one finally obtains:

$$\begin{aligned}\Delta\theta_{3'} &= m_{11}(\Delta a_{3'} - \Delta a_2) + m_{12}(\Delta a_{4'} - \Delta a_{2'})\\ \Delta\theta_{4'} &= m_{21}(\Delta a_{3'} - \Delta a_2) + m_{22}(\Delta a_{4'} - \Delta a_{2'})\end{aligned} \tag{6.18}$$

where:

$$m_{11} = \frac{1}{\hat{a}_2}.\frac{c(\hat{\theta}_2 - \hat{\theta}_{2'})}{s(\hat{\theta}_2 - \hat{\theta}_{2'})}, \qquad m_{12} = -\frac{1}{\hat{a}_2}.\frac{1}{s(\hat{\theta}_2 - \hat{\theta}_{2'})}$$
$$m_{21} = \frac{1}{\hat{a}_{2'}}.\frac{1}{s(\hat{\theta}_2 - \hat{\theta}_{2'})} - m_{11}, \; m_{22} = -\frac{1}{\hat{a}_{2'}}.\frac{c(\hat{\theta}_2 - \hat{\theta}_{2'})}{s(\hat{\theta}_2 - \hat{\theta}_{2'})} - m_{12} \tag{6.19}$$

Combining equations (6.16) and (6.18) yields:

$$\begin{aligned}\Delta\theta_{3'} &= \Delta\theta_2 - \Delta\theta_{2'} + m_{11}(\Delta a_{3'} - \Delta a_2) + m_{12}(\Delta a_{4'} - \Delta a_{2'})\\ \Delta\theta_{4'} &= \Delta\theta_{2'} - \Delta\theta_2 + m_{21}(\Delta a_{3'} - \Delta a_2) + m_{22}(\Delta a_{4'} - \Delta a_{2'})\end{aligned} \tag{6.20}$$

with m_{ij} given in (6.19). This is the internal error model of the parallelogram structure to be merged with the existing open chain error model. Indeed, equation (6.11) can be written as:

$$\begin{aligned}\Delta x &= H_\theta\Delta\theta + H_d\Delta d + H_a\Delta a + H_\alpha\Delta\alpha + H_\beta\Delta\beta\\ &= .. + H_{\theta 2'}\Delta\theta_{2'} + H_{\theta 3'}\Delta\theta_{3'} + H_{\theta 4'}\Delta\theta_{4'} + .. + H_{a2'}\Delta a_{2'} + H_{a3'}\Delta a_{3'} + H_{a4'}\Delta a_{4'} + ..\end{aligned} \tag{6.21}$$

By replacing $\Delta\theta_{3'}, \Delta\theta_{4'}$ in (6.21) by (6.20) then re-arranging the resulting matrix equation, we obtained the desirable kinematic error model for serial manipulators having the parallelogram mechanism. One might realize the existences of $\Delta\theta_2, \Delta a_2$ in the resulting vector Δg to be identified even though link 2 is not part of the kinematic chain (12'…3…6). New Jacobian coefficients for the parameters in (6.20) are obtained as:

$$
\begin{aligned}
H_{\theta 2} &= H_{\theta 3'} - H_{\theta 4'} \\
H_{\theta 2'} &= H_{\theta 2'} - H_{\theta 3'} + H_{\theta 4'} \\
H_{a 2'} &= H_{a 2'} - m_{12} H_{\theta 3'} - m_{22} H_{\theta 4'} \\
H_{a 4'} &= H_{a 4'} + m_{12} H_{\theta 3'} + m_{22} H_{\theta 4'} \\
H_{a 3'} &= H_{a 3'} + m_{11} H_{\theta 3'} + m_{21} H_{\theta 4'} \\
H_{a 2} &= -m_{11} H_{\theta 3'} - m_{21} H_{\theta 4'}
\end{aligned}
\tag{6.22}
$$

where the formulation of H_{gi} on the right hand side was given in (6.12).

6.2.2 Modelling of errors in the base and tool transformations

In order to solve the error model, measurements of the robot poses using an external sensor (e.g., a laser tracker) are usually required. However, it is usually not possible to measure directly the (virtual) robot's end frame F_6 but a target (e.g., the TMAC) relatively fixed in it. Furthermore, the measurements are usually expressed in the sensor's predefined frame instead of the robot base frame. In such case, equation (6.1) must be pre-multiplied and post-multiplied by two additional transformations to form the measurement as:

$$T = BASE \cdot T_6^0 \cdot PROBE \tag{6.23}$$

where $BASE$, $PROBE$ are the transformations defining the robot base frame F_0 w.r.t. the metrology frame F_m and the target frame F_t w.r.t. the flange frame F_6, respectively (Figure 6-5). Parameters of these two constant transformations are usually determined beforehand (see Appendix D) and will be identified to a higher degree of accuracy. Of the 12 parameters required to model inaccuracies in the $BASE$ and $PROBE$ (3 positions and 3 orientations for each), only 6 are identifiable (section 4.2.1.4) and thus, these two transformations require proper coordinate arrangements.

The $BASE$ transformation can be set up as:

$$BASE = Tran(x, a_0) Tran(y, b_0) Rot(x, \alpha_0) Rot(y, \beta_0) Rot(z, \theta_0) Tran(z, d_0) \tag{6.24}$$

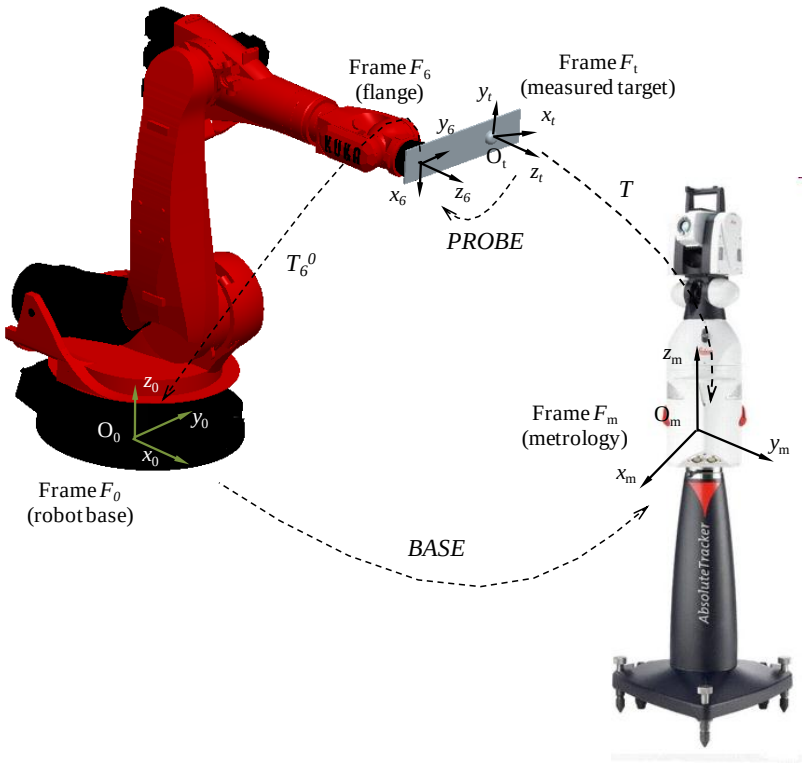


Figure 6-5 Transformations between robot and metrology systems

With reference to the first DH frame F_1 of the robot described in equation (A.2) in Appendix A, one might see that of the six parameters $\Delta a_0, \Delta b_0, \Delta\alpha_0, \Delta\beta_0, \Delta\theta_0, \Delta d_0$ that model errors in the $BASE$, only the first four are independent while $\Delta\theta_0, \Delta d_0$ are grouped into $\Delta\theta_1, \Delta d_1$. Their Jacobians were derived by symbolic programming in Matlab and are given in Appendix D. To account for errors in $PROBE$, a virtual DH - based frame F_7 is inserted after frame F_6 such that $\alpha_6 = -90°$, $\alpha_7 = 90°$, $a_7 = d_7 = 0$, following the suggestion in (Veitsschegger *et.al.*, 1988). As depicted in Figure 6-6, errors in $PROBE$ can be modelled by 3 positions $\Delta d_6, \Delta a_6, \Delta a_7$ and 3 orientations $\Delta\theta_6, \Delta\alpha_6, \Delta\theta_7$, of which only $\Delta a_7, \Delta\theta_7$ need to be identified. Therefore, errors in $BASE$ and $PROBE$ are successfully represented by 6 parameters: $\Delta a_0, \Delta b_0, \Delta\alpha_0, \Delta\beta_0$ and $\Delta a_7, \Delta\theta_7$.

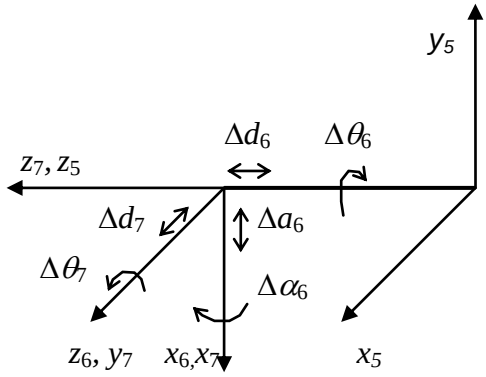


Figure 6-6 Errors in the *PROBE* transformation are modelled by $\Delta a_0, \Delta b_0, \Delta\alpha_0, \Delta\beta_0$ and $\Delta a_7, \Delta\theta_7$

6.2.3 Elimination of redundant parameters

Table 6-2 Identifiable geometric error parameters

Link i	Δg_i	Link i	Δg_i
l_0^*	$\Delta a_0, \Delta b_0, \Delta\alpha_0, \Delta\beta_0$	l_3	$\Delta d_3, \Delta a_3, \Delta\alpha_3$
l_1	$\Delta\theta_1, \Delta d_1, \Delta a_1, \Delta\alpha_1$	l_4	$\Delta\theta_4, \Delta d_4, \Delta a_4, \Delta\alpha_4$
l_2^{**}	$\Delta\theta_2, \Delta a_2$	l_5	$\Delta\theta_5, \Delta d_5, \Delta a_5, \Delta\alpha_5$
$l_{2'}^{**}$	$\Delta\theta_{2'}, \Delta a_{2'}, \Delta\alpha_{2'}, \Delta\beta_{2'}$	l_6	$\Delta\theta_6, \Delta d_6, \Delta a_6, \Delta\alpha_6$
$l_{3'}^{**}$	$\Delta a_{3'}, \Delta\alpha_{3'}, \Delta\beta_{3'}$	l_7^*	$\Delta\theta_7, \Delta d_7$
$l_{4'}^{**}$	(none)	Total	34

(*: *BASE* and *PROBE* transformations, **: parallelogram linkage).

Parameter redundancy exists due to the design of the parallelogram structure whose opposite links are nominally parallel (link 2' // link 4', link 3'// link 2). Thanks to the simple analytic form of the Jacobian matrix, redundant error parameters of the loop can be simply determined without the need to use the trivial technique presented in section 4.2.1.2 as follows:

- Since link 4' and 2' are designed to be parallel: $x_{4'}$=-$x_{2'}$, it can be seen from (6.22) that $H_{a4'}$ = -$H_{a2'}$, thus $\Delta a_{4'}$ and $\Delta a_{2'}$ are linearly dependent. Therefore, only one is identifiable, say $\Delta a_{2'}$. In contrast, despite link 3' being parallel to link 2, $\Delta a_{3'}$ and Δa_2 are independent and thus both of them are identifiable because $H_{a3'} \neq H_{a2}$.

- Similarly, $\Delta\theta_3$ and $\Delta\theta_{2'}$ are dependent terms; $\Delta\alpha_{4'}$ and $\Delta\beta_{4'}$ are dependent on ($\Delta\alpha_3, \Delta\theta_4$).

The redundant parameters must be omitted from the error vector Δg and their corresponding columns must be discarded from the identification Jacobian matrix $H(g)$. The final 34 identifiable parameters are given in Table 6-2.

6.3 Modelling of joint deflections

6.3.1 Joint deflections due to structural loading

It is straightforward that joint deflections due to structural loading occur mostly at joint 2 and 2' due to the masses of the upper arm (link 2), the forearm (link 4) and the counterweight mounted at the end of link 2'. In order to calculate $\Delta\theta_2^s$ and $\Delta\theta_{2'}^s$ in equation (4.16) in Chapter 4, it is necessary to formulate the torques τ_2 and $\tau_{2'}$ generated at these joints to counteract the gravitational forces. These torques are calculated from the static equilibrium condition at these joints.

Figure 6-7 displays the free body diagram of the forces acting on the links and joints of the robot; P_2, P_4, P_w denote the masses of link 2, 4 and the counterweight. Pins B, C, D are cut open to examine reaction forces at the pins. Because these pins do not transmit moment (M_B=M_C=M_D=0), internal forces acting at point B and C, R_B and R_C, must be in the same direction of link 3'. Furthermore, it can be seen that:

$$M_D(R_C) = -M_A(R_B) \tag{6.25}$$

where the notation $M_D(R_C)$ denotes the moment created at point D by the force R_C applied at point C and so forth.

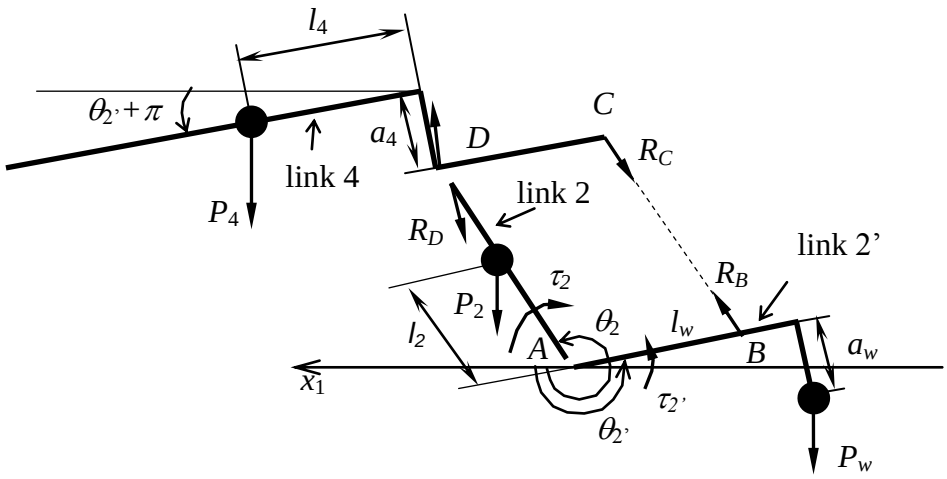


Figure 6-7 Free body diagram of forces in the x_1y_1 plane of frame 1

At point D, since $\sum F_D = 0$:

$$R_D = P_4 + R_C \tag{6.26}$$

At joint 2, since $\sum M_A = 0$:

$$\tau_2 = M_A(P_2) + M_A(R_D) \tag{6.27}$$

Substituting (6.26) to (6.27) gives:

$$\begin{aligned} \tau_2 &= M_A(P_2) + M_A(P_4) + M_A(R_C) \\ &= M_A(P_2) + M_A(P_4) \text{ since } M_A(R_C) = 0 \\ &= (P_2 l_2 + P_4 a_2) c\theta_2 \end{aligned} \tag{6.28}$$

Likewise, at joint 2', since $\sum M_A = 0$:

$$\tau_{2'} = M_A(R_B) + M_A(P_w) \tag{6.29}$$

Substituting (6.25) to (6.29) gives:

$$\begin{aligned} \tau_{2'} &= M_D(P_4) + M_A(P_w) \\ &= (P_4 l_4 c(\theta_{2'} + \pi) + P_4 a_4 s(\theta_{2'} + \pi)) + (P_w l_w c\theta_{2'} + P_w a_w s\theta_{2'}) \\ &= (P_w l_w - P_4 l_4) c\theta_{2'} + (P_w a_w - P_4 a_4) s\theta_{2'} \end{aligned} \tag{6.30}$$

By combining equation (4.15) in Chapter 4 with (6.28) and (6.30), the deflections of joint 2 and 2' due to structural loading are obtained as:

$$\begin{aligned}\Delta\theta_2^s &= G_1 c\theta_2 \\ \Delta\theta_{2'}^s &= G_2 c\theta_{2'} + G_3 s\theta_{2'}\end{aligned} \tag{6.31}$$

where:

$$\begin{aligned}G_1 &= c_2(P_2 l_2 + P_4 a_2) \\ G_2 &= c_{2'}(P_w l_w - P_4 l_4) \\ G_3 &= c_{2'}(P_w a_w - P_4 a_4)\end{aligned}$$

are dimensionless constants formed by joint compliances c_2, $c_{2'}$, link weights and distances from joint axes to mass centres. Although these parameters are unknown, G_1, G_2, G_3 are identifiable by merging equations (6.31) and (4.16) of Chapter 4 to the existing error model (6.11). The columns H_{G1}, H_{G2}, H_{G3} in the identification Jacobian matrix $H(g)$ are:

$$\begin{aligned}H_{G1} &= c\theta_2(H_{\theta 3'} - H_{\theta 4'}) \\ H_{G2} &= c\theta_{2'}(H_{\theta 3'} - H_{\theta 3'} + H_{\theta 4'}) \\ H_{G3} &= s\theta_{2'}(H_{\theta 3'} - H_{\theta 3'} + H_{\theta 4'})\end{aligned} \tag{6.32}$$

where $H_{\theta i}$ is calculated from (6.12).

6.3.2 Joint deflections due to payload

Considering the robot carries a payload with mass m (kg) and center of gravity M located at constant position $p_M^n = (x_M, y_M, z_M)^T$ (m) in the flange frame F_n (Figure 6-8). The wrench (force and moment) applied at O_n by the gravitational force P of the weight is:

$$W = \begin{pmatrix} P \\ p_M^0 \times P \end{pmatrix} = \begin{pmatrix} P \\ (R_n^0 p_M^n) \times P \end{pmatrix} \tag{6.33}$$

where the operator $\times$ denotes the cross vector product. Notice that the wrench W in (6.33) is expressed in the base frame F_0 with $z_0 // P$. With $P = (0,0,mg)^T$, g=-

9.81(m/s^2) and recall equation (A.3) in Appendix A that the rotation matrix R_n^0 describing the orientation of frame F_n w.r.t. frame F_0 has the general form of:

$$R_n^0 = \begin{bmatrix} n_x & s_x & a_x \\ n_y & s_y & a_y \\ n_z & s_z & a_z \end{bmatrix}$$

where $n_n^0 = (n_x, n_y, n_z)$, $s_n^0 = (s_x, s_y, s_z)$, $a_n^0 = (a_x, a_y, a_z)$ are the directional cosine vectors of the x_n, y_n , z_n axes of the flange frame F_n in frame F_0, equation (6.33) yields:

$$W = (0 \quad 0 \quad mg \quad (n_y x_M + s_y y_M + a_y z_M)mg \quad -(n_x x_M + s_x y_M + a_x z_M)mg \quad 0)^T \tag{6.34}$$

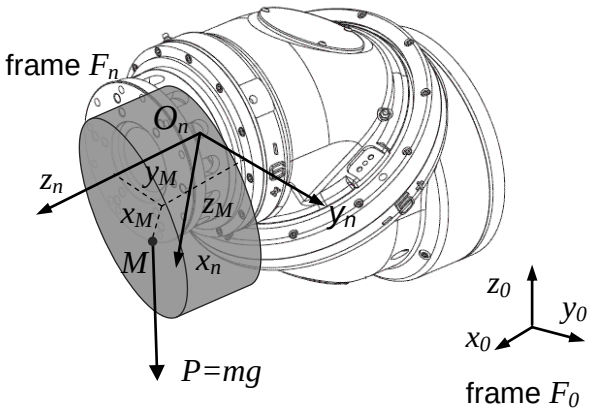


Figure 6-8 The robot carrying a deadweight.

The above wrench W is related with the torques τ at actuated joints of the robot as:

$$\tau = J(q)^T W \tag{6.35}$$

In the above equation, $\tau = (\tau_1, \tau_{2'}, \cdots, \tau_n)$ is the $(n \times 1)$ – vector of torque generated at n actuated joints to counteract with W (n=6 in this case), $J(q)$ is the $(6 \times n)$ – manipulator Jacobian matrix. For convenience, denote the transpose of $J(q)$ as K:

$$K = J(q)^T$$

K is thus a $(n \times 6)$ – matrix.

Equation (6.35) can be written for actuated joint i as:

$$\tau_i = K_i W \tag{6.36}$$

where K_i is the i^{th} row of matrix K corresponding to joint i. For actuated rotary joints 1, 4, 5 and 6 of the Smart H4 robot which belong only to the open chain, K_i has the form (Spong *et.al.*, 2004 – eq. (5.1.27)):

$$K_i = \begin{pmatrix} z_{i-1} \times (p_n - p_{i-1}) \\ z_{i-1} \end{pmatrix}^T \tag{6.37}$$

For joints 2 and 2', the torques τ_2 and $\tau_{2'}$ must also account for $\tau_{3'}$ and $\tau_{4'}$ (those at the passive joints). From [(Siciliano *et.al.*, 2007) - eq. (3.121)], τ_2 and $\tau_{2'}$ are given as:

$$\begin{aligned} \tau_2 &= \tau_2 + \tau_{3'} - \tau_{4'} = (K_{3'} - K_{4'})W \\ \tau_{2'} &= \tau_{2'} - \tau_{3'} + \tau_{4'} = (K_{2'} - K_{3'} + K_{4'})W \end{aligned} \tag{6.38}$$

Therefore:

$$\begin{aligned} K_2 &= K_{3'} - K_{4'} \\ K_{2'} &= K_{2'} - K_{3'} + K_{4'} \end{aligned} \tag{6.39}$$

where K_i on the right hand side is calculated following (6.37).

Combining equation (4.15) in Chapter 4 with (6.34) and (6.36), the deflections of joint i $\Delta\theta_i^e$ due to the deadweight can be obtained as:

$$\Delta\theta_i^e = K_i \begin{pmatrix} 0 \\ 0 \\ c_i mg \\ n_y(x_M c_i mg) + s_y(y_M c_i mg) + a_y(z_M c_i mg) \\ -n_x(x_M c_i mg) - s_x(y_M c_i mg) - a_x(z_M c_i mg) \\ 0 \end{pmatrix}$$

$$= K_i \begin{pmatrix} 0 \\ 0 \\ A_i \\ n_y B_i + s_y C_i + a_y D_i \\ -n_x B_i - s_x C_i - a_x D_i \\ 0 \end{pmatrix} \tag{6.40}$$

where the constants $A_i = c_i mg$, $B_i = x_M c_i mg$, $C_i = y_M c_i mg$, $D_i = z_M c_i mg$ are formed by the joint compliance, the deadweight's mass and position of its mass center and K_i is given in (6.37) for i=1,4,5,6 and in (6.39) for i=2,2'. Equation (6.40) can also be written as:

$$\Delta\theta_i^e = \begin{pmatrix} K_{(i,3)} & K_{(i,4)} n_y - K_{(i,5)} n_x & K_{(i,4)} s_y - K_{(i,5)} s_x & K_{(i,4)} a_y - K_{(i,5)} a_x \end{pmatrix} \begin{pmatrix} A_i \\ B_i \\ C_i \\ D_i \end{pmatrix} \tag{6.41}$$

where $K_{(i,j)}$ denotes the j^{th} element of K_i.

From this point, one might see that the four parameters A_i, B_i, C_i, D_i for joint i can be identified via the existing error model of the robot. By merging equations (6.41) and (4.16) in Chapter 4 with (6.11), the corresponding columns H_{Ai}, H_{Bi}, H_{Ci}, H_{Di} in the identification Jacobian matrix $H(g)$ are obtained as:

$$\begin{aligned} H_{Ai} &= H_{\theta i} K_{(i,3)} \\ H_{Bi} &= H_{\theta i} (K_{(i,4)} n_y - K_{(i,5)} n_x) \\ H_{Ci} &= H_{\theta i} (K_{(i,4)} s_y - K_{(i,5)} s_x) \\ H_{Di} &= H_{\theta i} (K_{(i,4)} a_y - K_{(i,5)} a_x) \end{aligned} \tag{6.42}$$

where $H_{\theta i}$, K_i on the right hand side are calculated in (6.12), (6.37) for i=1,4,5,6 and (6.22), (6.39) for i=2,2'. Notice that among the maximum of $4 \times 6 = 24$ parameters to be identified, some are unidentifiable, meaning that they have no effects on the corresponding joint deformation $\Delta\theta_i^e$. Identifiable parameters are listed in Table 6-3.

Table 6-3 Identifiable compliance parameters due to payload

Joint	Identifiable	Unidentifiable	Explanations of un-identifiablity
1	(none)	A_1, B_1, C_1, D_1	Force P // z_0: joint 1's axis
2	A_2,	B_2, C_2, D_2	$K_{(3',4)} = K_{(4',4)}$ and $K_{(3',5)} = K_{(4',4)}$, thus $H_{B2}=H_{C2}=H_{D2}=0$
2'	$A_{2'}$, $B_{2'}$, $C_{2'}$, $D_{2'}$	(none)	
4	A_4, B_4, C_4, D_4	(none)	
5	A_5, B_5, C_5, D_5	(none)	
6	B_6, C_6	A_6 D_6	$K_{(6,3)}=0$, thus $H_{A6}=0$ $K_{(6,4)}=a_x$ and $K_{(6,5)}=a_y$, thus $H_{D6}=0$
Total	15	9	

6.4 Error identification

The total 52 parameters, including 34 for modelling geometric errors (section 6.2), 3 for modelling structural loading effect (section 6.3.1) and 15 for modelling external loading effect (section 6.3.2), are concatenated into vector Δg in equation (6.11) for identification. The columns of the (6×52) - identification Jacobian matrix $H(g)$ are calculated using:

- Equation (6.12) for geometric parameters of the main open loop, including those of the $PROBE$ transformation,
- Equation (D.6) in Appendix D for parameters of the $BASE$ transformation,
- Equation (6.22) for geometric parameters of the parallelogram linkage,
- Equation (6.32) for the parameters G_1, G_2, G_3 that model the effect of link masses,
- Equation (6.42) for the parameters A_i, B_i, C_i, D_i that model the effect of the carrying deadweight.

In this work, tool poses are measured by a laser tracker providing both position and orientation components (6dof), thus the error Δx is a (6×1) – vector. Therefore, at least 9 measurements are required to solve the error model (6.11) for Δg in least square sense. The identification process is then carried out as illustrated in Figure 6-9.

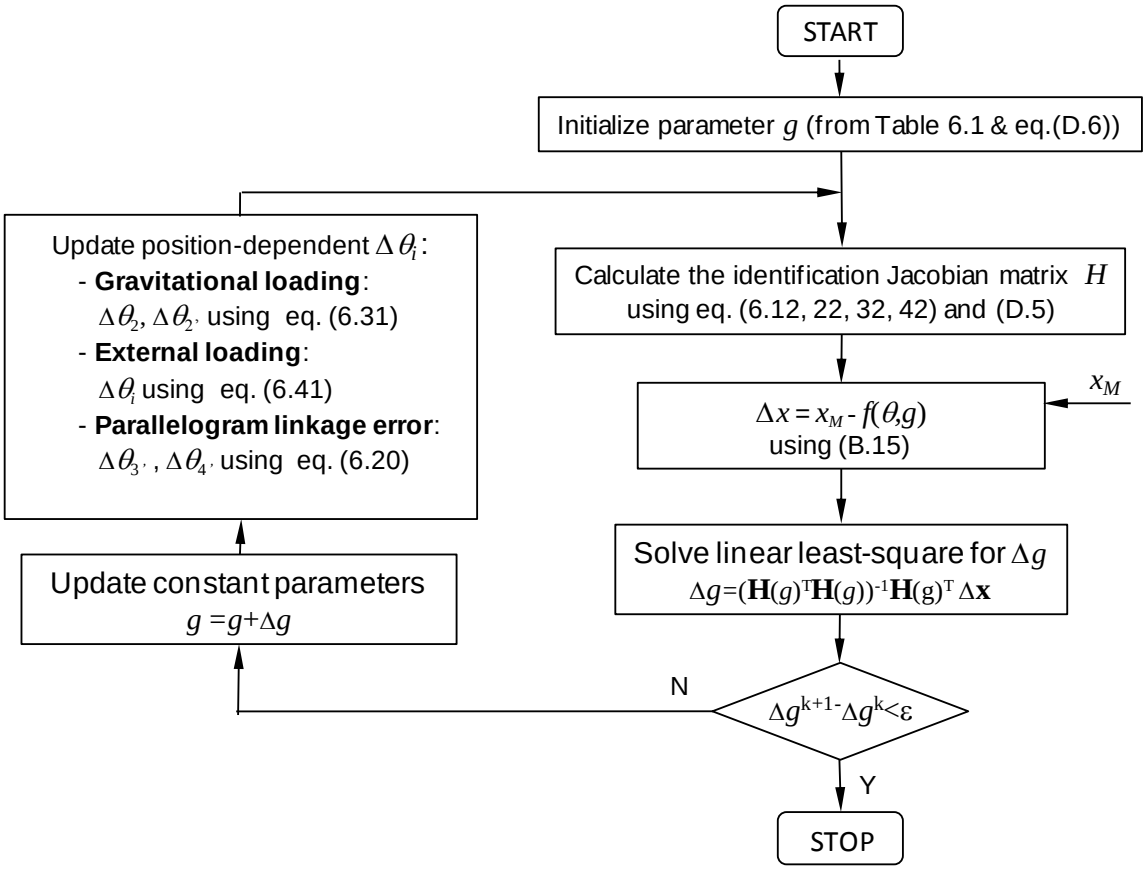


Figure 6-9 The error parameters identification algorithm

6.5 Error compensation

As introduced in Chapter 4, a two-stage error compensation scheme is adopted to improve robot positioning accuracy: firstly, tool pose errors will be compensated using the model developed thus far and secondly, the residuals will be corrected using a global metrology system.

6.5.1 Model-based error compensation

At the first stage, tool errors are compensated using the developed error model as follows:

a. From the desired position x^d, calculate the joint solution $\theta = (\theta_1, \theta_2, \theta_{2'}, \theta_4, \theta_5, \theta_6)$ using the nominal inverse kinematics. Calculate the manipulator Jacobian matrix $J(\theta)$ using equation (6.37) and (6.39).

b. Calculate actual joint values θ taking into account the errors $\Delta\theta$ as the results of joint offset, structural loading using equations (6.31), external loading using equations (6.41) and parallelogram errors using equations (6.20).

c. With the actual joint values θ, estimate the actual position x^a using the forward kinematics with other identified constant DH parameters in g (section 6.3.1).

d. Calculate the differential translation and orientation vector $\Delta x = x^d - x^a$ using equation (B.15).

e. Calculate the infinitesimal joint increment $\delta\theta = J(\theta)^{-1}\Delta x$ to compensate for Δx. Update joint values $\theta = \theta + \delta\theta$.

f. Repeat from the step (b) until Δx is sufficiently small (in this work, $\|\Delta x\| < 10^{-6}$).

g. Download the final joint values θ to the robot controller for execution.

The above correction process is depicted in Figure 6-10.

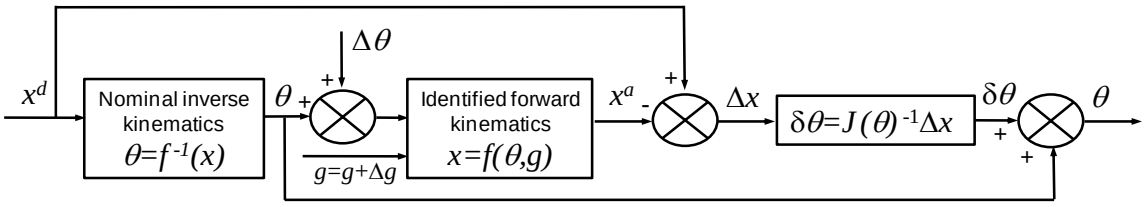


Figure 6-10 Flowchart of the model-based error compensation

6.5.2 Sensor-based error compensation

At the second stage, the global metrology system is utilised to correct residual errors of the end-effector (static correction). Two methods of correction are developed depending on whether the (generic) metrology system is able to measure full pose (6dof) or just the position components (3dof).

If 6dof measurements are provided:

a. Calculate the differential translation and orientation vector $\Delta x = x^d - x^m$ between the desired pose x^d and the measured pose x^m using equation (B.15) in Appendix B.
b. Calculate the infinitesimal joint $\delta\theta = J(\theta)^{-1}\Delta x$ to correct Δx and download the modified joint solution to the robot controller for motion execution.
c. Repeat from step (a) until Δx is sufficiently small.

The sensor based correction process is depicted in Figure 6-11.

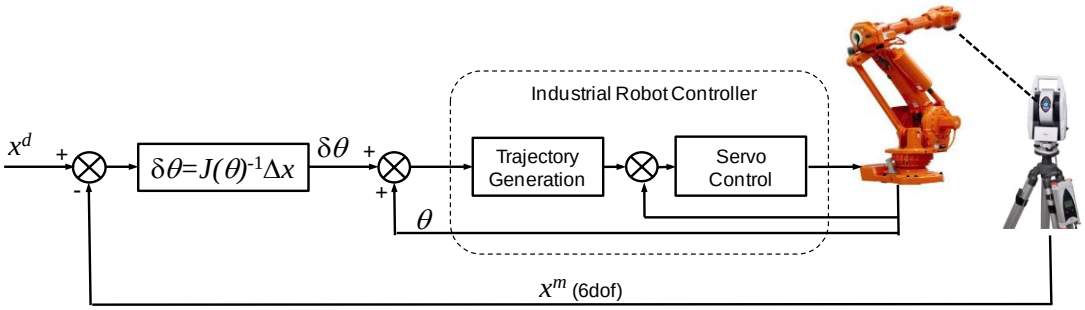


Figure 6-11 Error correction using a 6dof measuring device

If only 3dof positional measurements are provided, error vector $\Delta p = p^d - p^m$ between the programmed p^d and measured position p^m is calculated and downloaded to the robot controller to execute a relative movement along this vector from the robot's current position (Figure 6-12). The iterative correction process continues until Δp is sufficiently small. Since it is only possible to correct the position components of the end-effector, this simple method is only feasible as long as errors in orientation have been compensated by the model in the first step to acceptable tolerance. Notice in both cases, measurements from the global metrology system are expressed in the robot base frame, thanks to the $BASE$ and $PROBE$ transformations already identified in the calibration process (section 6.2.2).

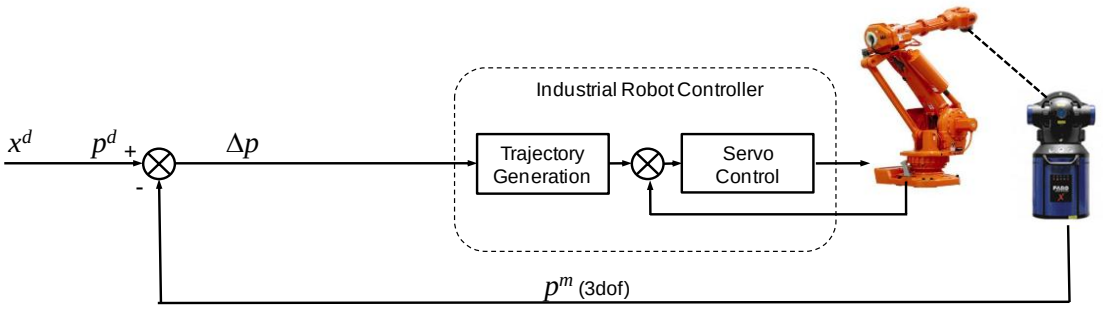


Figure 6-12 Error correction using 3dof measuring device

The thresholds that terminate the iterative sensor based correction process were determined experimentally based on the resolution of the robot used. In this work, the threshold for the position components $\|\Delta p\|$ and orientation components $\|\Delta \phi\|$ of the representative Smart H4 robot were chosen as 0.08mm and 0.05°, where $\|.\|$ denotes the 2-norm of the corresponding vector.

7 EXPERIMENTAL SETUP

This chapter describes the experimental setup developed to demonstrate the application of the proposed framework for the purpose of robot calibration and error compensation.

7.1 Overview

A distributed, service-based control system for the laboratory facilities used in this work is developed using the design template presented in Chapter 5. The facilities include a Comau Smart H4 robot, a Leica AT901-MR laser tracker that measures the position of the robot via the TMAC reflector (Figure 7-1) and Matlab computing software used to implement the calibration and error compensation models presented in Chapter 6.

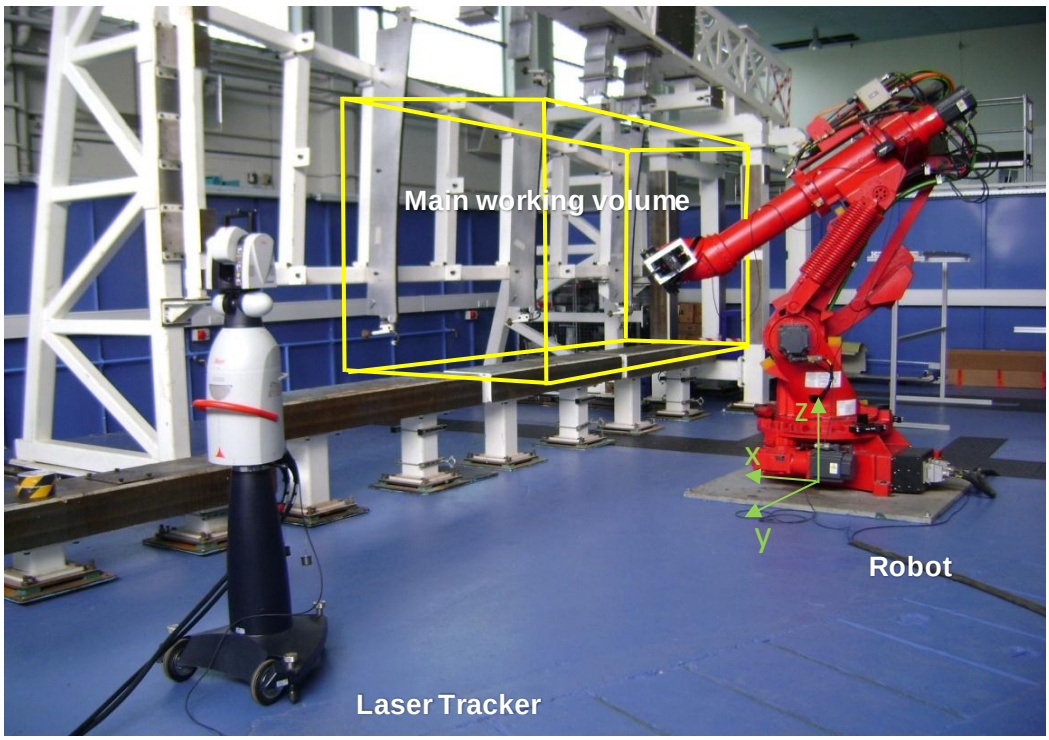


Figure 7-1 The hardware setup

The network architecture of the hardware and software components is depicted in Figure 7-2. The software components (services) include:

- The Smart H4 robot service, which communicates with the C3G controller of the robot via RS-232 serial interface;
- Two laser tracker services, one for automatic control and another for visualization of the measurements of the AT901-MR laser tracker. They communicate with the laser tracker's AT controller via the common Ethernet (TCP/IP);
- Matlab computing service, which interfaces with the Matlab (version 7.4) software via Windows's DDE (Dynamic Data Exchange) technology.
- Cell controller service, which performs overall management of other services.

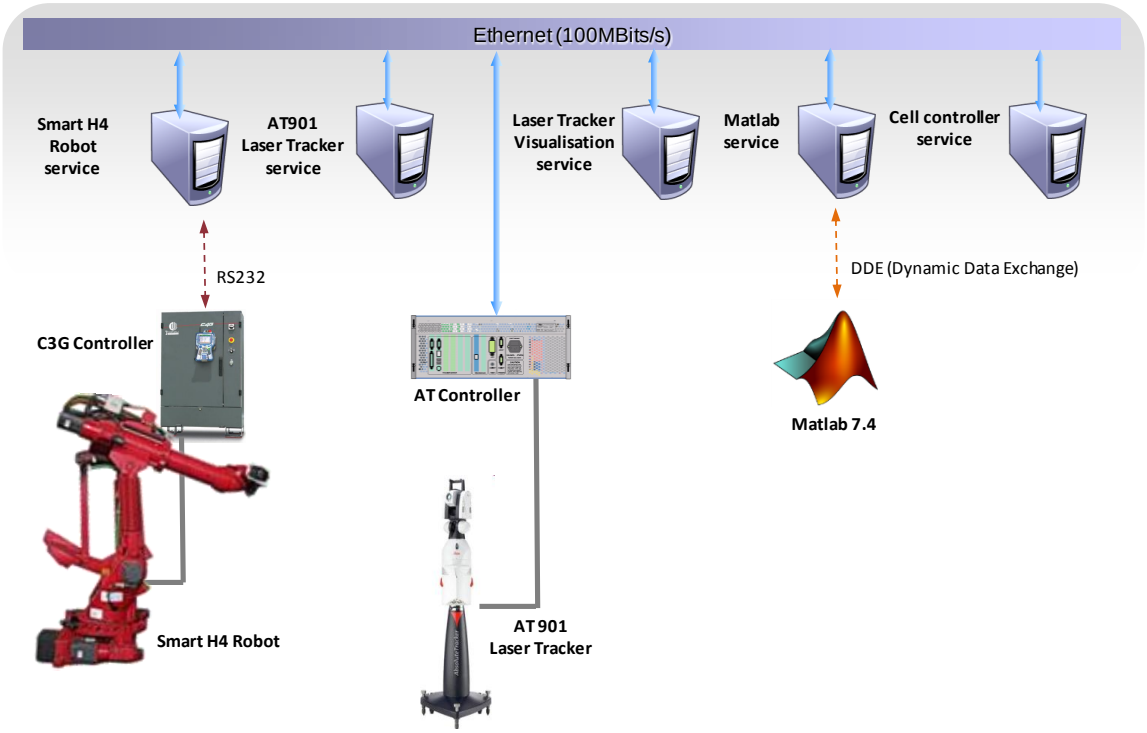


Figure 7-2 Network architecture

All the services are programmed in Visual C#, except the laser tracker visualization service which is programmed in Visual C++. The interconnections between these PnP services are established at run-time: the cell controller service subscribes to all other services and assists the robot service to subscribe to the laser tracker and Matlab services (Figure 7-3). After the connections has been set up, the cell controller dispatches robot programs to

the robot services and monitors all activities in the work-cell during the execution of these programs whilst the robot service is able to actively control its associated resources. The following sections will describe the developed services and their operations in more details.

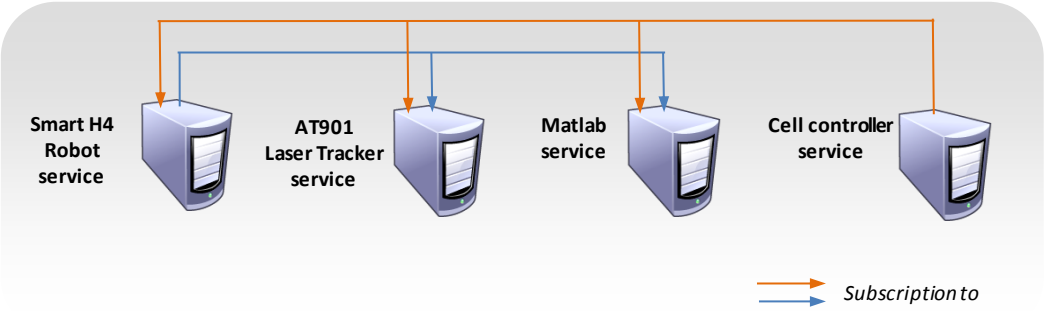


Figure 7-3 The software (service) setup.

7.2 Robot service

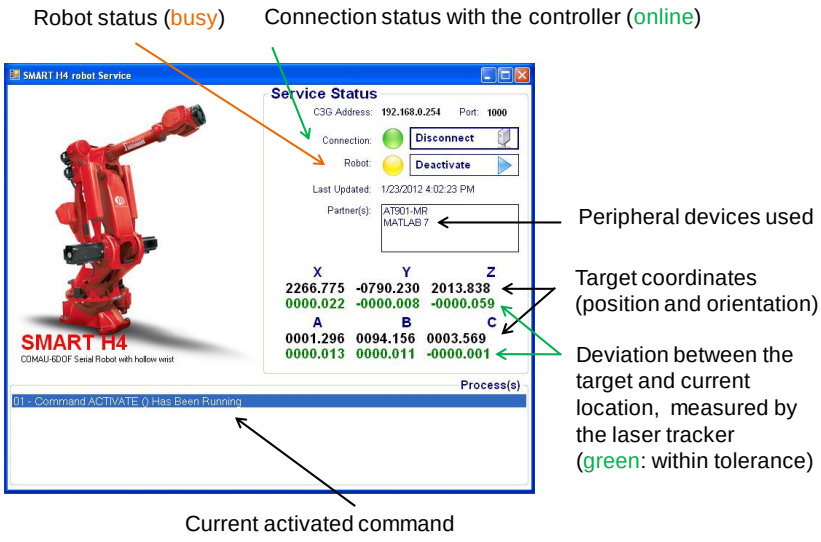


Figure 7-4 The robot service

The main responsibility of the Smart H4 robot service is activating robot programs dispatched from the cell controller in the C3G controller. When these programs are being executed, they might send control commands to other peripheral devices; in such cases, the robot service will setup the communication channels between them. Since the robot service is automatically controlled from the robot program, its Graphical User Interface (GUI) is

designed in a simple way to avoid unnecessary complication to the user (Figure 7-4). Control commands provided by the service are listed in Table 7-1.

Table 7-1 Control commands of the robot service

Command	Input Data	Output Data	Type	Comment
ACTIVATE	name: String	n/a	NoData	Run a robot program
DEACTIVATE	name: String	n/a	NoData	Stop a robot program
ARM_POS	n/a	pos: Double [6]	SingleData	Get current robot position
ARM_JNTP	n/a	joint: Double [6]	SingleData	Get current joint angles
JNTP_TO_POS	joint: Double [6]	pos: Double [6]	SingleData	Nominal forward kinematics
POS_TO_JNTP	pos: Double [6]	joint: Double [6]	SingleData	Nominal inverse kinematics

7.3 Matlab service

The Matlab service (Figure 7-5) serves as the interface for Matlab software in order that the robot service can invoke Matlab commands at run-time. The service's control commands are listed in Table 7-2.

Figure 7-5 The Matlab service

The command *Set Matrix* is for setting a matrix given the name *matrix_name* and the data *matrix* into the Matlab workspace. These parameters are provided in the fields *Option* and *InputData* of the class *Command*, respectively. Likewise, the command *Get Matrix* is for retrieving data from a matrix named

matrix_name from Matlab workspace. The command *Evaluate* is for evaluating an *expression*, which could be a Matlab internal command or a user-defined function written in the language. By using these three basic commands, the robot can delegate complex numerical computations to Matlab and retrieve the final result. For example, the error identification and compensation algorithms presented in sections 6.4, 6.5.1 are programmed in separate Matlab functions that will be called by the robot to perform the relevant processes.

Table 7-2 Control commands of the Matlab service

Command	Option	Input Data	Output Data	Type
Set Matrix	matrix_name	matrix: Double [m,n]	n/a	NoData
Get Matrix	matrix_name	n/a	matrix: Double [m,n]	SingleData
Evaluate	n/a	expression: String	n/a	NoData

7.4 Laser tracker service

The laser tracker service (Figure 7-6) is the control application for Leica laser trackers. The service's control commands are listed in Table 7-3.

Figure 7-6 The laser tracker service

The control commands are designed in such a way that multiple robots (if available) are able to control the laser tracker and make use of its measurements directly. Recall equation (6.23) that the relationship between the laser tracker's measurement T and the position T_n^0 of a robot to be measured is:

$$T = BASE \cdot T_n^0 \cdot PROBE \tag{7.1}$$

where $BASE$, $PROBE$ are the transformations defining the robot base frame F_0 w.r.t. the laser tracker frame and the TMAC frame w.r.t. the robot flange frame F_n, respectively (see Figure 6.5). Therefore, in order that the measurement can be used by the robot (for positioning), it must be transformed into the right frame as:

$$T_n^0 = BASE^{-1} \cdot T \cdot PROBE^{-1} \tag{7.2}$$

In addition, the result of equation (7.2) in matrix form must be converted into the data types that the robot can understand. They are usually in the form of a 6-vector of position and orientation components in which the latter can either be represented as Euler, Roll-Pitch-Yaw or projective angles (Spong *et.al.*, 2004), depending on the robot brand. Comau robots use Euler (Z-Y-Z) angles to represent the orientation.

Table 7-3 Control commands of the laser tracker service

Command	Option	Input Data	Output Data	Type
Set Base	rpy2tr/euler2tr/prj2tr	base: Double [6]	n/a	NoData
Set Probe	rpy2tr/euler2tr/prj2tr	probe: Double [6]	n/a	NoData
Move	n/a	point: Double [6]	n/a	NoData
Single Measurement	tr2rpy/tr2euler/tr2prj	duration: Integer	data: Double [6]	SingleData
Continuous Measurement	tr2rpy/tr2euler/tr2prj	interval: Integer	data: Double [6]	MultipleData
Stop Measurement	n/a	n/a	n/a	StopData

The laser tracker service will perform the data transformation and conversion mentioned above for each measurement automatically. Once the $BASE$ and $PROBE$ have been determined via the robot calibration process, the robot can assign these two transformations in the laser tracker service by using the commands *Set Base* and *Set Probe*, respectively. When calling the measuring commands (*Single Measurement* and *Continuous Measurement*), the robot also provides them with the optional data types that the measurements are

converted into (Figure 7-7). The command *Continuous Measurement* is of type *MultipleData*, thereby, the robot must send a command *Stop Measurement* to terminate the process. Multiple robots sharing the laser tracker must call the command *Move* provided with their instant positions before any measurement can take place. When receiving this command, the laser tracker calculates equation (7.1) to find out the position of the TMAC that the laser beam will be pointed toward then uses the ADM (section 2.3.1.1) to re-establish the connection.

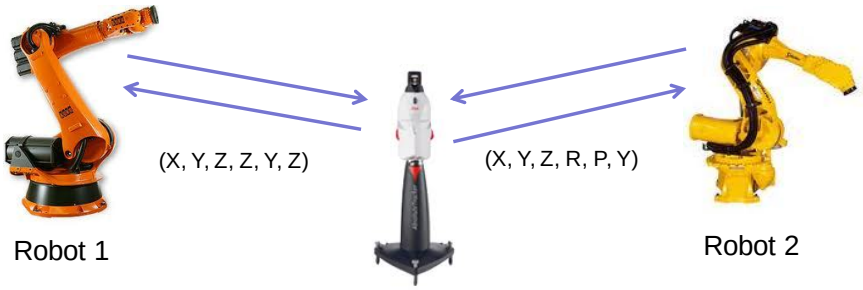


Figure 7-7 Measurements of the laser tracker are transformed and converted to proper robot data types.

7.5 Laser tracker visualization service

The laser tracker visualization service was originally developed as a stand-alone control application for the laser tracker and has been used by laboratory members in other research projects. In this work, it is mainly used for the visualization of the laser tracker's measurements using OpenGL engine (Figure 7-8). The software also contains built-in functions for calculating best-fit geometries (e.g. lines, circles, planes…) of the measurements, based on which initial estimates of the $BASE$ and $PROBE$ transformations between the tracker and the robot can be determined and input to the calibration process. Details on the procedures developed in this work to determine these two transformations using the software are described in Appendix D.

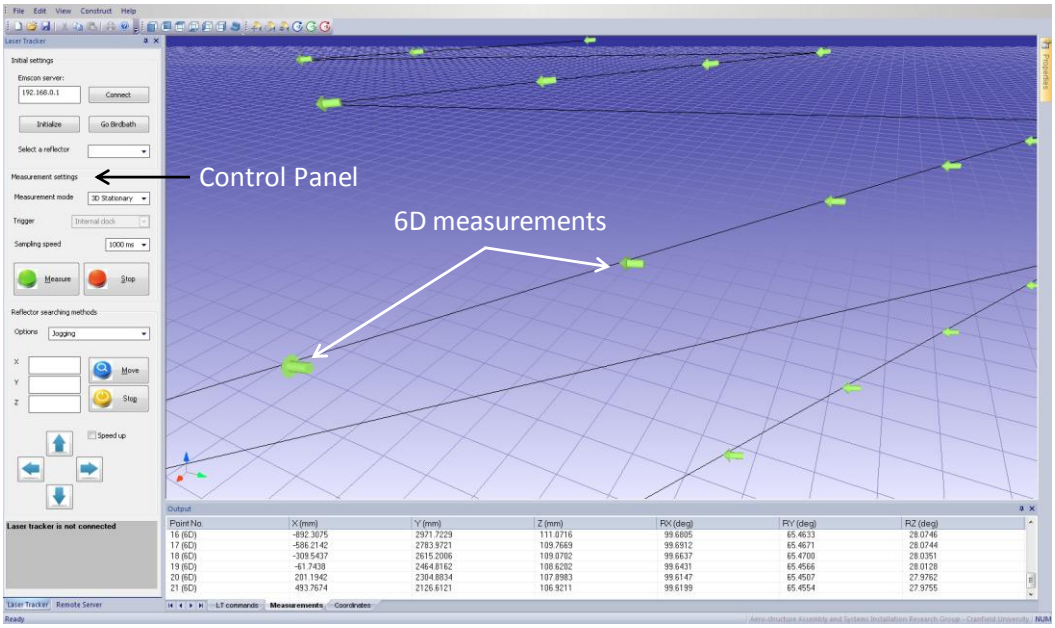


Figure 7-8 The laser tracker visualization service

7.6 Cell controller service

The cell controller service is responsible for the overall management of services in the work-cell. It allows the system operator to:

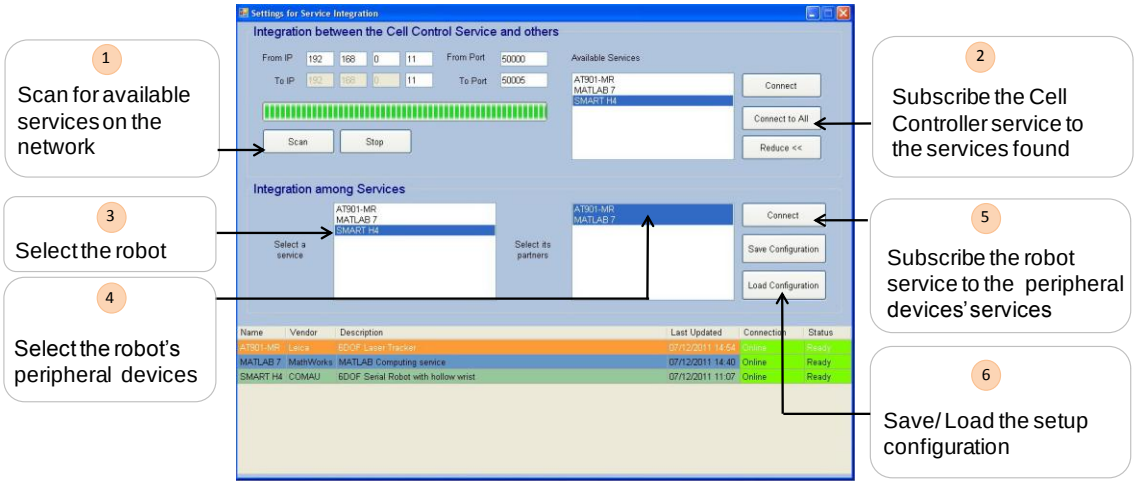


Figure 7-9 Discovery and setting up the integration of services

- Discover the availability of services and assign the interconnection between them in a PnP manner. To establish the connection configuration given in Figure 7-3, the operator firstly scans for the services available on the network then subscribes the cell controller to

these services. By doing this, the cell controller is able to monitor all the activities of the services in the work-cell. Next, the operator subscribes the robot service to those of the peripheral devices that it will use for the current manufacturing process (i.e., the laser tracker and the Matlab services). The result of this service integration can be saved and loaded for the next run (Figure 7-9). As can be seen, the integration of services can be performed in a simple way without having to follow the procedure previously described in section 5.1.1.3 (or in order words, re-programming of services).

- Assign the tasks to the services. Figure 7-10 demonstrates the procedure. The operator firstly selects a service in the list of available services on the top left panel. All the commands provided by the selected service to control its device will be displayed on the bottom left panel. Selected commands will be displayed on the right panel, from which the operator provides necessary optional parameters and input data before activating them. Although it is possible to create the tasks to all of the services to which the cell controller has subscribed, the main purpose is assigning and activating a robot program in the robot service. If the work cell contains several robots, the operator can also specify the synchronization between them: whether they will run sequentially or concurrently to each other.

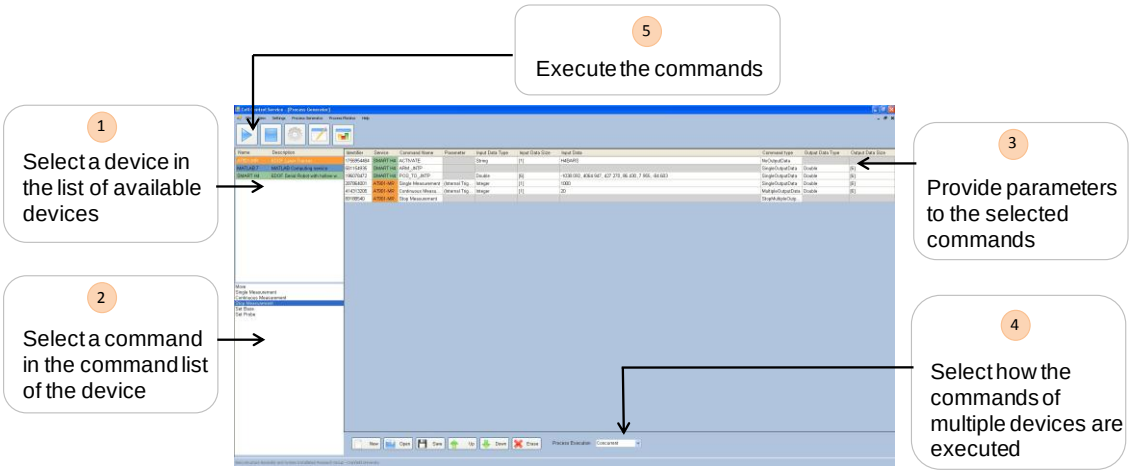


Figure 7-10 Assigning the tasks to services

- Monitor activities of the services. When the robot execute the given robot program, it will generate and send multiple processes to its peripheral devices. The cell controller monitors and visualizes the evolution of the generated processes, including their statuses and data (Figure 7-11).

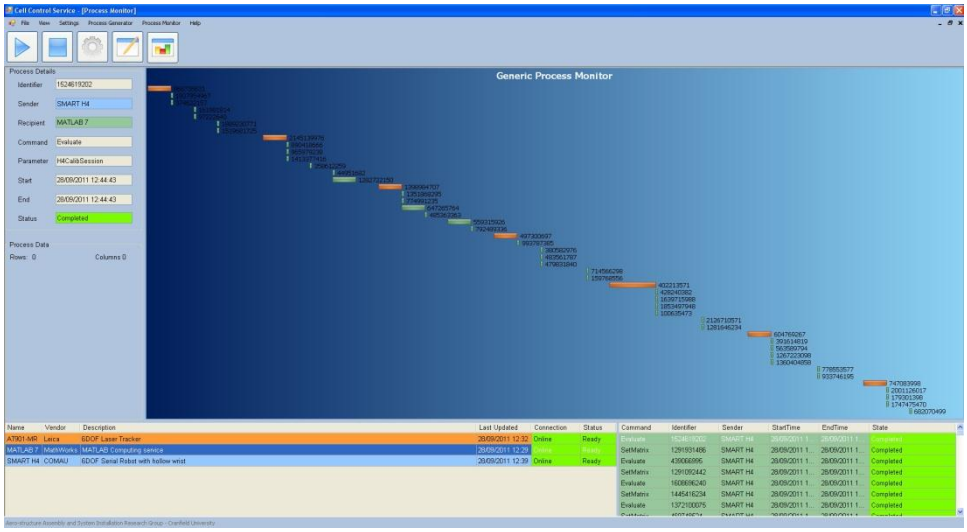


Figure 7-11 Monitoring service activities at run-time

The distributed control system is depicted in Figure 7-12. The services run on four Intel Core 2 PCs and Windows XP OS. To activate a service, the operator only needs to double click the icon on the desktop screen then uses the cell controller to wire it up with the others as presented in this section. When the service is closed, it sends the *Shutdown* notifications to the subscribers to delete its corresponding instance in these services as presented in section 5.2.3.6.

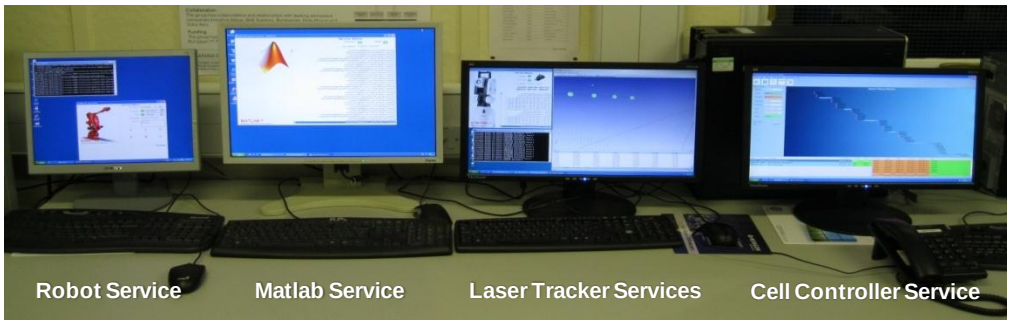


Figure 7-12 The system of developed services

8 SIMULATION, EXPERIMENT RESULTS AND ANALYSIS

This chapter presents the simulation and experiments undertaken in order to:

- Validate the work on error modelling and identification proposed in Chapter 6 for serial manipulators having a parallelogram linkage,
- Demonstrate the use of the distributed control system in Chapter 7 developed by using the design template given in Chapter 5 for the automation of the calibration and error compensation processes.

8.1 Simulation

A simulation was conducted to validate the developed error model in Chapter 6 and prove its novelty when compared with existing methods introduced in Chapter 4. To begin with, simulated dimensional and angular errors of in the range of $\pm 4\times10^{-3}$ m and $\pm 4\times10^{-3}$ radians were added to the nominal parameters of the Smart H4 robot given in Table 6-1. In this case, the parallelogram mechanism degenerated into a four bar linkage (since $a_2 \neq a_{3'}, a_{2'} \neq a_{4'}$) and thus, passive joint angles $\theta_{3'}, \theta_{4'}$ were computed from the simulated $\theta_2, \theta_{2'}$ and link lengths $a_2, a_{2'}, a_{3'}, a_{4'}$ following the position equations of a four bar linkage provided in Appendix C. As a result of the artificial errors, “actual” tool positions deviated from the ones computed by the nominal kinematic model about 11mm. If the proposed error model in section 6.2.1 is correct, identified error parameters will be identical with the simulated and tool pose errors will be significantly reduced. The calibration result of this model, denoted as model 1, will be compared with result of a competitive model, namely model 2, implemented based on the method given in (Marie *et.al.*, 2008). In this model, the equations (C.2) were differentiated directly to obtain an error model of the parallelogram structure somewhat similar to equation (6.20). Twenty eight simulated geometric error parameters were solved by the identification algorithm presented in section 6.4, except for non-geometric errors, which are not considered in this simulation. The threshold ε that terminates this Gauss-Newton identification process is selected as 10^{-6}. The condition number of the

regression matrices of the two identification systems, defined in equation (4.11), is 92.04<100 indicating both of them are well-conditioned. The simulations were carried out with different set of simulated data.

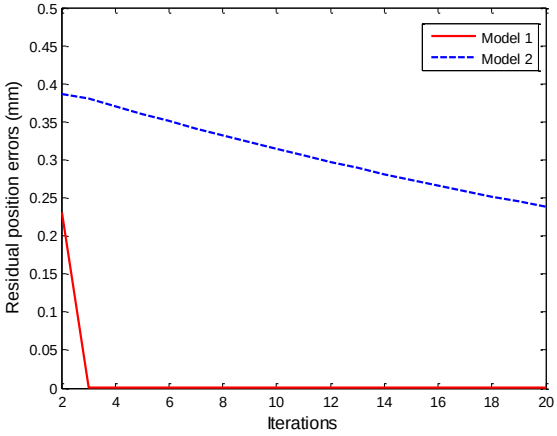


Figure 8-1 Accuracy of the proposed calibration model (Model 1) and a competitive model (Model 2) from (Marie *et.al.*, 2008) after the 2nd iteration

Figure 8-1 shows the results of the calibration models versus the iterations produced from one set of simulated data. Model 1 converged quickly after 4 iterations and the mean residual error is 3.4×10^{-5} mm. Model 2 converged much slower, taking more than 450 iterations to obtain a comparable result. This can be explained as the linearization error of model 1 is mild since it is only due to *sine* and *cosine* functions in equations (6.17-18) while that of model 2 is much more severe. In addition to the speed of convergence, model 1 offers the advantage of simplicity whereas the derivation of model 2 is a tedious work, even with the aid of symbolic Matlab programming. Apart from these, error parameters identified by model 1 and model 2 (after 450 iterations) are almost identical, as depicted in Table 8-1.

It can be seen from the table that dimensional and angular parameters of the parallelogram linkage (links 2'3'4'3) were identified with an accuracy better than 10^{-5} m and 10^{-5} rad while those of the open chain were identified with higher accuracy (10^{-7} m and 10^{-7} rad, respectively). The simulation results have proved that the proposed model is accurate and computationally efficient.

Table 8-1 Identified errors parameters from the simulation

Link	Δ	Input	Model 1 (after 7 iterations)		Model 2 (after 450 iterations)		Note about identified values
			Identified	Error	Identified	Error	
1	$\Delta\theta_1$	-1	-0.9999	$5.72e^{-6}$	-0.9999	$3.90e^{-6}$	
	Δd_1	-4	-4.0000	$-2.67e^{-5}$	-4.0000	$1.64e^{-5}$	
	Δa_1	-1	-1.0000	$-1.04e^{-5}$	-0.9999	$1.58e^{-5}$	
	$\Delta\alpha_1$	4	4.0000	$-2.33e^{-6}$	4.0000	$9.26e^{-6}$	
2	$\Delta\theta_2$	3	2.9905	$-9.54e^{-3}$	2.9905	$-9.54e^{-3}$	
	Δa_2	-1	-1.0000	$-1.61e^{-5}$	-1.0001	$-8.50e^{-4}$	
2’	$\Delta\theta_{2'}$	-2	-1.9992	$8.38e^{-4}$	-1.9992	$8.43e^{-4}$	$\Delta\theta_{2'}+\Delta\theta_3$
	$\Delta d_{2'}$	-3	Dependent parameter, identified via Δd_3				
	$\Delta a_{2'}$	-2	2.0089	$8.91e^{-3}$	2.0089	8.91^{-3}	$\Delta a_{2'}-\Delta a_{4'}$
	$\Delta\alpha_{2'}$	3	2.9962	$-3.77e^{-3}$	2.9962	$-3.77e^{-3}$	
	$\Delta\beta_{2'}$	0	$-4.6e^{-6}$	0	$-1.61e^{-6}$	0	
3’	$\Delta\theta_{3'}$	Position dependent error					
	$\Delta d_{3'}$	-2	Dependent parameter, identified via Δd_3				
	$\Delta a_{3'}$	-3	-3.0089	$-8.88e^{-3}$	-3.0089	$-8.98e^{-3}$	
	$\Delta\alpha_{3'}$	2	2.0000	$-2.05e^{-5}$	2.0000	$-1.96e^{-5}$	
	$\Delta\beta_{3'}$	0	$-2.16e^{-5}$	0	$-1.48e^{-5}$	0	
4’	$\Delta\theta_{4'}$	Position dependent error					
	$\Delta d_{4'}$	-1	Dependent parameter, identified via Δd_3				
	$\Delta a_{4'}$	-4	Dependent parameter, identified via $\Delta a_{2'}$				
	$\Delta\alpha_{4'}$	1	Dependent parameter, identified via $\Delta\alpha_3$ and $\Delta\theta_4$				
	$\Delta\beta_{4'}$	0	Dependent parameter, identified via $\Delta\alpha_3$ and $\Delta\theta_4$				
3	$\Delta\theta_3$	0	Dependent parameter, identified via $\Delta\theta_{2'}$				
	Δd_3	4	-2.0024	$-0.24e^{-3}$	-2.0024	$-0.24e^{-3}$	$\Delta d_{2'}+\Delta d_{3'}+\Delta d_{4'}+\Delta d_3$
	Δa_3	1	0.9949	$-5.05e^{-3}$	0.9949	$-5.00e^{-3}$	
	$\Delta\alpha_3$	-1	-0.8785	$-4.17e^{-4}$	-0.8785	$-4.13e^{-4}$	$\Delta\alpha_3+\Delta\alpha_{4'}\cos(7°)$
4	$\Delta\theta_4$	3	3.9888	$-3.73e^{-3}$	3.9888	$-3.74e^{-3}$	$\Delta\theta_4+\Delta\alpha_{4'}\sin(7°)$
	Δd_4	3	3.0004	$3.66e^{-4}$	3.0004	$3.52e^{-4}$	
	Δa_4	2	2.0000	$1.31e^{-5}$	2.0000	$4.67e^{-6}$	
	$\Delta\alpha_4$	-2	-2.0000	$-1.01e^{-6}$	-2.0000	$-9.05e^{-6}$	
5	$\Delta\theta_5$	2	2.0000	$2.57e^{-6}$	2.0000	$-8.22e^{-7}$	
	Δd_5	2	2.0000	$-3.06e^{-6}$	2.0000	$1.29e^{-6}$	
	Δa_5	3	3.0000	$-9.90e^{-6}$	3.0000	$-9.04e^{-6}$	
	$\Delta\alpha_5$	-3	-3.0000	$1.10e^{-6}$	-3.0000	$7.94e^{-6}$	
6	$\Delta\theta_6$	1	1.0000	$-9.67e^{-7}$	1.0000	$1.10e^{-6}$	
	Δd_6	1	1.0000	$4.53e^{-6}$	0.9999	$-5.00e^{-6}$	
	Δa_6	4	4.0000	$-2.31e^{-6}$	4.0000	$-8.88e^{-6}$	
	$\Delta\alpha_6$	-4	-4.0000	$-1.63e^{-6}$	-4.0000	$3.15e^{-6}$	

(Units: length: 10^{-3}m, angles: 10^{-3} rad)

8.2 Calibration

8.2.1 Experiments

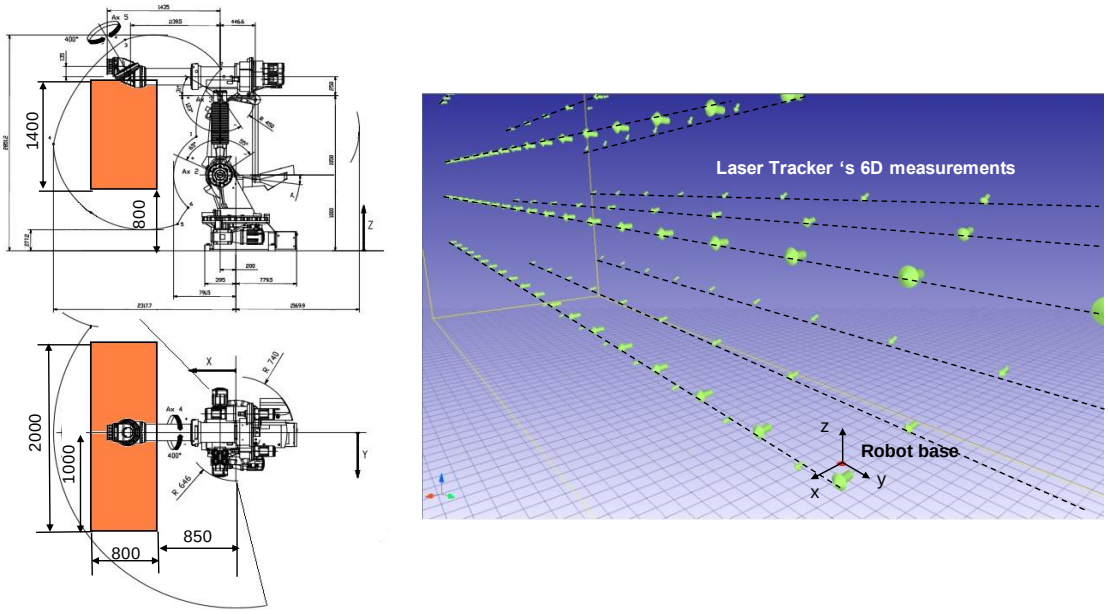


Figure 8-2 Calibration is performed in the main working volume of the robot. From left to right: a. Location of the volume in the robot workspace; b. Visualization of the laser tracker's measurements in the volume.

Calibration experiments on the real Smart H4 robot were performed to further verify the proposed calibration model. The robot is calibrated within its designated working volume, which is a rectangular box (L: 2000×W: 800×H: 1400) dominating its workspace. Actual positions of the robot when moving in this volume are measured by the AT901-MR laser tracker having accuracy better than 5×10^{-2}mm in position and 10^{-2} degree in orientation via the TMAC reflector (Figure 8-2). Nominal positions of the robot are calculated from joint angles and will be pre-multiplied and post-multiplied with the $BASE$ and $PROBE$ transformations to form the nominal 6D position of the TMAC in the laser tracker base (equation (6.23) in Chapter 6). Initial (nominal) estimates of the $BASE$ and $PROBE$ transformations are determined following the procedures given in Appendix D. Deviations between these nominal TMAC positions and their measurements, calculated in equation (B.15), are used to

identify intrinsic error parameters of the robot structure as well as those of the $BASE$ and $PROBE$ above following the algorithm given in section 6.4.

The robot was tested in unloaded and loaded with 18kg (40lbs) and 36kg (80lbs) cases (Figure 8-3). The deadweight simulates the mass of an end-effector in practice and is mounted at an offset distance to the centre of the flange surface in order that it will create moment to all the joints (except joint 1).

Figure 8-3 Applied loading at the robot TCP. From left to right: a. No loading; b. 18kg (40lbs) loading; c. 36kg (80lbs) loading.

The accuracy measures of the robot before and after calibration are defined as the mean of the deviations in position and orientation between the laser tracker measurements and the kinematic models' estimates. The orientation components in this section are represented in Roll-Pitch-Yaw angles. Denote the 6D laser tracker measurement of the TMAC $x_M = (p_{Mx}, p_{My}, p_{Mz}, \phi_{Mx}, \phi_{My}, \phi_{Mz})$ and the corresponding estimate provided by a kinematic model $x = (p_x, p_y, p_z, \phi_x, \phi_y, \phi_z)$ then the means of the errors for an entire set of *m* data points are:

$$\Delta\bar{p} = \frac{\sum_{i=1}^{m} \Delta p_i}{m} = \frac{\sum_{i=1}^{m} \sqrt{(p_{Mx} - p_x)_i^2 + (p_{My} - p_y)_i^2 + (p_{Mz} - p_z)_i^2}}{m}$$
$$\Delta\bar{\phi} = \frac{\sum_{i=1}^{m} \Delta \phi_i}{m} = \frac{\sum_{i=1}^{m} \sqrt{(\phi_{Mx} - \phi_x)_i^2 + (\phi_{My} - \phi_y)_i^2 + (\phi_{Mz} - \phi_z)_i^2}}{m} \quad (8.1)$$

and the standard deviations of these parameters are:

$$\sigma_{\Delta p} = \sqrt{\frac{\sum_{i=1}^{m}(\Delta p_i - \Delta p)^2}{m-1}} \qquad (8.2)$$

$$\sigma_{\Delta \phi} = \sqrt{\frac{\sum_{i=1}^{m}(\Delta \phi_i - \Delta \phi)^2}{m-1}}$$

The same calculations are also applied for individual components $\Delta\overline{p_x}, \Delta\overline{p_y}, \Delta\overline{p_z}$ of $\Delta\overline{p}$, and $\Delta\overline{\phi_x}, \Delta\overline{\phi_y}, \Delta\overline{\phi_z}$ of $\Delta\overline{\phi}$.

The standard deviations of the identified parameters Δg in the identification system (4.7) of Chapter 4 are determined as in (Lange, 1999; Montgomery *et.al.*, 2003). Firstly, let σ^2 be the variance of the residual error $\Delta\mathbf{x}$ of the model (4.7):

$$\sigma^2 = \frac{\sum_{i=1}^{n}\Delta x_i^2}{n-r} \qquad (8.3)$$

where n is the total number of observations in m TMAC measurements, n=6m since one TMAC measurement provides 6 observations; r is the number of identified parameters in Δg; n-r is called *the statistical degrees of freedom*. The variances of the identified parameters in Δg are calculated as:

$$\sigma_{\Delta g}{}^2 = \sigma^2(\mathbf{H}(\mathrm{g})^T\mathbf{H}(\mathrm{g}))^{-1} = \sigma^2\mathbf{C} \qquad (8.4)$$

where $\mathbf{C}$ is a ($r \times r$) symmetric matrix and the standard deviation of a parameter Δg_j in Δg therefore is:

$$\sigma_{\Delta g\, j} = \sigma\sqrt{\mathbf{C_{jj}}} \qquad (8.5)$$

8.2.2 Implementation

Using the control system presented in Chapter 7, the calibration process was automated as illustrated in Figure 8-4. A robot program was developed in Comau PDL language to generate the desired robot configurations and send commands to the laser tracker and Matlab software. When the robot has moved

to a position, it requests the laser tracker to take single measurement in one second then adds the result to a matrix named mT in Matlab. It also records the joint position reading corresponding to this instant position (by using the internal PDL statement *ARM_JNTP*) then adds it to a Matlab matrix named mJ. The process is then repeated for all the points in the volume. At the end of the program, the robot activates a developed Matlab function named "*H4Calibration*". The function accepts the mJ and mT as its inputs, performs the identification for error parameters then displays the calibration results to the user and saves the identified parameters onto hard disk.

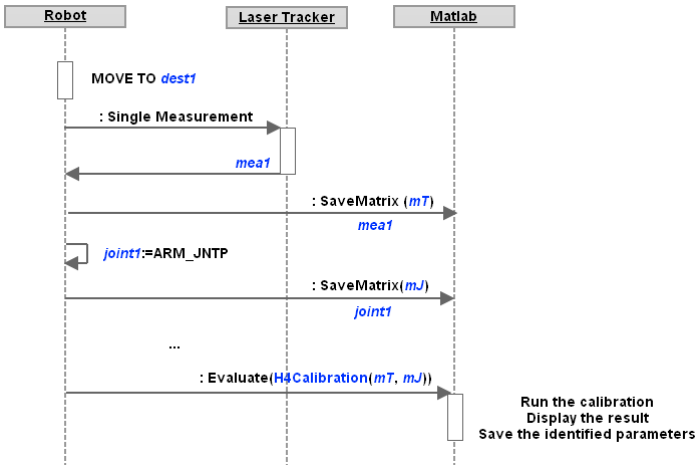


Figure 8-4 Implementation of the calibration process

8.2.3 Results and analysis

To verify whether the proposed error model is accurate, the calibration was performed using several models. When the robot was unloaded, four models were used to identify parameter errors from a set of 90 measurements. In the first model, the robot is regarded as a simple serial-link manipulator (by computing equation (6.1) with (6.3), thus neglecting the parallelogram structure). This "standardized" model uses the 6×4+6=30 parameters (Schröer *et.al.*, 1997). The second is the proposed model using 34 geometric parameters described in sections 6.2.1, 6.2.2. The third and fourth models are extended from the first and second with the inclusion of the compliance parameters G_1,

G_2, G_3 presented in section 6.3.1. The calibration results are summarized in Table 8-2.

Table 8-2 Residual errors in position and orientation of the models

Models	Errors in position (10^{-3}m)		Errors in orientation (°)					
	$\overline{\Delta p}$	$\sigma_{\Delta p}$	$\overline{\Delta\phi_x}$	$\sigma_{\Delta\phi_x}$	$\overline{\Delta\phi_y}$	$\sigma_{\Delta\phi_y}$	$\overline{\Delta\phi_z}$	$\sigma_{\Delta\phi_z}$
Nominal	4.03	0.66	0.380	1.283	0.079	0.571	0.530	0.917
Model 1	1.14	0.60	0.015	0.013	0.026	0.020	0.021	0.016
Model 2	1.08	0.38	0.015	0.011	0.020	0.014	0.020	0.015
Model 3	0.78	0.35	0.015	0.010	0.020	0.010	0.020	0.010
Model 4	0.40	0.21	0.008	0.007	0.008	0.005	0.010	0.007

It can be seen from Table 8-2 that all the models were able to predict the orientation components accurately. This can be explained as parameters of the last three links which affect the TCP orientation are identical in the four models. On the other hand, differences between the models: the parallelogram and compliance parameters of joints 2, 2' mostly affect the positional accuracy of the TCP. The mean values of residual errors in position initially were 4.03mm (before calibration) then reduced to 1.14mm, 1.08mm, 0.78mm and 0.40mm with the four models. There was only a slight improvement in the result of the proposed model 2 over the model 1's because they were both deteriorated by the deflections caused by link weights (Figure 8-5a). When this non-geometric error was eliminated, the difference became much clearer (Figure 8-5b):

- Model 4 outperformed model 2 by more than 250%, proving that the work on modelling of joint deflections due to structural loading is accurate.
- Model 4 outperformed model 3 by almost 100% even though they utilized the same deflection error model. Since the joint deflections $\Delta\theta_2^s$, $\Delta\theta_{2'}^s$ which contribute to the accuracy improvements of these models over the formers are functions of joint angles θ_2 and $\theta_{2'}$ (equation (6.31)), it can be inferred that θ_2 and $\theta_{2'}$ were better estimated by model 4, as the result of the developed error model of the parallelogram linkage.

Therefore, the proposed error modelling for serial manipulators having the parallelogram structure does improve calibration accuracy.

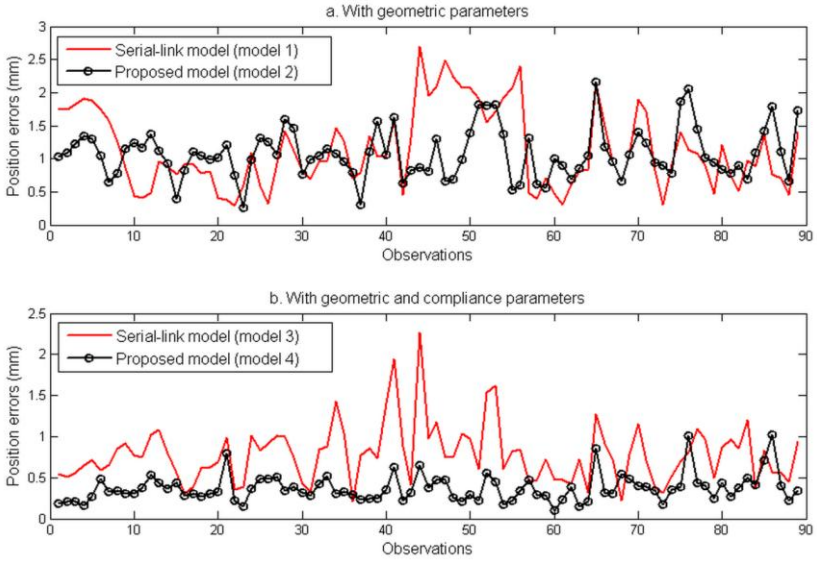


Figure 8-5 Experimental evaluation of the standardized serial link model and the proposed calibration model when the robot is unloaded

Next, the robot was tested in the 18kg and 36kg loaded cases. The deflections of the robot structure due to the payloads, measured as the deviations between the positions of the robot before and after being loaded, are shown in Figure 8-6. In order to verify the work on modelling of the deflections, the robot was firstly calibrated by using the model 4 above then by using the complete model, namely model 5, which includes 52 geometric and compliance parameters described earlier in section 6.4 of Chapter 6. Figure 8-7 displays the position errors of the robot before and after calibration using these models.

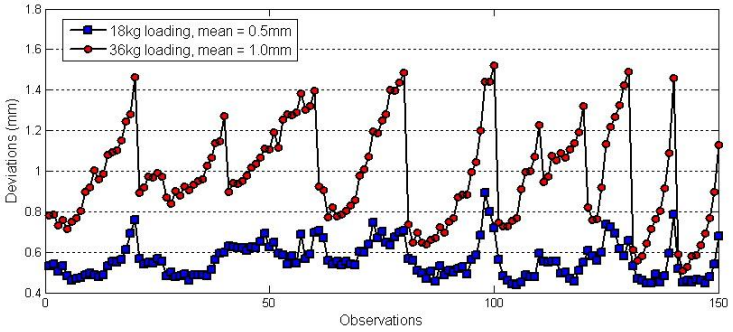


Figure 8-6 Elastic deflections caused by the deadweight, measured over 150 data points.

Table 8-3 Accuracy of the calibration models

Load case		Errors in position (10^{-3}m)		Errors in orientation (°)					
		$\overline{\Delta p}$	$\sigma_{\Delta p}$	$\overline{\Delta \phi_x}$	$\sigma_{\Delta \phi_x}$	$\overline{\Delta \phi_y}$	$\sigma_{\Delta \phi_y}$	$\overline{\Delta \phi_z}$	$\sigma_{\Delta \phi_z}$
18 kg	Nominal	3.95	0.90	-0.519	2.098	-0.046	0.067	-0.360	2.090
	Calibrated	0.50	0.29	-0.016	0.081	0.002	0.008	-0.016	0.083
36 kg	Nominal	4.08	1.00	-0.390	2.106	-0.055	0.066	-0.229	2.098
	Calibrated	0.57	0.33	-0.027	0.100	0.002	0.009	-0.027	0.100

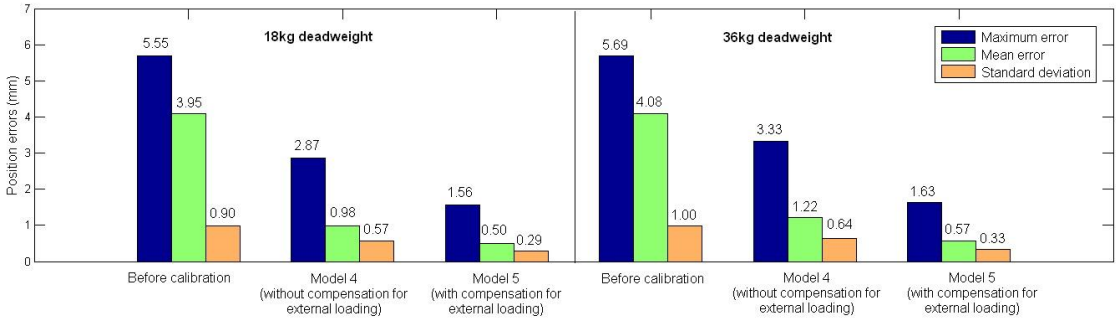


Figure 8-7 Experimental evaluations of the proposed calibration model without/with compensation for external loading.

It is interesting to see from Figure 8-6 and Figure 8-7 that despite the payloads having caused significant deflections (0.5mm and 1.0mm, respectively), the average deviations between nominal and "actual" robot positions before calibration remain almost unchanged at 4mm. This might be explained as the errors induced by inaccurate geometric parameters and the deflections are not always in the same directions: the latter might augment or suppress the former, depending on the configurations of the robot. Model 4, though it was able to reduce the initial errors by more than 300% (from 4.0mm to 1.22mm), lost its accuracy by more than 300%, compared with the unloaded case (from 0.4mm to 1.22mm). Model 5, having parameters modelling joint compliances induced by the payloads, produced better results: its average accuracy is 0.50mm for the 18kg and 0.57mm for the 36kg deadweight. Although there is a loss of nearly 0.1mm in the accuracy of this model for every applied 18kg, the results are considered acceptable, given that the deflections are 5 times higher (0.5mm). This 20% of uncompensated deflections might be owed to the link

compliances and deformations of joint bearings that are unaccounted for in the model (DeVlieg, 2010). The calibration results of model 5 for the two load cases are summarized in Table 8-3.

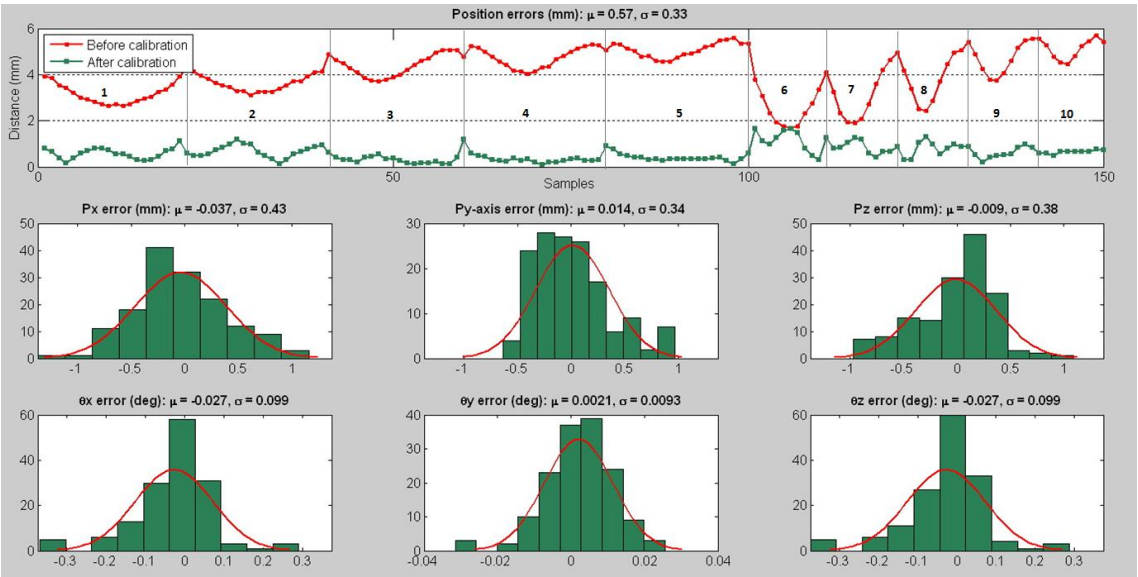


Figure 8-8 Output of the Matlab function *H4Calibration* at the end of the automated calibration process

From the experimental evaluations performed, it can be concluded that the calibration model (model 5) is sufficiently accurate [<1mm as stated in the research objective and comparable to respective research (Schröer *et.al.*, 1997)]. Figure 8-8 displays output to the user of the Matlab function *H4Calibration* that implements the model at the end of the automated calibration process. The result is of the calibration process of the 36kg loaded case, which takes about 15 minutes to collect the 150 laser tracker's measurements (in 10 line segments roughly parallel to the Y axis) and 5 seconds to complete the parameter identification in Matlab. It can be seen from the probability distributions shown in the figure that the calibration model was able to predict individual error components $\Delta\overline{p_x}, \Delta\overline{p_y}, \Delta\overline{p_z}$ of $\Delta\overline{p}$ and $\Delta\overline{\phi_x}, \Delta\overline{\phi_y}, \Delta\overline{\phi_z}$ of $\Delta\overline{\phi}$ within the accuracy of ±1.0mm and ±0.4°, respectively. The geometric and compliance error parameters of the robot identified by the model are summarized in Table 8-4.

Table 8-4 Identified parameters of the calibration model in the 36kg loaded case

a. Geometric parameters

Link	Δθ (°)		Δd (10^{-3}m)		Δa (10^{-3}m)		Δα (°)		Δβ (°)	
	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
1	0.04	0.08	-1.05	0.39	0.10	0.04	0.06	0.009		
2	0.02	0.06			1.48	0.10				
2’	-0.14	0.02			0.24	0.03	0.03	0.02	-0.02	0.01
3’					1.67	0.10	0.004	0.006	0.02	0.005
4’										
3			-2.73	1.00	0.66	0.21	0.08	0.01		
4	0.08	0.02	0.32	0.37	-1.74	0.58	-0.03	0.005		
5	-0.05	0.006	-0.37	0.12	1.86	0.63	0.02	0.005		
6	-0.25	0.08	0.06	0.51	0.15	0.49	-0.01	0.02		
7	0.01	0.006	-1.26	0.46						

Link	Δa (10^{-3}m)		Δb (10^{-3}m)		Δα (°)		Δβ (°)	
	μ	σ	μ	σ	μ	σ	μ	σ
0	-0.54	0.17	0.48	0.23	-0.03	0.01	0.025	0.005

b. Compliance parameters

Link	A_i (rad/m)		B_i (rad)		C_i (rad)		D_i (rad)	
	μ	σ	μ	σ	μ	σ	μ	σ
2	$-3.7.10^{-4}$	$6.1.10^{-4}$						
2’	$-6.8.10^{-3}$	$9.0.10^{-4}$	$4.6.10^{-4}$	$2.7.10^{-4}$	$-2.1.10^{-4}$	$2.4.10^{-4}$	$-1.4.10^{-3}$	$2.7.10^{-4}$
4	$-1.8.10^{-2}$	$1.0.10^{-3}$	$1.9.10^{-4}$	$2.5.10^{-4}$	$-2.2.10^{-3}$	$2.0.10^{-4}$	$-5.0.10^{-3}$	$2.7.10^{-4}$
5	$-8.0.10^{-3}$	$2.4.10^{-4}$	$1.7.10^{-4}$	$2.0.10^{-4}$	$-3.9.10^{-4}$	$1.9.10^{-4}$	$-2.5.10^{-3}$	$3.2.10^{-4}$
6			$1.6.10^{-4}$	$3.3.10^{-4}$	$-4.3.10^{-3}$	$1.0.10^{-3}$		

Link	G_1		G_2		G_3	
	μ	σ	μ	σ	μ	σ
2	$7.7.10^{-3}$	$1.1.10^{-3}$				
2’			$4.9.10^{-5}$	$1.5.10^{-3}$	$-3.6.10^{-3}$	$1.0.10^{-3}$

8.3 Error compensation

8.3.1 Experiments

Other experimental evaluations were performed to validate the two stage error compensation method proposed in this work. The robot was programmed to

travel along 6 straight lines parallel to the Y axis within its main working volume; each line contains 13 coordinate locations at an equal distance of 150mm (Figure 8-9). The robot was firstly positioned by the error compensation model whose parameters were determined from the calibration above and then was guided by the laser tracker to the target positions.

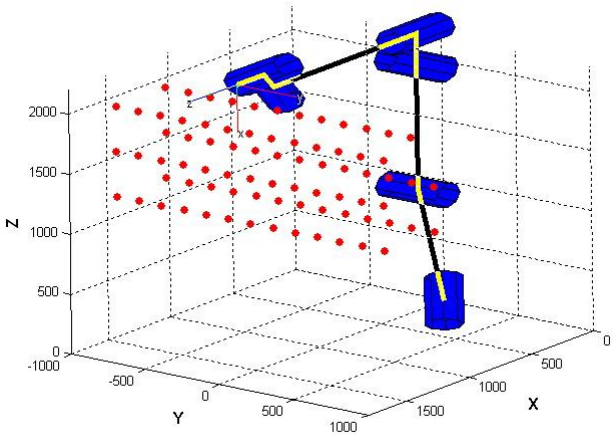


Figure 8-9 The test points for error compensation

The positions of the robot before and after each error compensation stage were measured by the laser tracker. These measurements are used to calculate the following accuracy measures, with reference to (ISO 9283, 1998) and (Young *et.al.*, 2000):

- Absolute accuracy of the robot, measured as the deviation between "actual" position x_{M} and the programmed position x following equations (8.1-2).
- The straightness of the Y axis, measured as the deviations in Z-axis and X-axis of the position x_{M} to the datum line that best fits through the 13 positions of the robot when travelling along the Y axis.

8.3.2 Implementations

Using the developed control system, the tests were automated by a developed PDL robot program as follows. At the start of the program, the robot activates a developed Matlab function named *LoadParams* which loads the identified

calibration parameters from the hard disk into Matlab workspace memory. These include the two *BASE* and *PROBE* transformations which will be set to the laser tracker service (part 7.4). This process is programmed in a PDL routine named *INITIALIZE* (Figure 8-10).

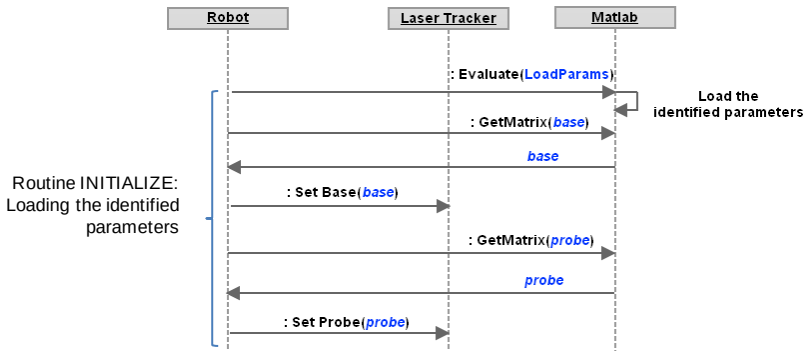


Figure 8-10 Loading identified calibration parameters into memory

For each target position, the robot firstly moves on its own nominal kinematic model then performs the two stage error compensation process as depicted in Figure 8-11.

- Model-based error compensation: The robot firstly calls the internal PDL statement *POS_TO_JNTP* to transform the target position from Cartesian to joint coordinates (the inverse kinematics). The joint angles are input to a developed Matlab function named "*H4Compensation*" which implements the error compensation algorithm in section 6.5.1. This function uses the identified parameters loaded by the Matlab function *LoadParams* above to calculate the joint solutions that compensate for the errors. The robot retrieves the modified joint solution calculated by the function then advances to the target position corresponding to these joint coordinates.
- Sensor-based error compensation: is implemented as a static ("move then measure") correction since the Smart H4 robot does not have a necessary low level interface to its C3G controller. The robot firstly requests the laser tracker to take multiple measurements, each of which is within 50ms. The measured current robot position and the target position are input to a developed PDL routine *CALC* that calculates the

corrective joint increment. Since it is not possible to calculate the inverse Jacobian matrix (section 6.5.2) in the PDL language, this routine actually delegates the work to the robot service and retrieves the result. The robot makes a differential move corresponding to the provided joint increment then calculates the residual errors in position and orientation. This process is repeated until the errors are smaller than 0.08mm and 0.05°, respectively. These tolerances are defined on the basis of the resolution of the robot, measured as around 0.05mm in translation and 0.02° in orientation by the laser tracker when the robot moves in infinitesimal joints increments (0.0005°).

The model-based and sensor-based error compensations are programmed into two separate PDL routines, *MODEL_COM* and *SENSOR_COM*. Other robot programs can reuse these shared routines as will be shown later in section 8.4. If another sensor is used instead of the laser tracker, the technician only has to modify the relevant commands in the *SENSOR_COM* routine whereas computer programs (services) remain unchanged.

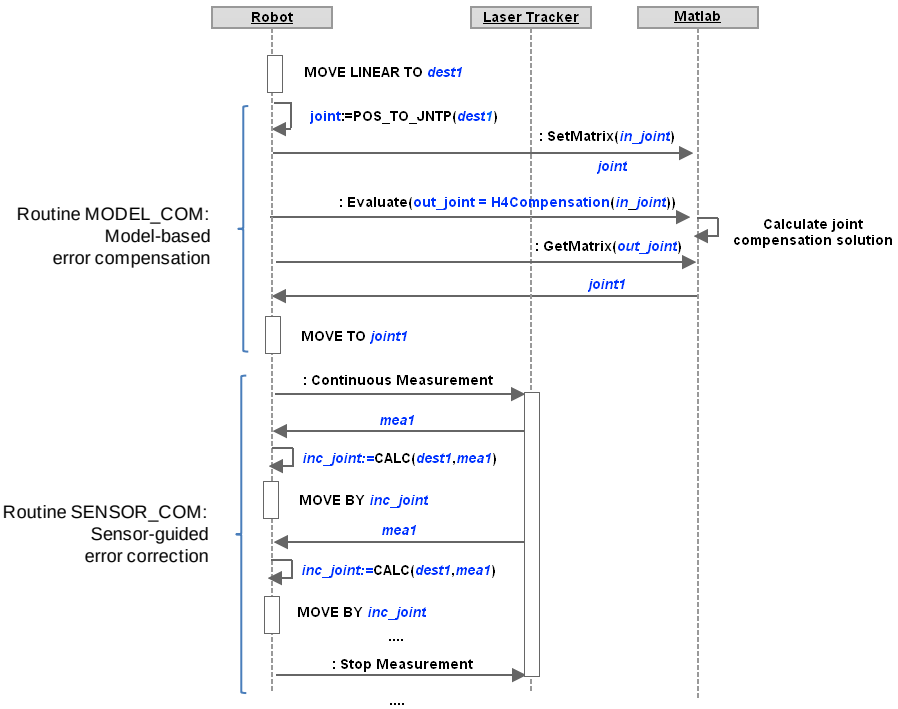


Figure 8-11 Implementation of the two-stage error compensation

8.3.3 Results and analysis

Initial position errors of the robot in the 36kg loaded case are shown in Figure 8-12. It can be seen from the figure that the deviations in the X and Z axes are biased toward positive and negative directions, respectively. These could be attributed to constant offsets at joint 2 and 2' as well as the deflections induced by the masses of the forearm and upper arm and the applied deadweight. When the robot moves from/to the two ends of the Y axis, the deflections cause errors not only in the X and Z axes but the Y axis as well. The 3σ absolute positional accuracy of the robot before correction is 6.50mm.

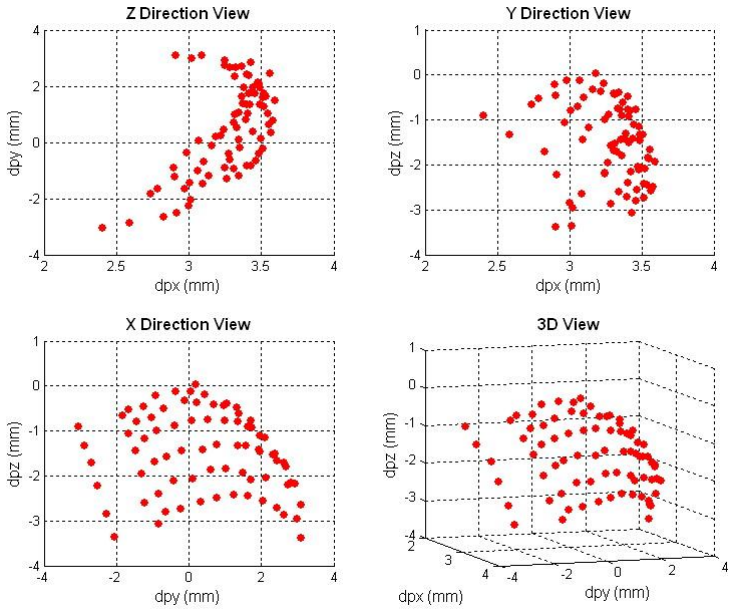


Figure 8-12 Initial position errors of the robot in the 36kg loaded case

Figure 8-13 depicts the positional accuracy of the robot after the model - based and sensor - guided error compensations. From the figure, it can be seen that the error compensation model reduced the deviations in the X, Y and Z axes to less than ±0.7mm while the laser tracker was able to reduce these errors within the range ±0.1mm. The 3σ absolute accuracies in the positions (the square root of these components) of the robot in each correction stage are achieved as 0.87mm and 0.12mm. The 3σ accuracies in the orientations of the robot,

calculated in a similar manner, are obtained as 0.44° and 0.05°. The results are summarized in Table 8-5.

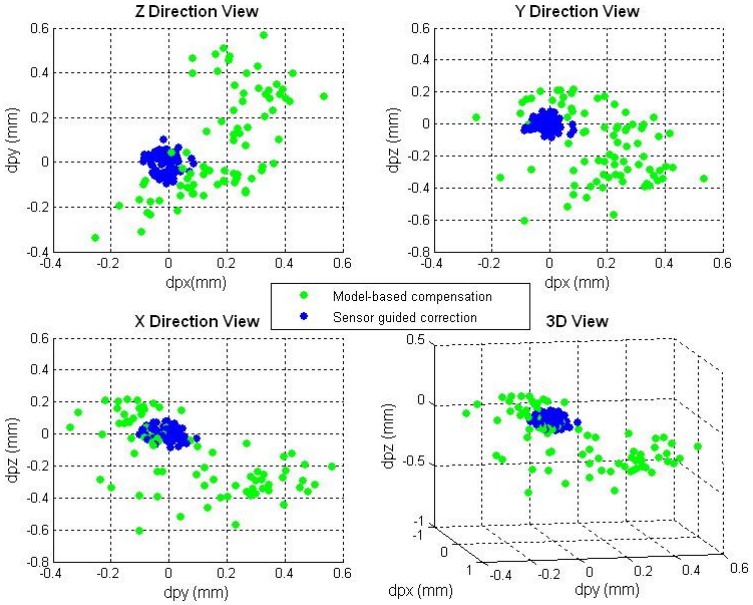


Figure 8-13 Residual position errors of the robot in the 36kg load case

Table 8-5 Absolute accuracy of the robot before and after the two stage error compensation

Models	Errors in position (10^{-3}m)		Errors in orientation (°)					
	$\overline{\Delta p}$	$\sigma_{\Delta p}$	$\overline{\Delta\phi_x}$	$\sigma_{\Delta\phi_x}$	$\overline{\Delta\phi_y}$	$\sigma_{\Delta\phi_y}$	$\overline{\Delta\phi_z}$	$\sigma_{\Delta\phi_z}$
(0)	4.04	0.82	0.665	0.468	-0.051	0.018	0.729	0.465
(1)	0.39	0.16	0.041	0.104	0.004	0.008	0.047	0.105
(2)	0.06	0.02	0.003	0.011	-0.005	0.009	0.003	0.013

(0): initial, (1): model-based compensation, (2): sensor-based correction.

Details on the sensor-based correction process for the robot's position and orientation components are illustrated in Figure 8-14 and Figure 8-15. During *t*=0,..,5s, measurements of the laser tracker, taken within 50ms, are used to adjust the robot's joint angles whereas at *t*= -2, -1 and 6s, the measurements are taken within 1s to verify the robot's locations at each stage and are used to calculate the results in Table 8-5. Deviations between the measurements at *t*=5s when the iterative correction process completes and *t*=6s are 0.02mm in

Δp and 0.015° in $\Delta\phi$. The deviations are due to the uncertainty of the measurement at *t*=5s, which is taken in much shorter time and when the robot still oscillates a small amount from a full stop. This explains why the overall 3σ accuracy obtained (0.12mm) is slightly worse than the tolerance for stopping the process (0.08mm). These variations can be eliminated by increasing the measuring time of the laser tracker and decreasing the velocity and deceleration of the robot at the cost of a longer cycle time for correction.

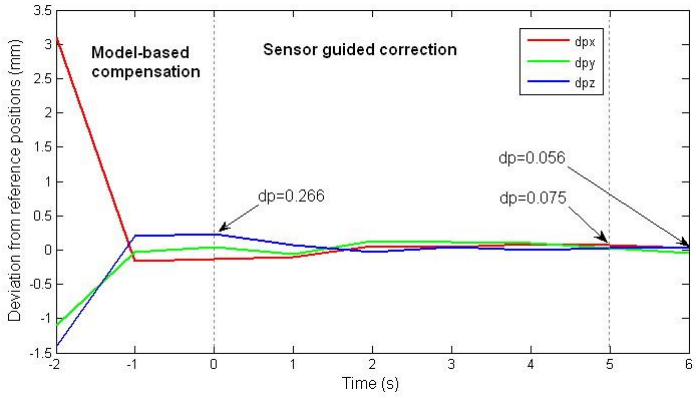


Figure 8-14 Correcting the robot position to ±0.08mm using the laser tracker

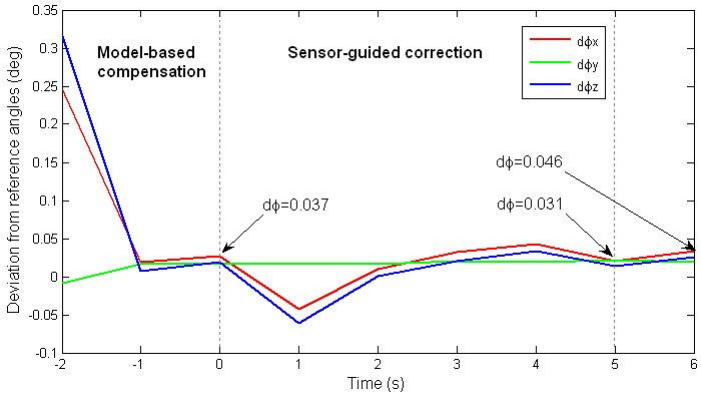


Figure 8-15 Correcting the robot orientation to ±0.05º using the laser tracker

The average time for correcting one robot position is 6 seconds for 6 iterations (or 1 iteration/sec). This includes the time for measuring and transferring the data by the laser tracker to the robot controller, computing the joint increment

solution by the robot service, data conversion, robot motion and computing the iterative condition by the C3G robot controller. Even though the performance is comparable to the work of (Kihlman, 2005), it is longer than expected. Slow computing performance originates from the C3G controller side: while modern robot controllers utilize Gigahertz PC for computing and 100MB/s Ethernet for communication, the old-fashioned C3G controller, manufactured in 1998, uses a 12MHz Motorola 68020 microcontroller and RS-232 at 19.2KB/s baud rate, respectively. Nevertheless, the model-based error compensation does reduce the time for sensor-based correction: when the call to the routine MODEL_COM was turned off in the PDL robot program, it usually took more than 11 seconds for the laser tracker to correct a robot position. The proposed two-stage error compensation, therefore, reduces nearly half of the time supposedly spent for robot positional correction.

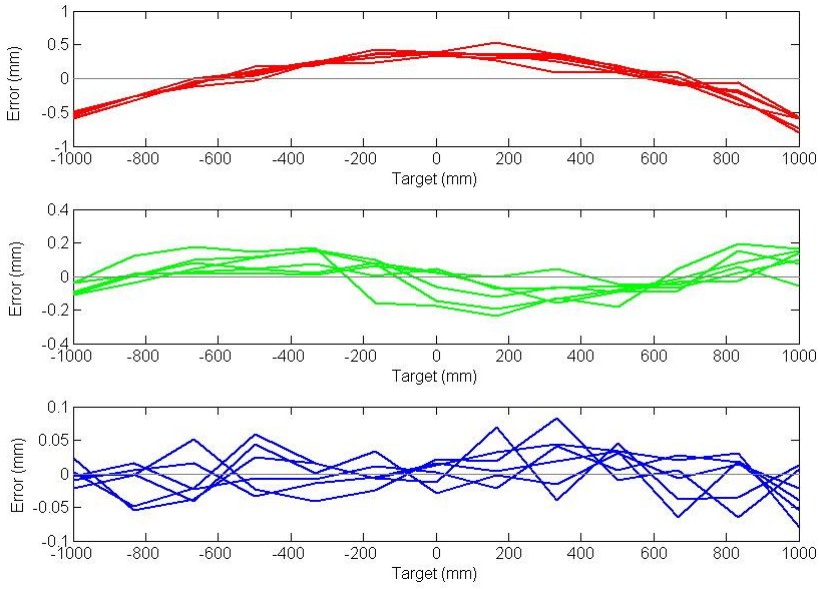


Figure 8-16 Straightness showing the deviation in the Z-axis when the robot travels along the Y axis. From top to bottom: a. Initial; b. After model-based compensation; c. After sensor-based correction.

Finally, the results of the straightness of Y axis calculated from the 6 trial runs are shown in Figure 8-16 and Figure 8-17. It can be seen from Figure 8-16a and Figure 8-17a that the deviations in the Z-axis and X-axis before calibration gradually increased when the robot moved toward the two ends of the Y axis

(when it extended the upper arm and forearm). Since the deviations calculated in this manner are independent of the coordinate system, this observation confirms the statement made earlier that the robot undergoes significant joint offsets and deflections at joint 2 and 2'. These errors cause large errors on the end-effector when joint 2 and 2' angles are large (see equations (6.12) and (6.31) of Chapter 6). This effect was, however, suppressed by the error compensation model as shown in Figure 8-16b and Figure 8-17b, proving the errors at these joints have been modelled properly. The deviations in the Z and X axes initially were within error bands of 1.32mm and 0.67mm then 0.43mm and 0.44mm after the model-based compensation and finally reduced to 0.16mm and 0.13mm after the sensor-based correction.

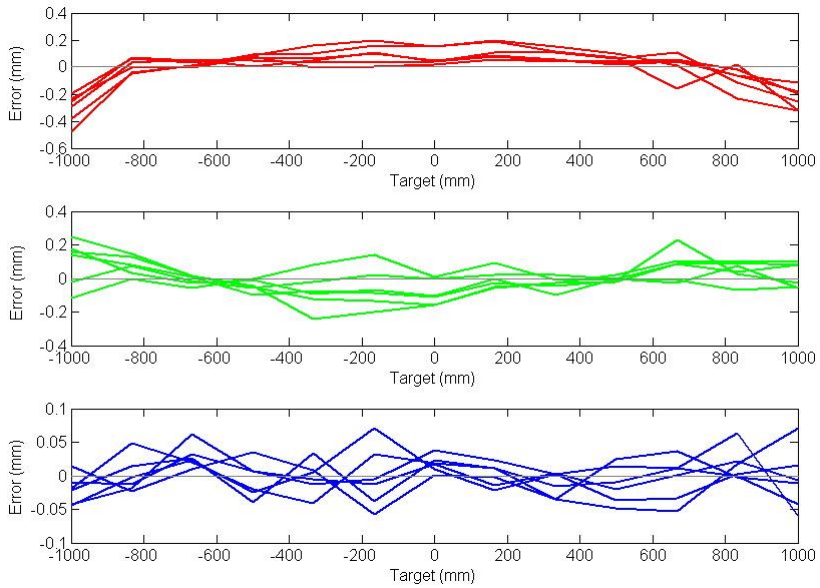


Figure 8-17 Straightness showing the deviation in the X-axis when the robot travels along the Y axis. From top to bottom: a. Initial; b. After model-based compensation; c. After sensor-based correction

8.4 Demonstration

8.4.1 Description

The final experiment was conducted to demonstrate the main ideas presented in this thesis, i.e., PnP system integration, the combination of model-based and

sensor-based error compensation for improving robot accuracy and the use of one metrology system for multiple robots. In the experiment, the Comau Smart H4 robot is commanded to carry an aluminium bracket to twenty four designated target locations on a real aircraft stabilizer structure. Precision holes were made on the centre of the bracket surface and four steel bars attached on the stabilizer to define the TCP frame and the target frames to be aligned with (Figure 8-18). A webcam placed behind the bracket is used to visualize the alignment of these holes.

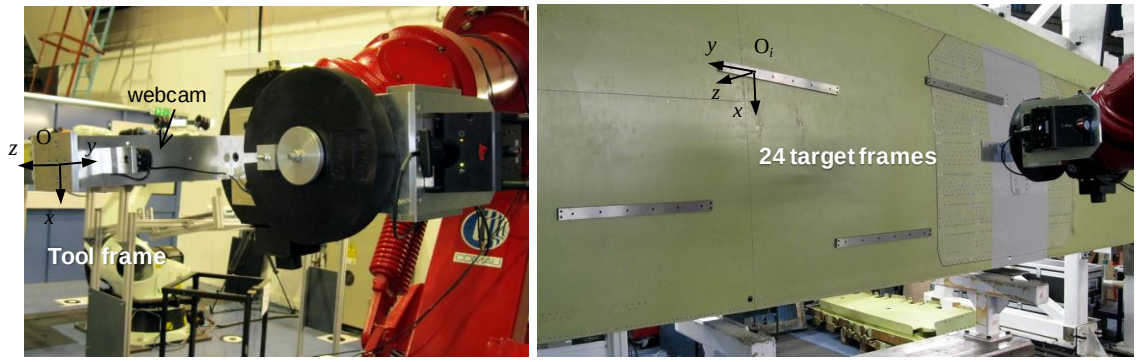


Figure 8-18 In the demonstration, the robot must align the tool frame with 24 target frames located on a stabilizer structure

The process is simulated in DELMIA using rough estimates of the target frames to generate the desired robot motions (Figure 8-19a). Precise coordinates of the target frames are determined from the laser tracker's measurements of the holes as follows:

- Each bar contains six target holes in the middle. Measured positions of a SMR put into these holes provide the origins of the target frames.
- The common z-axis of these target frames is the normal vector of the best-fit plane passing through the four holes at the corners of the bars.
- The common y-axis of these target frames is the directional vector of the best-fit line passing through the six holes; the last x-axis is determined from the right hand rule: $x = y \times z$ (where $\times$ denotes the cross product). The origins and the x, y and z axes constitute fully the position and orientation components of these target frames.

- The 24 target frames, calculated in the laser tracker frame, are transformed into the robot base frame (Figure 8-19b).

The intention of these calculations is to simulate the part localization process (sections 2.3.3.2 and 3.2.1) where a robot usually employs a local sensor to measure features on the part and applies similar best-fit algorithms to determine the actual location of the part before performing an assembly task. The robot used in this work is not equipped with such a local sensor and thus, the holes are measured manually and the measurements are stored in a developed Matlab function named *CalcTarget* which calculates the target frames for later retrieval.

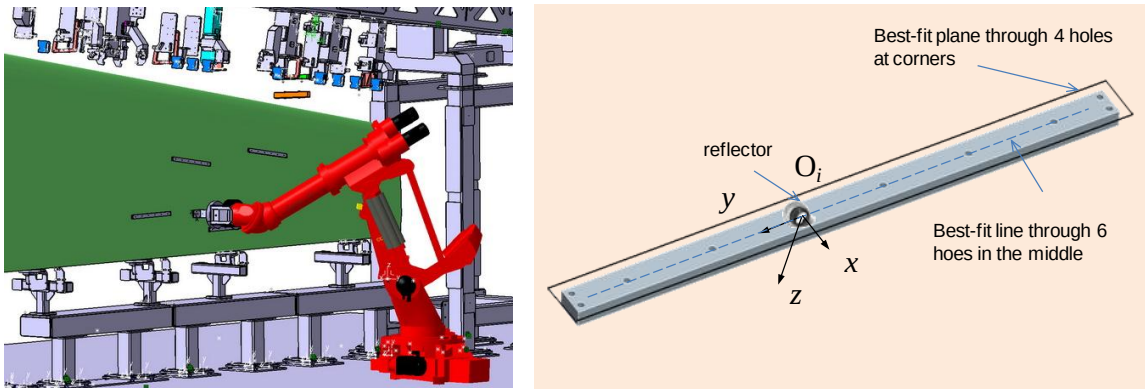


Figure 8-19 Robot motions are generated in DELMIA whereas actual target coordinates are constructed from the best-fit geometries on the measurements of the laser tracker

8.4.2 Implementation

The robot operation is carried out as depicted in Figure 8-20. At the start of the robot program, the routine *INITIALIZE* is called to load the identified parameters into Matlab workspace memory. Next, the robot invokes the Matlab function *CalcTarget* to calculate the 24 target frames and transform them into robot coordinates with the given matrix $BASE$. The robot retrieves the targets sequentially, advances to them in the motion profile programmed in DELMIA then calls the *MODEL_COM* and *SENSOR_COM* routines that initiate the model-based and sensor-based error compensations. The calls to the above

routines are inserted at every name tag in the offline robot program created by DELMIA.

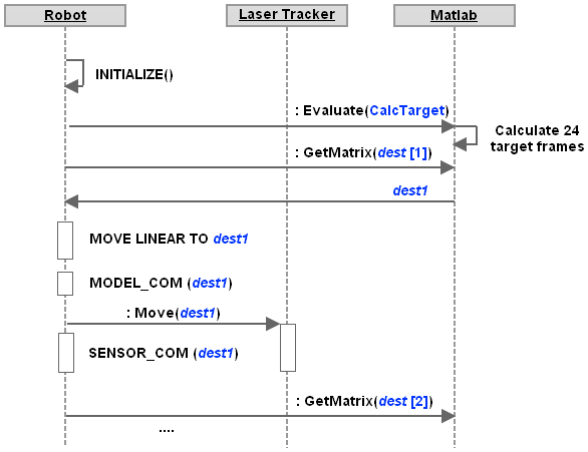


Figure 8-20 The process of hole alignment performed by the Smart H4 robot

During the process, another service representing a virtual Kuka robot is activated. This service, developed using the design template in Chapter 5, is wired with the laser tracker service by the cell controller and commands the laser tracker to take a single measurement of a SMR fixed in space periodically. The laser tracker, therefore, must serve two robots, the Comau and the (virtual) Kuka robots (Figure 8-21). Each robot must send a command *Move* to laser tracker service to rotate the laser head toward the current location of its reflector before the measurement takes place.

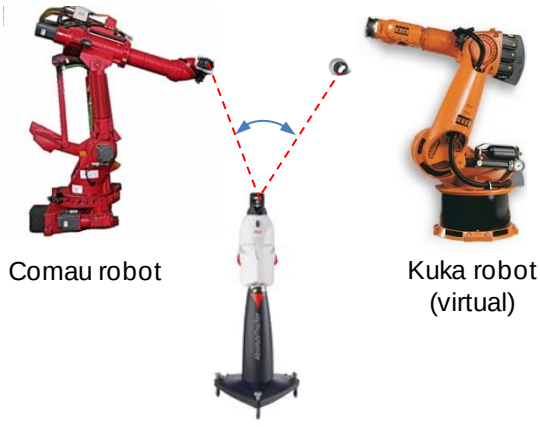


Figure 8-21 The laser tracker serving the Comau and virtual Kuka robots

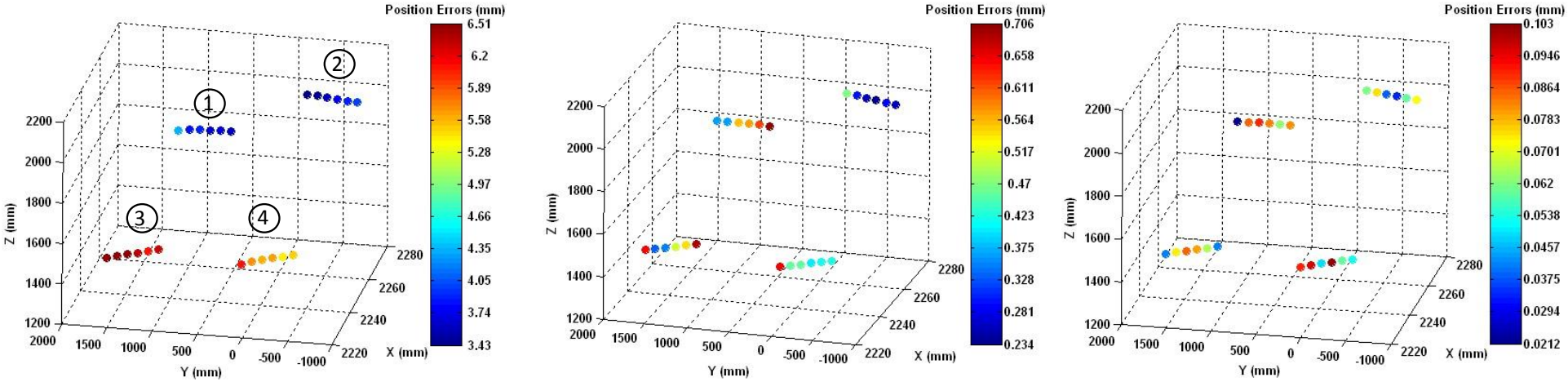


Figure 8-22 Accuracy in the position of the robot.

From left to right: a. Before correction; b. After model-based compensation; c. After sensor-based correction

(note: the scales of colour maps are different)

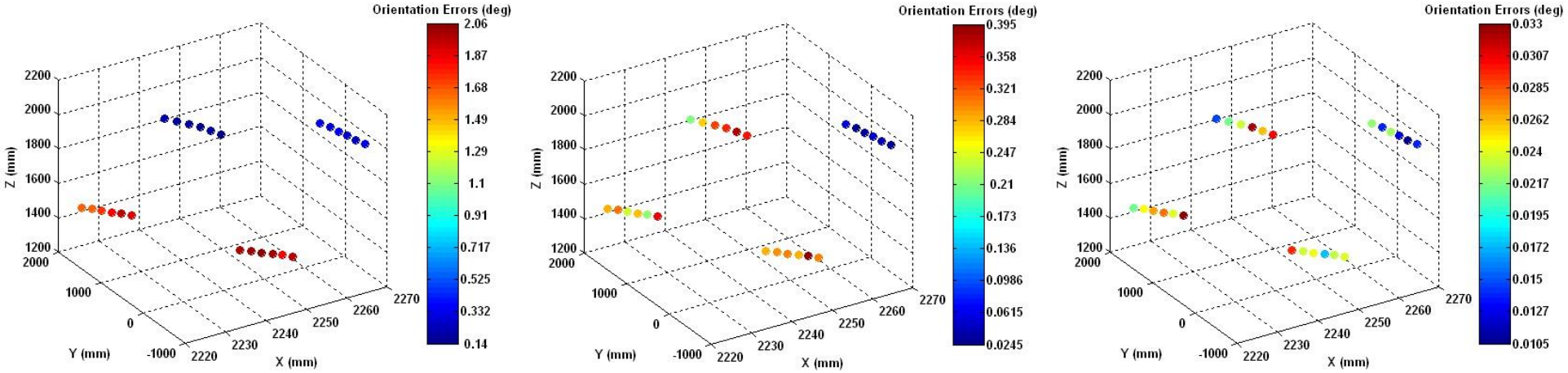


Figure 8-23 Accuracy in the orientation of the robot.

From left to right: a. Before correction; b. After model-based compensation; c. After sensor-based correction

(note: the scales of colour maps are different)

8.4.3 Results and analysis

Figure 8-22 and Figure 8-23 show the positional and angular accuracies of the robot before and after the two stage error compensation versus its coordinate locations. The mean values of the errors in positions and orientations of the robot before and after corrections are (4.91mm, 0.45mm, 0.06mm) and (1.08°, 0.23°, 0.03°), respectively. It can be seen from Figure 8-22a that the robot positional accuracy before correction is worst at bars 3 and 4. Details on the errors in individual X, Y, Z components of these measurements reveal that those in the Y and Z axes at bars 3 and 4 are dominant at these locations.

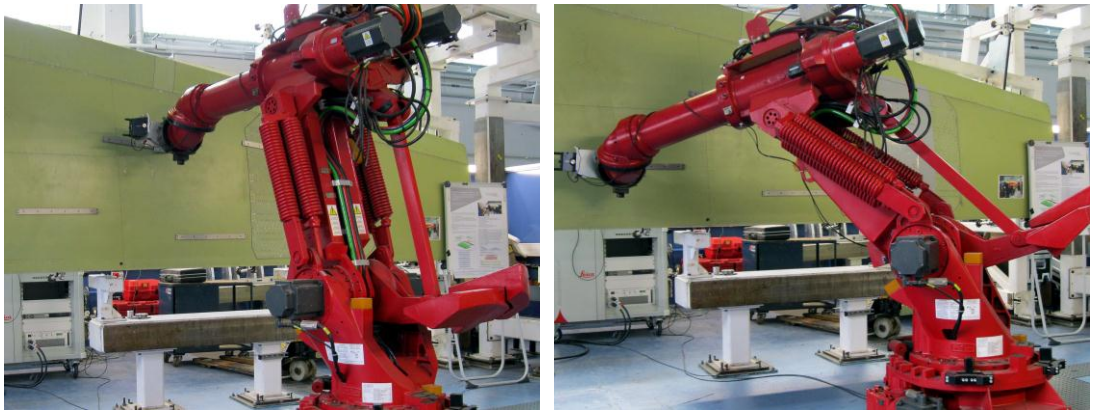


Figure 8-24 . Robot configurations during the process. From left to right: a. At bar 1, where the initial accuracy is highest; b. At bar 3, where the initial accuracy is lowest.

The above results are consistent with those obtained in sections 8.2.3, 8.3.3 and can be explained from the configurations of the robot during the alignment process depicted in Figure 8-24. At bars 1 and 2 positions, the upper arm is almost vertical and thus, the deflection at joint 2 induced by the link masses is small while at bar 3 positions, the arms are extended and thus, the deflections are larger. Deflection at joint 2', on the other hand, might be less severe due to the counterweight that balances the structure. These deflections result in the deviations in the Y and Z axes. The errors, however, have been compensated during the two stage correction process, as shown in Figure 8-22b,c. Figure 8-25 shows some images taken by the webcam during the alignment process.

In the figure, the complete circle seen is the hole on the bracket manipulated by the robot (the TCP). The dark circle seen through the bracket is a hole on the bars attached on the stabiliser (the target). One might see that these holes are overlapped after the sensor-based compensation.

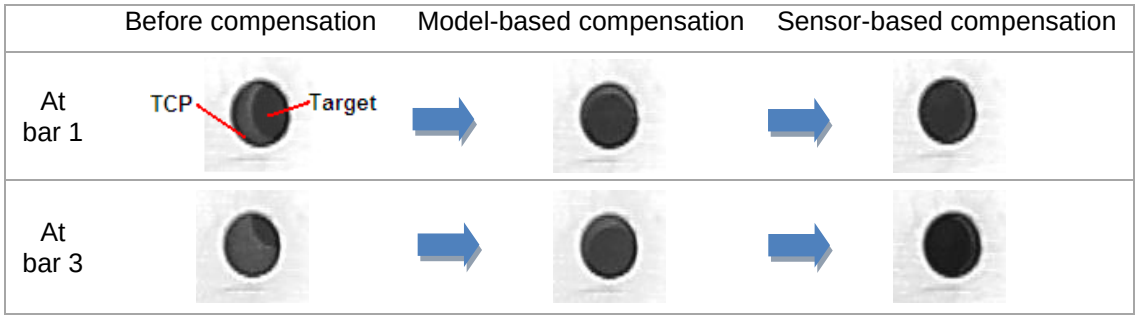


Figure 8-25 Visualisation of the alignment process

The utilization of one laser tracker for two robots is illustrated in Figure 8-26. Thanks to the error compensation model that narrows down the error band of the Comau robot to less than 1mm, the laser beam, rotated by the command *Move*, always finds the TMAC prism having diameter of 10mm in space. At a specific time when the laser tracker is busy correcting the position of the Comau robot, a command sent from the virtual Kuka robot will be queued and re-scheduled. This command will be processed after the laser tracker has been released (i.e., when it receives a *Stop Measurement* command sent by the Comau robot within the SENSOR_COM routine). The experiment has proved that it is possible to deploy one laser tracker for more than one robot.

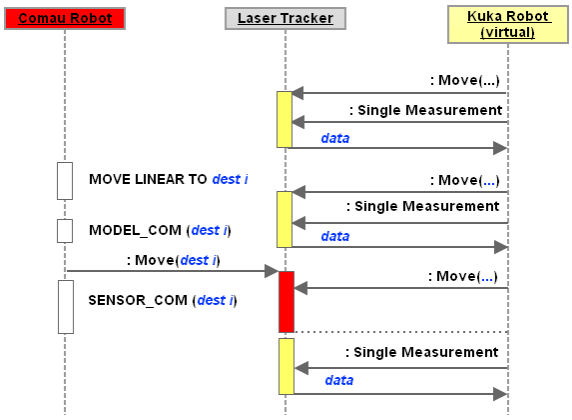


Figure 8-26 Two robots sharing one laser tracker

9 CONCLUSSIONS

9.1 Summary

Insufficient accuracy has been one of the main barriers to a wide adoption of robotics in airframe assembly. To overcome this, the robot must have improved accuracy delivered through some forms of error compensation from a global metrology system and other local sensors. Adding this equipment to a robot, however, raises issues on the flexibility and cost of such a system. The work presented within this thesis attempts to provide technical solutions to these problems.

9.1.1 A framework for flexible system integration

To improve the flexibility of a metrology-integrated robot system, a software framework which enables reconfigurable system integration was developed. Background research and literature review on the middleware technology and component-based software engineering that promotes flexibility have been performed. The framework provides a design template to develop distributed software components, namely services, each of which controls one or a subset of automation equipment (e.g., robots, actuators, sensors). These services can be integrated in a “Plug and Produce” (PnP) manner, that is, the connectivity between them can be established at runtime, instead of compile time. This allows for the robot system or, in a larger scale, a multi-robot work-cell to be assembled on demand for various assembly applications.

In order to achieve the PnP integration capability, the services are provided with a common interface inherited from an abstract service. They also share a comprehensive set of predefined data structures that facilitates the exchange of various command types, data sizes and formats among the represented equipment. As a result, all the services in the framework appear to be identical from their viewpoints and thus, they can be hot-plugged together at runtime. After the control system has been setup in this manner, it is possible to use robot programming language to program a new application for the robot and its

peripheral devices. This is a relatively simple job compared with programming new control software, and thus, can be done by technicians on the floor with aids of OLP software.

Internally, each service utilizes a task queue guarded by a rescheduling algorithm to sequence simultaneous requests that it might receive from other services. This is necessitated by the fact that the work-cell might contain some shared equipments supposedly used by several robots, such as a global metrology system (e.g., a laser tracker).

The middleware used by the framework for the communication between its services is the Robotics Developer Studio from Microsoft, selected on the basis of its support for concurrent programming and asynchronous communication. The shortcoming of this middleware platform is its limited communication rate, which is mainly due to the non real-time characteristics of the Windows OS. Assessments on the performance of the developed framework in terms of communication throughput and latency have verified that the safe update frequency of the framework is 100Hz. This update rate is not problematic for static correction, such as robot positioning and part localisation techniques performed in this work but might be insufficient for other dynamic correction activities that demand for particularly high rate and low latency communication, such as force control. This suggests future work to be done to improve the framework performance in this regard.

9.1.2 Error modelling and compensation for robots

In order to reduce the investment cost, a two-stage (model-based and sensor-based) error compensation scheme that promotes one expensive piece of metrology system for several robots was developed. The main purpose of the model-based compensation in the first stage is to narrow down the initial error band of the robot and thereby, reducing the time needed for sensor-based correction in the second stage. As a result, the metrology system does not have to service one robot all the time and thus, is able to support a number of robots simultaneously.

The error compensation model that improves robot accuracy in the first correction stage is obtained from a kinematic calibration procedure. The literature review undertaken in this thesis has shown that robot calibration is a well-established research topic and somewhat standardized for purely open-loop serial robots (e.g., the elbow-type manipulators). Nevertheless, there has been no simple calibration model for the ones having a closed kinematic chain (e.g., a parallelogram linkage). In this thesis, a novel calibration model for this type of industrial robot was presented. The error model of the parallelogram linkage was derived from the linearization of the chain's constraint equation and merged with that of the main open-loop branch to form the global calibration model of the manipulator. This model accounts for not only the errors of the robot structure but those in the pre-calibrated transformations between the robot and the global metrology system. The advantages of the proposed model are its simplicity and computing efficiency, validated by a benchmarking simulation study against another competitive model undertaken in this work. Then, the model was further expanded to include the joint deflections induced by the robot link masses and applied load (e.g., weight of the end-effector). Algorithms for the robot calibration and the aforementioned two-stage error compensation strategy are also provided.

9.1.3 Experimental evaluations

Experiments were conducted to evaluate the two pieces of work presented above. The design template of the framework is adopted to develop a distributed control system that performs the calibration and error compensation for a real parallelogram linkage-type robot. The control system was formed at runtime by "plugging" the control applications of the robot, laser tracker and computing software that implements the calibration and model-based compensation algorithms together. Thereafter, the calibration and the two-stage error compensation processes were carried out by activating different robot programs. The experimental results are summarized as follows:

- When the robot is unloaded, the proposed calibration model is able to predict the tool pose errors with an average accuracy of 0.4mm, 100% better than the conventional model that neglects errors in parallelogram linkage. Therefore, taking these errors into calibration does improve the accuracy. When the robot is loaded, the prediction accuracy degrades roughly 0.1mm for every incremental 18kg, which is mainly due to the un-modelled deformations of the robot links and joint bearings.
- The 3σ positional accuracy of the robot before correction is 6.50mm. The model-based error compensation is able to improve the accuracy of the robot up to 0.87mm in position and 0.44º in orientation. This is a good result compared with respective research in the literature and commercial software. The accuracy achieved by the sensor-based correction method is 0.12mm in position and 0.05º in orientation, smaller than the 0.2mm tolerance required in airframe assembly.
- The straightness of the Y axis of the robot, measured as the deviations in the Z and X directions when moving along the Y axis, initially were 1.32mm and 0.67mm. The model-based compensation reduces these deviations to 0.43mm and 0.44mm whereas the sensor-based correction further suppresses them to 0.16mm and 0.13mm, respectively.
- The model-based correction helps to reduce nearly half of the time for the iterative sensor-based correction. The average time for correcting one robot position by using this two-stage error compensation technique is 6s.
- The performance of the sensor-based correction stage in terms of accuracy and time is not entirely satisfactory and is mainly due to limitations in the motion resolution of the robot and the computing capability of its industrial controller. The author believes that newer robots will deliver higher performances than the one used in this work.
- The experiment also demonstrates successfully a part localisation algorithm and the use of one laser tracker for two robots.

9.2 Contributions

In summary, the major contributions of the thesis are as follows:

- Development and implementation of a framework for PnP system integration. Services adopted the design template provided by the framework are able to integrate dynamically at runtime, making it possible for the robot work-cell to be assembled on demand for various applications. Once the services have been connected, it is possible to use a robot's programming language to control the peripheral devices as if they were local resources of the robot.
- Demonstration of a two-stage error compensation strategy for industrial robots in airframe assembly that promotes the use of one expensive metrology system for several off-the-shelf robots.
- Development of a new calibration model for serial robots having a parallelogram linkage that takes into account geometric errors and joint deflections. The model has proved to be simple and computing-efficient.

The work presented in this thesis envisages a highly dynamic robot work-cell which might be suitable not only for the aerospace industry but other small and medium enterprises that also have to deal with the small batch, product diversity and cost issues (Brogårdh, 2007). In this work-cell, the number of the robots, metrology and their locations can be altered to adapt to various part sizes and shapes whereas the robots are also able to use different end-effectors to perform various operations, e.g., machining, measuring or part handling. Under these circumstances, the proposed framework helps to realize a customized control system in which generated OLP robot programs for the new manufacturing process can be streamlined directly to the robot controllers for execution without the need for translation. The calibration technique helps to determine accurately the new locations of the robots in the work-cell area as well as develop models for the robots that compensate for their inherent errors and deflections induced by the weights of the new end-effectors or the parts they handle. The distributed control system and the two-stage error

compensation scheme allow for a number of robots to share one metrology system whenever it is possible. These technical solutions eventually help to reduce the lead time and cost and increase the responsiveness of such a robotic manufacturing system compared with a conventional one that utilizes strongly-coupled hardware and proprietary control/calibration software applications.

9.3 Future works

It is suggested that the following research activities should be performed in the future to address remaining limitations of this work:

- Improving the communication rate of the framework in terms of latency so that it is potentially used for a wider range of dynamic correction applications. A viable solution is porting the services of the equipments supposedly used for this purpose to the Windows Embedded CE, a real-time operating system (RTOS) for embedded automation devices including Intel PCs. Critical assessment on the performance of the services on this RTOS needs to be performed.

- Improving the data structures of the framework to support redirection of sensory information. In the calibration process presented in Chapter 8, measurement data from the laser tracker must be sent to the robot first then forwarded to the Matlab software. A better *Command* structure will allow for the data to be transferred directly to the terminal device, provided their services are connected (as in Figure 4-4). This will save unnecessary time for data delivery/conversion and memory space for the robot controller.

- Developing calibration and compensation models and procedures for robots located on linear rails or mobile platforms. The accuracy of such a robot system is further degraded by the straightness and orthogonality of the augmenting axes.

- Improving the accuracy and time-efficiency of the iterative sensor-based positional correction. This can be achieved through an optimized selection of the measuring time of the laser tracker and velocity of the robot in order to reduce the measurement uncertainty.

- Implementing more realistic assembly applications. For example, the robot performs a machining application then the positional accuracy and surface quality of the drilled holes will be verified by an independent CMM. Next, the robot replaces its end-effector to perform a different application, e.g., part handling, to demonstrate the versatility of the system. For each application, a new distributed control application of the robot system is formed by connecting the robot service with those of the corresponding end-effectors and metrology together.

REFERENCES

ABB, ***Absolute Accuracy - Industrial Robot Option***, ABB Robotics, 2010.

ABB, **ABB Review** – ***The Corporate Technical Journal of the ABB Group***, ABB Robotics, http://www.abb.com/abbreview, 2010.

Ahn, S.C., Lee, J.W., Lim, K.W., Ko, H., Kwon, Y.M and Kim, H.G., **Requirements to UPnP for Robot Middleware**, IEEE/RSJ Int. Conf. on Intelligent Robots and Systems, Beijing, China, Oct 9-15 2006.

Airbus**,** ***How an aircraft is built*****,** www.airbus.com, visited on Jan 04 2012.

ALCAS Project**,** ***Advanced Low Cost Aircraft Structures*****,** http://ec.europa.eu/research/transport-/projects/items/alcast_en.htm, visited on Jan 04 2012**.**

Alici, G., ***A systematic approach to develop a force control system for robotic drilling,*** *Industrial Robot*, Vol. 25, No. 5, pp. 289-397, 1999.

Alici, G., Shirinzadeh, B., ***A systematic technique to estimate positioning errors for robot accuracy improvement using laser interferometry based sensing***, Mechanism and Machine Theory, Vol. 40, pp. 879–906 , 2005.

Amoretti, M., Caselli S., and Reggiani, M.**,** ***Designing Distributed, Component-Based Systems for Industrial Robotic Applications*****,** Industrial Robotics: Programming, Simulation and Applications, InTech, Dec 2006.

Ananthananyanan, S.P., Szymczyk, C., Goldenberg, A.A., ***Identification of Kinematic Parameters of Multiple Closed Chain Robotic Manipulators Working in Coordination***, IEEE Int. Conf. on Robotics Automation, Nice, France 1992.

Angeles, J., ***Fundamental of Robotic Mechanical Systems: Theory, Methods and Algorithms 2nd edition***, Springer, 2003.

Antonelli, G., Chiavellini, S., Perna, V., Romanelli, F., ***A Modular and Task-oriented Architecture for Open Control System: the Evolution of C5G Open towards High Level Programming***, The ICRA 2010 Workshop on Innovative Robot Control Architectures for Demanding (Research) Applications, Lund University, Sweden, Nov 2010.

ARFLEX Project, ***Adaptive Robots for Flexible Manufacturing System project summary***, http://www.arflexproject.eu, visited on Jan 2012.

Arnone, J., ***Bombardier to Combine Efficiency and Quality in Manufacturing of CSeries Aircraft***, Bombardier Press Release, May 31 2011.

Atkinson, J., Hartman, J., Jones, S., Gleeson, P.**,** ***Robotic Drilling System for 737 Aileron*****,** SAE AeroTech Congress & Exhibition, Los Angeles, CA, USA, Sep 2007.

Axelsson, S., ***Offline Programming of Robots at Volvo Cars – shop floor preparation***, 33rd Int. Symp. On Robotics, 7 Oct 2002 Cited in: Kihlman, H., ***Affordable Automation for Airframe Assembly***, PhD Dissertation at Linköping University, 2005.

Bai, Y., Wang, D., ***Improve the Robot Calibration Accuracy Using a Dynamic Online Fuzzy Error Mapping System***, IEEE Trans. Systems, Man, and Cybernetics — Part B: Cybernetics, Vol. 34, No. 2, 2004.

Bai, Y., Zhuang, H., ***On the Comparison of Bilinear, Cubic Spline, and Fuzzy Interpolation Techniques for Robotic Position Measurements***, IEEE Trans. Instrumentation Mea. , Vol. 54, No. 6, 2005.

Bäuml, B., Hirzinger, G., ***When hard realtime matters: Software for complex mechatronic systems***, Robotics and Autonomous Systems, Vol.56, pp.5–13, 2008.

Bennett, D.J., Hollerbach, J., **Autonomous calibration of single-loop closed kinematic chains formed by manipulators with passive endpoint constraints**, *IEEE J. Robot. Autom.* 11(5), 597–606, 1995.

Bernhardt, R., Albright, S.L., ***Robot Calibration***, Chapman & Hall 1993.

Blomdell, A., Bolmsjö, G., Brogårdh, T., Cederberg, P., Isaksson, M., Johansson, R., Haage, M., Nilsson, K., Olsson, M., Olsson, T., Robersson, A., Wang, J., ***Extending an Industrial Robot Controller***, IEEE Robotics & Automation Magazine, Sep 2005.

Bone, G., Capson, D., ***Vision-guided fixtureless assembly of automotive components***, Robotics and Computer Integrated Manufacturing, Vol. 19, pp.79–87, 2003.

Brogårdh, T., ***Present and future robot control development – An industrial perspective***, Annual Reviews in Control, Vol. 31, pp. 69-79, 2007.

Brugali, D., ***Software Engineering for Experimental Robotics***, Springer Tracts in Advanced Robotics, Vol. 30, 2007.

Bruyninckx, H., ***Open Robot Control Software: the OROCOS project***, Int. Conf. Robotics & Automation, Seoul, Korea, May 21-26 2001.

COMET Project, ***COMET robot machining consortium meets at Fraunhofer***, http://www.cometproject.eu, visited in Jan 2012.

Calder, N., ***New dawns for robotics***, Aerospace Manufacturing, Jan 07 2011.

Colon, E., Sahli,H**.,** ***CoRoBA, an Open Framework for Multi-Sensor Robotic System Integration*****,** Proceedings of the IEEE International Symposium on Computational Intelligence in Robotics and Automation, Espoo, Finland, June 27-30 2005.

Comau Robotics – ***Smart H4 robot manual*** – 1998

Costlow, T.**,** ***Robots Improve Drilling Precision For F-35 Fuselage*****,** SAE Aerospace Engineering, Feb 12 2009.

Da Costa, S., ***A New Type of Robotic Cell for Assembly in Aeronautics*****,** SAE Aerospace Automated Fastening Conference & Exposition, Bellevue, Washington, USA, Oct 1992.

Da Costa, S., ***Dassault Adaptive Cells***, Industrial Robots, Vol.3, No. 1, pp.34 40, 1996.

Deitert, L., ***Orbital Drilling***, SAE 2011 AeroTech Congress & Exhibition, Toulouse, France, Oct 11.

Deter, S., ***Plug and participate for limited devices in the field of industrial automation***, 8th IEEE Int. Conf. on Emerging Technologies and Factory Automation, Antibes-Juan les Pins, France, Oct 15-18 2001.

Dilts, D.M., Boyd, N.P., Whorms, H.H., ***The evolution of Control Architectures for Automated Manufacturing Systems***, Journal of Manufacturing Systems, Vol.10, No.1, 1991.

Dombre, E., Khalil, W., ***Modeling, Performance Analysis and Control of Robot Manipulators***, ISTE, 2007.

Driels, M.R., Pathre, U.S., ***Generalized Joint Model for Robot Manipulator Calibration and Compensation***, J. Robotic Systems 4(1), 77-114, 1987.

Drouet, Ph., Dubowsky, S., Zeghloul, S., Mavroidis, C., ***Compensation of geometric and elastic errors in large manipulators with an application to a high accuracy medical system***, Robotica, Vol. 20, pp. 341–352, 2002.

Duelen, G., Wendt, W., ***Research and Development of Industrial Robot Systems in Europe***, Robotics and Computer Integrated Manufacturing, Vol. 1, No.3/4, pp.339–348, 1984.

DeVlieg, R., Sitton, K., Freikert, E., Inman, J., ***ONCE (One Side Cell End-effector) Robotic Drilling System***, ElectroImpact Inc., 2002.

DeVlieg, R., Freikert, E**., One-up Assembly with Robots, SAE** Aerospace Manufacturing and Automated Fastening Conf. & Ex**.,** North Charleston, SC, USA, Sep 2008.

DeVlieg, ***Expanding the Use of Robotics in Airframe Assembly via Accurate Robot Technology***, *SAE Int. J. Aerospace. Vol.* 3, No. 1 pp. 198-203, 2010.

Eastwood, S.J., Webb, P., McKeown, C**.,** ***The use of the TI2 manufacturing system on a double-curvature aerospace panel*****,** Proc. Instn Mech. Engrs Vol. 217 Part B: J. Engineering Manufacture, 2003.

Eastwood, S.J., Webb, P., ***A gravitational deflection compensation strategy for HPKMs***, Robotics and Computer-Integrated Manufacturing, Vol. 26, Issue 6, pp. 694–702, 2010.

Editor Staff, ***First sale of novel robot goes to Boeing***, The Engineer at www.theengineer.co.uk/news/first-sale-of-novel-robot-goes-to-boeing/288194.article, Jan 15 2000.

Fayad, C., Chitiu, A., Webb, P., ***Control Of A Flexible Automatic Riveting System***, Proc. 8th Mechatronics Forum Int. Conf., Uni. Twente, Enschede, Netherlands, June 24-26 2000.

Garcia, J.G., Ortega, J.G., Garcia, A.S., Martinez, S.S., ***Robotic Software Architecture for Multisensor Fusion System***, IEEE Trans. Industrial Electronics, Vol. 56, No. 3, 2009.

Gooch, R., ***Optical metrology in manufacturing automation***, Sensor Review, Vol. 18, No.2, pp. 81-87, 1998.

Goodrich, ***Robotic Cell Coming to Chula Vista for Testing before Transitioning to its Permanent Home in Toulouse***, Around Aerostructures, Vol.3, No.10, Oct. 2011.

Gong, C., Yuan, J., Ni, J., ***Nongeometric error identification and compensation for robotic system by inverse calibration***, International Journal of Machine Tools & Manufacture, 40 2119–2137, 2000.

Grasson, M.**,** ***Laser Trackers Integrate with Automated Systems*****,** American Manufacturing, Oct 2011.

Gupta, A.K., Arora, S.K., ***Industrial Automation and Robotics***, Laxmi Publications, 2007.

Hayati, S.A., Mirmirani, M., ***Improving the absolute positioning accuracy of robot manipulator***, J. Robot. Syst. 2, 397–413, 1985.

Hanisch, H.M., Vyatkin, V., ***Achieving Reconfigurability of Automation Systems by using the New International Standard IEC 61499: A Developer's View***, Chapter 8 in Integration Technologies for Industrial Automated Systems, CRC Press, 2007.

Hochgurtel, B.*,* ***Cross-Platform Web Services Using C# and Java***, Charles River Media, 2003.

Hemsteads, B., DeVlieg, R., Mistry, R., Sheridan, M., ***Drill and Drive End-Effector***, ElectroImpact Inc., 2001.

Holden, R., Lightowler, P., and Brady, N., **Robot Integrated Metrology for Complex Part Manufacturing**, SAE Technical Paper 2010-01-1859, 2010.

Hong, K.S., Choi, K. H., Kim, J.G., Lee, S., ***A PC-based open robot control system: PC-ORC***, Robotics and Computer Integrated Manufacturing 17 (2001) 355–365.

Hsu, T.W., Everett, J.L., ***Identification of the kinematic parameters of a robot manipulator for positional accuracy improvement***, Proceedings of the International Computers in Engineering Conference and Exhibition, 263-267, 1985.

Hu, Z., Kruse, E., ***Web Services for Integrated Automation Systems — Challenges, Solutions, and Future***, Chap. 4 in Integration Technologies for Industrial Automated Systems, CRC Press, 2007.

Hung, M.H., WeiWu, S., LiWang, T., TienCheng, F., YunFeng, Y., ***An Efficient Data Exchange Scheme for Semiconductor Engineering Chain Management System***, Robotics and Computer-Integrated Manufacturing 26, pp. 507–516, 2010.

IEEE Standard, ***A Compilation of IEEE Standard Computer Glossaries***, Institute of Electrical and Electronics Engineers. New York, NY, 1990.

INS-NEWS, ***Swedish robot manufacturer flexes its muscles in the USA***, Industrial News Services, May 22 1998.

Integration Engineering Lab. at University of California, ***Four Bar Linkage***, http://iel.ucdavis.edu/chhtml/toolkit/mechanism/fourbar/fourbarpos.html, visited in Dec 2011.

Jackson, J., ***Robotic Studio: A Technical Introduction***, IEEE Robotics & Automation Magazine, 2007.

Jamshidi, J., Kayani, A., Iravani, P., Maropoulos, P., Summers, M., ***Manufacturing and Assembly Automation by Integrated Metrology Systems for Aircraft Wing Fabrication***, Proc. IMechE Vol. 224 Part B: J. Engineering Manufacture, 2010.

Jayaweera, N., Webb, P., ***Adaptive assembly of compliant aero-structure components***, Robotics and Computer Integrated Manufacturing, Vol. 23, pp. 180-194, 2007.

Johns, K., Taylor. T., ***Professional Microsoft Robotics Studio***, Wiley Publishing, 2008.

Judd, R.P., Knasinski, A.B., ***Technique to Calibrate Industrial Robots with Experimental Verification***, IEEE J. Robotics Automation Vol. 6. No. 1, 1990

Karan, B., Vukobratovic, M., ***Calibration and accuracy of manipulation robot models-An overview***, Mech. Mach. Theory Vol.29, No.3. pp.479-500,1994.

Kayani, A., Jamshidi, J., ***Measurement Assisted Assembly for Large Volume Aircraft Wing Structure***, Large Volume Metrology Conference, Liverpool, UK, Nov 4 2008.

Khalil, W., ***Modeling, Identification and Control of Robots***, Elsevier, 2004.

Kihlman, H., ***Affordable Automation for Airframe Assembly***, PhD Dissertation at Linköping University, 2005.

Kochan, A., ***New Production Technology For The New Generation of Airbus***, Industrial Robots, Vol. 11, No. 2, 1991.

Kröger, T. and Wahl, F.M., ***Multi-Sensor Integration and Sensor Fusion in Industrial Manipulation: Hybrid Switched Control, Trajectory Generation, and Software Development***, IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems, Seoul, Korea, Aug 20 - 22, 2008.

Kubus, D., Sommerkorn, A., Kröger, T., Maaß, J., and Wahl, F.M., ***Low-Level Control of Robot Manipulators: Distributed Open Real-Time Control Architectures for***

Stäubli RX and TX Manipulators, The ICRA 2010 Workshop on Innovative Robot Control Architectures for Demanding (Research) Applications, Lund University, Sweden, Nov 2010.

KUKA, KMC – ***Kuka Motion Control***, Kuka Roboter, www.kuka-robotics.com, 2008.

Kurfess, T.R., ***Robotics and Automation Handbook***, CRC Press, 2005.

Lange, K., ***Numerical Analysis for Statisticians***, Springer, 1999.

Lee, K., ***A Smart Transducer Interface Standard for Sensors and Actuators***, Chap. 10 in Integration Technologies for Industrial Automated Systems, CRC Press, 2007.

Leica Geosystems – ***Absolute Interferometer Whitepaper and Laser tracker manuals*** – 2008.

Leitão, P., ***Agent-based distributed manufacturing control: A state-of-the-art survey***, Engineering Applications of Artificial Intelligence Vol. 22, pp 979-991, 2009.

Lott, S., ***A robotic revolution***, Aerospace Manufacturing Magazine, June 2011.

Lu, W., Gunarathne, T., Gannon, D., ***Developing a Concurrent Service Orchestration Engine in CCR***, Proceedings of the 1st international workshop on Multi-core software engineering, ACM New York, USA, 2008.

Marie, S., Maurine, P., ***Elasto-Geometrical Modelling of Closed-Loop Industrial Robots Used For Machining Applications***, IEEE Int. Conf. on Robotics and Automations, Pasadena, CA, USA, May 19-23 2008.

Microsoft, ***Microsoft Robotics Developer Studio R3 Documentation***, 2008.

Minhas, S. H., Lehmann, C., Staedter, J. P., Berger, U.**, *Reconfigurable Strategies for Manufacturing Setups to confront Mass Customization Challenges***, Proceedings of the 21st International Conference on Production Research (ICPR 21), Stuttgart, Germany, July 31 – August 4, 2011.

Mizukawa, M., Sakakibara, S., Otera, N., ***Implementation and Applications of Open Data Network Interface 'ORiN'***, SICE Annual Conference in Sapporo, Aug 4-6 2004.

Montgomery, D., Runger, G., **Applied Statistics and Probability for Engineers**, John Wiley & Sons, 2003.

Mooring, B.W., Tang, G.R., ***An improved method for identifying the kinematic parameters in a six axis robot***, Proceedings of the International Computers in Engineering Conference and Exhibition, 1, pp.79-84, 1984.

Morey, B., **Robotics seeks its role in aerospace**, Manufacturing Engineering Magazine, Vol.139 No. 4, Oct. 2007.

MRDS, ***Microsoft Robotics Forums***, http://social.msdn.microsoft.com/Forums/en-US/category/robotics/, visited on Jan 2012.

Namoshe, M., Tlalel, N.S., Kumile, C.M., Bright, G., ***Open middleware for robotics***, 15th Int. Conf. on Mechatronics and Machine Vision in Practice (M2VIP08), Auckland, New-Zealand, Dec 2-4 2008.

Naumann, M., Wegener, K., Schraft, R.D. ***Control Architecture for Robot Cells to Enable Plug'n'Produce*****,** IEEE International Conference on Robotics and Automation Roma, Italy, April 10-14 2007.

Neumann, P., ***Communication in industrial automation: What is going on?***, Control Engineering Practice, Vol.15, pp. 1333-1347, 2007.

Nikon Metrology Inc., ***Transforming Industrial Robots into Precision Machine Tools***, Aug 30 2011.

Olsson, T., Haage, M., Kihlman, H., Johansson, R., Nilsson, K., Robertsson, A., Björkman. M., Isaksson, R., Ossbahr, G., Brogårdh, T., ***Cost-efficient drilling using industrial robots with high-bandwidth force feedback***, Robotics and Computer-Integrated Manufacturing, Vol. 26, pp. 24–38, 2010.

OMG, ***Documents Associated with CORBA 3.2***, http://www.omg.org/spec/ CORBA, Nov 2011.

OROCOS Project, **Open Robot Control Software**, www.orocos.org. visited in Dec 2011.

Pan, Z., Polden, J., Larkin, N., Van Duin, S., Norrish, J., ***Recent progress on programming methods for industrial robots***, Robotics and Computer Integrated Manufacturing, Vol. 28, pp.87–94, 2012.

Pan, M*.,* ***Real time communications over UDP protocol***, article on the Code Project developers' forum (www.codeproject.com), Nov 16 2011.

Paul, R.P, ***Robot Manipulators: Mathematics, Programming and Control***, MIT Press, Cambridge, 1981.

Paolini, C., Vuskovic, M., ***Integration of Robotic Laboratory using COBRA***, IEEE International Conference on Systems, Man, and Cybernetics, Florida, US, Oct 12-15 1997.

Parziale, L., Britt, D.T., Davis, C., Forrester, J., Liu, W., Matthews, C., Rosselot, N., ***TCP/IP Tutorial and Technical Overview***, IBM Redbooks, Dec. 2006.

Pereira, C., Carro, L., ***Distributed real-time embedded systems: Recent advances, future trends and their impact on manufacturing plant control***, Annual Reviews in Control, Vol.31, pp.81–92, 2007.

Pedrocchi, N., Malosio, M., Vicentini, F., Molinari Tosatti, L., and Legnani, G., ***Commercial Controllers enhancements and Open Source Robot Control Software: addressed solutions for demanding applications***, The ICRA 2010 Workshop on Innovative Robot Control Architectures for Demanding (Research) Applications, Lund University, Sweden, Nov 2010.

Pires, J.R., Sa da Costa, J.M.G., ***Object-oriented and distributed approach for programming robotic manufacturing cells***, Robotics and Computer Integrated Manufacturing 16, pp.29-42, 2000.

Pires, J.N., Ramming, J., Rauch, S., Araujo, R., ***Force/Torque Sensing Applied to Industrial Robotic Deburring***, Sensor Review, Vol. 22 No. 3, pp.232 – 241.

Pitzek, S., Elmenreich, W., **Configuration and Management of Fieldbus Systems**, Chapter 16 in Integration Technologies for Industrial Automated Systems, CRC Press, 2007.

Ple, P. , Gabory, J.F., Charles, F. , ***Force Controlled Robotic System for Drilling and Riveting One Way Assembly*****,** SAE 2011 AeroTech Congress & Exhibition, Oct 2011.

POPJIM Project, ***Plug and Produce Joint Interface Modules***, http://cordis.europa.eu/search/index.cfm?fuseaction=proj.document&PJ_RCN=11467257, visited in Jan 2012.

Renders, J., Rossignol, E., Becquet, M., Hanus, R., ***Kinematic Calibration and Geometrical Parameter Identification for Robots***, IEEE J. Robotics Automation*, Vol.* 7, No. 6, 1991.

Richards, M., ***KUKA's flexible solution for wing manufacture*****,** Article news on the Machinery at www.machinery.co.uk/machinery-news/kuka-s-flexible-robotic-solution-for-wing-manufacture/**,** May 17 2010.

Rubin, W., Brain, M., **Understanding DCOM**, Prentice Hall, 1999.

Saadat, M., Cretin, L., ***Measurement systems for large aerospace components***, Sensor Review, Vol. 22, No.3, 2002.

Saadat, M., Cretin, L., Sim, R., Najafi, F., ***Deformation analysis of large aerospace components during assembly***, Int. J. Adv. Manuf. Technol., Vol. 41, pp. 145-155, 2009.

Sanz., R. and Alonso, M., ***COBRA for Control Systems***, Annual Reviews on Control Vol. 25, pp.161-181, 2001.

Siciliano, B., Sciavicco, L, Villani, L., Oriolo, G., ***Robotics: Modelling, Planning and Control***, Springer, 2007.

Siciliano, B., Khatib, O., ***Handbook of Robotics***, Springer, 2008.

Schmelzer, R., Vandersypen, T., Bloomberg, A., Siddalingaiah, M., Hunting, S., Qualls, M., Houlding, D., Darby, C., Kennedy, D., ***XML and Web Services Unleashed***, SAMS Publishing, 2002.

Schmidt, D.C, ***Real-time COBRA with TAO (The ACE ORB)***, www.cs.wustl.edu-/~schmidt/TAO.html, 2010.

Schröer, K., Albright, S.L., Grethlein. M., ***Complete, minimal and model-continuous kinematic models for robot calibration***, Robotics & Computer Integrated Manufacturing, Vol. 13, No. 1, pp.73-85, 1997.

Schröer *K.,* Albright, S.L., Lisounkin, A., ***Modeling Closed-Loop Mechanisms in Robots for Purposes of Calibration***, IEEE J. Robotics Automation*,* Vol. 13, No. 2, 1997.

Short, M., Burn, K., ***A Generic Controller Architecture for Intelligent Robotic Systems***, Robotics and Computer-Integrated Manufacturing Vol.27, pp. 292-305, 2011.

SMERobot Project, ***The European Robotics Initiative for Strengthening the Competitiveness of SMEs in Manufacturing***, www.smerobot.org, 2009.

Song. H., Li, Y., Zhou., F., Jia, L., ***The Research and Application of Real Time CORBA in Software Framework for Industrial Robot***, Int. Conf. Integration Technology, Shenzhen, China, March 20 - 24, 2007.

Song, I., Karray, F., Gueda, F., ***A Distributed Real-time System Framework Design for Multi-Robot Cooperative Systems using Real-Time CORBA***, Proceedings of the 2003 IEEE International Symposium on Intelligent Control Houston. Texas, Oct 5-8 2003.

Spong, M.W., Vidyasagar, M., ***Robot dynamics and control***, John Wiley & Sons, 2004.

Stone, H.W., Sanderson, A.C., ***A prototype arm signature identification system***, Proceedings of IEEE International Conference on Robotics and Automation, pp.175-182, 1987.

Summers, M., ***Robot Capability Test and Development of Industrial Robot Positioning System for the Aerospace Industry***, SAE Technical Paper, Oct 03 2005.

To, M., Webb, P., ***An improved kinematic model for calibration of robots having closed-chain mechanisms***, Robotica, Vol.30, Issue 6, pp. 963-971, 2012.

Tu, H.T., Chen, M.S. and Kao, Y.C., ***A Web-Based Remote Machining Cell,*** *Proceedings of the IEEE International Conference on Mechatronics*, Taipei, Taiwan, July 10-12 2005.

UPnP Forum, ***UPnP Specification***, http://upnp.org, visited in Jan 2012.

Veitschegger, W.K., Wu, C.H., ***Robot Accuracy Based on Kinematics***, IEEE J. Robotics Automation, Vol.RA-2, No.3, 1986.

Veitsschegger, W.K., Wu, C.H., ***Robot Calibration and Compensation***, IEEE J. Robotics Automation*,* Vol. 4, No. 6, 1988.

Vitturi, S., ***On the use of Ethernet at low level of factory communication systems***, Computer Standards & Interfaces, Vol.23, pp.257-277, 2001.

Vyatkin, V., ***The IEC 61499 Standard and Its Semantics***, IEEE Industrial Electronics Magazine, Dec. 2009.

Wang, Z., Mastrogiacomo, L., Franceschini, F., Maropoulos, P., ***Experimental comparison of dynamic tracking performance of iGPS and laser tracker***, Int. J. Adv. Manuf. Technol., 2011.

Warhburg, J., ***Integrating multiple sensors and industrial robots: System architecture and control aspects***, IEEE Int. Sym. Intelligent Control, Arlington, VA, USA, Aug 24-26 1988.

Warwick, G., ***Airbus UK Using Automotive Robots in Wing Manufacture***, Flight International, Jun 05 2007.

Webb, P., Eastwood, S., ***An evaluation of the TI2 manufacturing system for the machining of airframe subassemblies*****,** Proc. Instn Mech. Engrs Vol. 218 Part B: J. Engineering Manufacture, 2004.

Webb, P., Eastwood, S., Jayaweera, N., Ye, C., ***Automated aero-structure assembly***, Industrial Robots, Vol. 32, No.5, pp. 383-387, 2005.

Weber, A., ***High Flying Robotics***, Assembly Magazine, Apr 29 2009.

Young, K., Pickin, C., ***Accuracy assessment of the modern industrial robots***, Industrial Robots, Vol.2, No.6, 2000.

Zhuang, H., Wang, K., Roth, Z.S., ***Error-model-based robot calibration using a modified CPC model***, Int. J. Robotics and Computer-Integrated Manufacturing 10(4), pp. 287-299, 1993.

APPENDICES

Appendix A Forward kinematic model

The position and orientation of a robot's end-effector in a reference frame are determined by the forward kinematic model. Suppose a serial manipulator has n+1 links numbered from 0 to n serially connected together via n actuated joints, numbered from 1 to n. The forward kinematic analysis of the robot starts out by assigning reference frames to the corresponding links following the popular Denavit - Hartenberg (DH) convention (Spong *et.al.*, 2004) (Figure A-1).

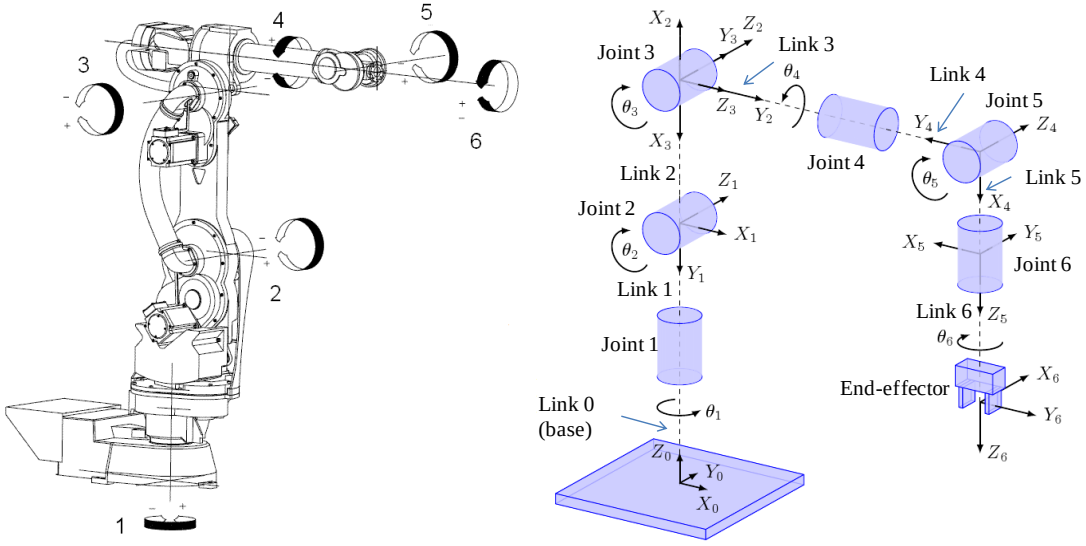


Figure A-1 The 6dof articulated robot (left) can be represented by a series of links and joints (right) with attached reference frames based on DH convention (*n*=6)

Position and orientation of the robot end-effector (the mobile frame F_n) w.r.t. the robot base (the fixed frame F_0) is represented by the homogeneous transformation matrix T_n^0 as:

$$T_n^0 = T_1^0 T_2^1 ... T_i^{i-1} ... T_n^{n-1} \tag{A.1}$$

in which, each T_i^{i-1} describes the relative position and orientation of link frame F_i w.r.t. link frame F_{i-1} as:

$$T_i^{i-1} = rot(z,\theta_i)tran(z,d_i)tran(x,a_i)rot(x,\alpha_i)$$

$$= \begin{bmatrix} c\theta_i & -s\theta_i c\alpha_i & s\theta_i c\alpha_i & a_i c\theta_i \\ s\theta & c\theta_i c\alpha_i & -c\theta_i s\alpha_i & a_i s\theta_i \\ 0 & s\alpha_i & c\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{(A.2)}$$

where $c\theta_i = cos\theta_i$, $s\theta_i = sin\theta_i$ and the four quantities θ_i, d_i, a_i, α_i are DH parameters associated with link i and joint i (Figure A-2). They are generally given the names joint angle, link offset, link length and twist angle, respectively. Three of the above four parameters are constant while the fourth parameter, θ_i for a rotary joint and d_i for a prismatic joint, is the joint variable.

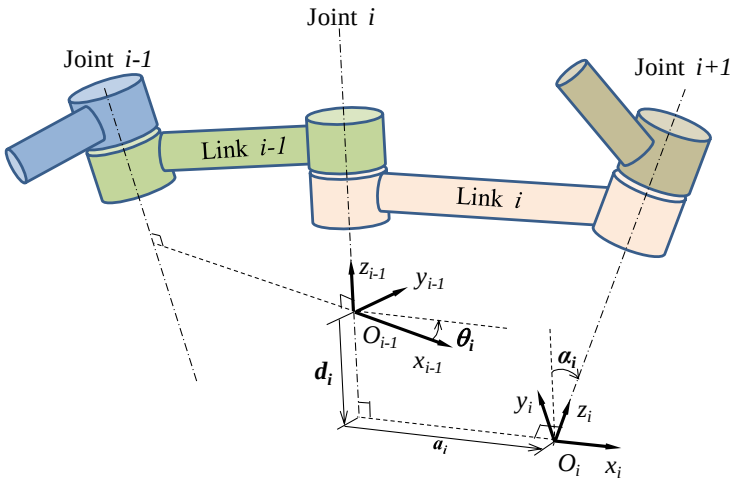


Figure A-2 Description of Denavit-Hartenberg parameters

The homogeneous transformation T_i^j ($i, j = 0 \ldots n$) in equations (A.1-2) has the general form of:

$$T_i^j = \begin{bmatrix} n_x & s_x & a_x & p_x \\ n_y & s_y & a_y & p_y \\ n_z & s_z & a_z & p_y \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} n_i^j & s_i^j & a_i^j & p_i^j \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} R_i^j & p_i^j \\ 0 & 1 \end{bmatrix} \quad \text{(A.3)}$$

where $n_i^j = (n_x, n_y, n_z)$, $s_i^j = (s_x, s_y, s_z)$, $a_i^j = (a_x, a_y, a_z)$ are directional cosine vectors of the x_i, y_i , z_i axes of frame F_i in a reference frame F_j. The vectors n_i^j, s_i^j, a_i^j can be grouped into the (3×3)- matrix R_i^j , namely the rotation matrix, representing the orientation of frame F_i w.r.t. the frame F_j. Vector $p_i^j = (p_x, p_y, p_z)$ represents the position of the origin O_i of frame F_i in the frame F_j. It can be seen from (A.1-3) that the position and orientation of the end-effector, p_n^0 and R_n^0, are functions of joint variables and other constant DH parameters.

For Comau robots, the position and orientation of the robot end-effector is represented as a vector $x = (x, y, z, A, B, C)$, where the A, B, C are the Euler ZYZ angles. These components can be converted from the given p_n^0 and R_n^0 as (Siciliano *et.al.*, 2007):

$$\begin{aligned} x &= p_n^0(1) \\ \mathrm{y} &= p_n^0(2) \\ z &= p_n^0(3) \\ A &= \mathrm{atan2}(R_n^0(2,3),\ R_n^0(1,3)) \\ B &= \mathrm{atan2}(\sqrt{R_n^0(1,3)^2 + R_n^0(2,3)^2},\ R_n^0(3,3)) \\ C &= \mathrm{atan2}(R_n^0(3,2),\ -R_n^0(3,1)) \end{aligned} \quad \text{(A.4)}$$

Appendix B Derivation of Kinematic Error Model using Differential Homogeneous Transformation

In terms of kinematic error modelling, the transformation T_i^{i-1} describing the relative position and orientation of link frame F_i w.r.t. link frame $F_{i\text{-}1}$ has the general form as:

$$T_i^{i-1} = rot(z,\theta_i)tran(z,d_i)tran(x,a_i)rot(x,\alpha_i)rot(y,\beta_i)$$

$$= \begin{bmatrix} c\theta_i c\beta_i - s\theta_i s\alpha_i s\beta_i & -s\theta_i c\alpha_i & c\theta_i s\beta_i + s\theta_i s\alpha_i c\beta_i & a_i c\theta_i \\ s\theta_i c\beta_i + c\theta_i s\alpha_i s\beta_i & c\theta_i c\alpha_i & s\theta_i s\beta_i - c\theta_i s\alpha_i c\beta_i & a_i s\theta_i \\ -c\alpha_i s\beta_i & s\alpha_i & c\alpha_i c\beta_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{(B.1)}$$

where β_i is the additional Hayati parameter for handling the case $z_{i\text{-}1}//z_i$. From this point onward, T_i^{i-1} is simply denoted as T_i.

The derivative of T_i due to small variations ($\Delta\theta_i, \Delta d_i, \Delta a_i, \Delta\alpha_i, \Delta\beta_i$) is:

$$dT_i = \frac{\partial T_i}{\partial \theta_i}\Delta\theta_i + \frac{\partial T_i}{\partial d_i}\Delta d_i + \frac{\partial T_i}{\partial a_i}\Delta a_i + \frac{\partial T_i}{\partial \alpha_i}\Delta\alpha_i + \frac{\partial T_i}{\partial \beta_i}\Delta\beta_i \quad \text{(B.2)}$$

Defining the differential homogenous transformation ΔT_i such as:

$$dT_i = T_i.\Delta T_i \quad \text{(B.3)}$$

or:

$$\Delta T_i = dT_i.T_i^{-1} \quad \text{(B.4)}$$

According to (Paul, 1980), such ΔT_i has the form:

$$\Delta T_i = \begin{bmatrix} 0 & -\Delta\phi_{zi} & \Delta\phi_{yi} & \Delta p_{xi} \\ \Delta\phi_{zi} & 0 & -\Delta\phi_{xi} & \Delta p_{xi} \\ -\Delta\phi_{yi} & \Delta\phi_{xi} & 0 & \Delta p_{xi} \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \text{(B.5)}$$

where $\Delta p_i = (\Delta p_{xi}, \Delta p_{yi}, \Delta p_{zi})^T$ and $\Delta\phi_i = (\Delta\phi_{xi}, \Delta\phi_{yi}, \Delta\phi_{zi})^T$ are equivalent differential translations and rotations of frame F_i about the axes of frame $F_{i\text{-}1}$. Indeed, expanding (B.4) gives:

$$\Delta T_i = \begin{bmatrix} 0 & -\Delta\theta_i & s\theta_i\Delta\alpha_i & c\theta_i\Delta a_i - d_i s\theta_i\Delta\alpha_i \\ \Delta\theta_i & 0 & -c\theta_i\Delta\alpha_i & s\theta_i\Delta a_i + d_i c\theta_i\Delta\alpha_i \\ -s\theta_i\Delta\alpha_i & c\theta_i\Delta\alpha_i & 0 & \Delta d_i \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

(B.6)

$$+\begin{bmatrix} 0 & -s\alpha_i\Delta\beta_i & c\theta_i c\alpha_i\Delta\beta_i & (a_i s\theta_i s\alpha_i - d_i c\theta_i c\alpha_i)\Delta\beta_i \\ s\alpha_i\Delta\beta_i & 0 & s\theta_i c\alpha_i\Delta\beta_i & (-a_i c\theta_i s\alpha_i - d_i s\theta_i c\alpha_i)\Delta\beta_i \\ -c\theta_i c\alpha_i\Delta\beta_i & -s\theta_i c\alpha_i\Delta\beta_i & 0 & a_i c\alpha_i\Delta\beta_i \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

One can see that (B.6) has the same form as (B.5). From (B.6), extracting the differential translation Δp_i and orientation $\Delta\phi_i$ vectors gives:

$$\Delta p_i = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}\Delta d_i + \begin{bmatrix} c\theta_i \\ s\theta_i \\ 0 \end{bmatrix}\Delta a_i + \begin{bmatrix} -d_i s\theta_i \\ d_i c\theta_i \\ 0 \end{bmatrix}\Delta d_i + \begin{bmatrix} a_i s\theta_i s\alpha_i - d_i c\theta_i c\alpha_i \\ -a_i c\theta_i s\alpha_i - d_i s\theta_i c\alpha \\ a_i c\alpha_i \end{bmatrix}\Delta\beta_i$$

$$\Delta\phi_i = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}\Delta\theta_i + \begin{bmatrix} c\theta_i \\ s\theta_i \\ 0 \end{bmatrix}\Delta\alpha_i + \begin{bmatrix} -s\theta_i c\alpha_i \\ c\theta_i c\alpha_i \\ s\alpha_i \end{bmatrix}\Delta\beta_i \qquad \text{(B.7)}$$

Notice that d_i and δ_i are expressed in frame $F_{i\text{-}1}$. When expressed in the base frame F_0, they are:

$$\Delta p_i^0 = R_{i-1}.\Delta p_i + p_{i-1}\times\Delta\phi_i$$

$$\Delta\phi_i^0 = R_{i-1}.\Delta\phi_i \qquad \text{(B.8)}$$

where R_{i-1} and p_{i-1} the rotation matrix and position of frame $F_{i\text{-}1}$ in F_0.

With reference to equations (A.2, A.3), substituting (B.7) into (B.8) yields:

$$\Delta p_i^0 = [z_{i-1}\times p_{i-1}]\Delta\theta_i + [z_{i-1}]\Delta d_i + [x_i]\Delta a_i + [x_i\times p_i]\Delta\alpha_i + [y_i\times p_i]\Delta\beta_i$$

$$\Delta\phi_i^0 = [z_{i-1}]\Delta\theta_i + [0]\Delta d_i + [0]\Delta a_i + [x_i]\Delta a_i + [y_i]\Delta\beta_i \qquad \text{(B.9)}$$

In the above equation x_i, y_i, z_i are directional vectors and p_i is the position of frame F_i in the base frame. Equation (B.9) can be written in matrix form as:

$$\Delta x_i^0 = \begin{bmatrix} z_{i-1}\times p_{i-1} & z_{i-1} & x_i & x_i\times p_i & y_i\times p_i \\ z_{i-1} & 0 & 0 & x_i & y_i \end{bmatrix}\Delta g_i \qquad \text{(B.10)}$$

where $\Delta x_i^0 = (\Delta p_i^0, \Delta \phi_i^0)^T$ and $\Delta g_i = (\Delta \theta_i, \Delta d_i, \Delta a_i, \Delta \alpha_i, \Delta \beta_i)^T$.

The differential translation and orientation vector of the end-effector are the linear summation of those of link frames, provided they are expressed in the same base frame:

$$\Delta x = \sum_{i=1}^{n} \Delta x_i^0 \tag{B.11}$$

Substituting (B.10) into (B.11) and one may obtain the following equation:

$$\Delta x = \mathrm{H}(g)\Delta g \tag{B.12}$$

where $\Delta x = (\Delta p, \Delta \phi)^T$, $\Delta g = (\Delta g_1 ... \Delta g_n)^T$ and $\mathrm{H}(g)$ is the $(6 \times 4n)$ identification Jacobian matrix:

$$\mathrm{H}(g) = \begin{bmatrix} z_{i-1} \times p_{i-1} & z_{i-1} & x_i & x_i \times p_i \; y_i \times p_i & \cdots \\ z_{i-1} & 0 & 0 & x_i \quad y_i & \end{bmatrix} \tag{B.13}$$

Equation (B.12) is the error model relating errors of the end-effector, represented as vector of differential translation and orientation, with errors in DH parameters for open-chained manipulators.

In order to solve (B.12), measurements of Δx from a sensor are needed. Denote the tool pose with nominal link parameters as T_N and the measured tool pose as T_M. The difference dT_N between T_M and T_N is:

$$dT_N = T_M - T_N \tag{B.14}$$

With reference to (B.4), the differential transformation ΔT can be obtained from (B.14) as:

$$\Delta T = (T_M - T_N).T_N^{-1} = T_M.T_N^{-1} - I \tag{B.15}$$

from which the differential translation and orientation vector Δx can be extracted. This is the measurement of Δx in equation (B.12).

In this work, the measurement T_M is provided by a laser tracker capable of measuring 6dof whilst the nominal T_N representing the kinematic chain from the laser tracker reflector to its base frame is given in equation (D.1).

Appendix C Position equations of a four bar linkage

A parallelogram structure with unequal opposite links can be considered a four bar linkage. Its position angles $\theta_{3'}, \theta_{4'}$ (Figure C-1) are computed based on the method given in (Integration Engineering Lab, 2011) as follows.

Define:

$$x = a_{2'} c\theta_{2'} - a_2 c\theta_2 \tag{C.1}$$

$$y = a_{2'} s\theta_{2'} - a_2 s\theta_2$$

$$\varphi_2 = \text{atan2}(y, x) + \text{acos}((a_{4'}{}^2 - x^2 - y^2 - a_{3'}{}^2) / 2a_{3'}\sqrt{x^2 + y^2})$$

$$\varphi_1 = \text{atan2}((y + a_{3'} s\varphi_2) / a_{4'}, (x + a_{3'} c\varphi_2) / a_{4'})$$

then:

$$\theta_{3'} = \varphi_2 - \theta_{2'}$$

$$\theta_{4'} = \pi + \varphi_1 - \varphi_2 \tag{C.2}$$

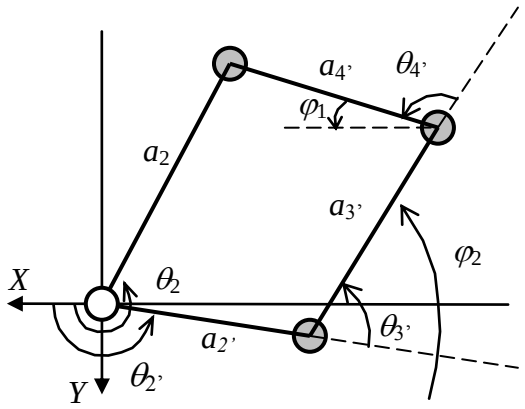


Figure C-1 A four bar linkage

Appendix D Determining the metrology and robot frames' transformations

The relationship between the laser tracker's measurement T and the position T_n^0 of a robot to be measured is:

$$T = BASE \cdot T_n^0 \cdot PROBE \tag{D.1}$$

where $BASE$, $PROBE$ are the transformations representing the robot base frame F_0 w.r.t. the laser tracker frame F_{LT} and the TMAC frame F_{TMAC} w.r.t. the robot flange frame F_n, respectively (Figure D-1).

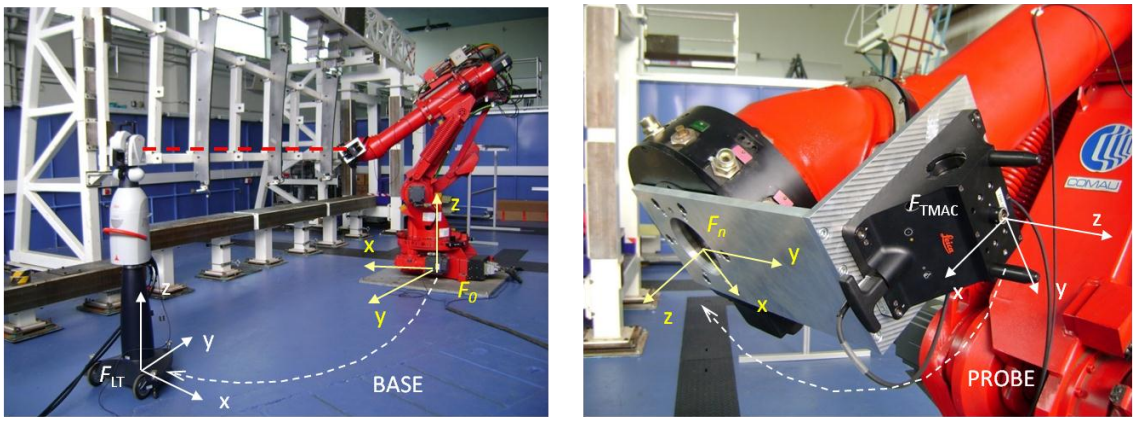


Figure D-1 Definitions of the BASE and PROBE transformations

Before the calibration process takes place, initial estimates of the $BASE$ and $PROBE$ transformations must be known. This section presents the procedures to determine the initial estimates of these two transformations.

D.1 Calibration of the PROBE transformation

Before the TMAC is mounted onto the flange surface, determine the instant position of frame F_n w.r.t. the laser tracker frame F_{LT} as follows:

- Firstly, fix a SMR reflector in a dowel hole on the flange surface and rotate the last joint (joint n) of the robot full round. Measured positions of the reflector during the rotation form a circle. By calculating a best-fit circle

through the measurements, position of the circle origin O_n (p_x,p_y,p_z) and normal vector z_n (a_x,a_y,a_z) of the plane containing it are obtained.

- Secondly, measure two dowel holes on the flange surface which define the y axis of the F_n frame then calculating the best-fit line passing the measurements, thus y_n (o_x,o_y,o_z) is known. The last axis x_n is then solved by using the right hand rule: $x_n(n_x,n_y,n_z)=y_n \times z_n$ (vector cross product).
- Thirdly, the transformation T_n^{LT} describing the instant F_n w.r.t. the laser tracker frame F_{LT} has the form:

$$T_n^{LT} = \begin{bmatrix} n_x & o_x & a_x & p_x \\ n_y & o_y & a_y & p_y \\ n_z & o_z & a_z & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{D.2}$$

- Finally, without moving the robot, mount the TMAC onto the flange surface, make one measurement of the TMAC frame F_{TMAC}. Denote the measurement as T_{TMAC}^{LT}. Since $T_{TMAC}^{LT} = T_n^{LT}.T_{TMAC}^{n}$, the constant $PROBE$ transformation, T_{TMAC}^{n}, is found as:

$$PROBE = T_{TMAC}^{n} = (T_n^{LT})^{-1}.T_{TMAC}^{LT} \tag{D.3}$$

The procedure is depicted in Figure D-2.

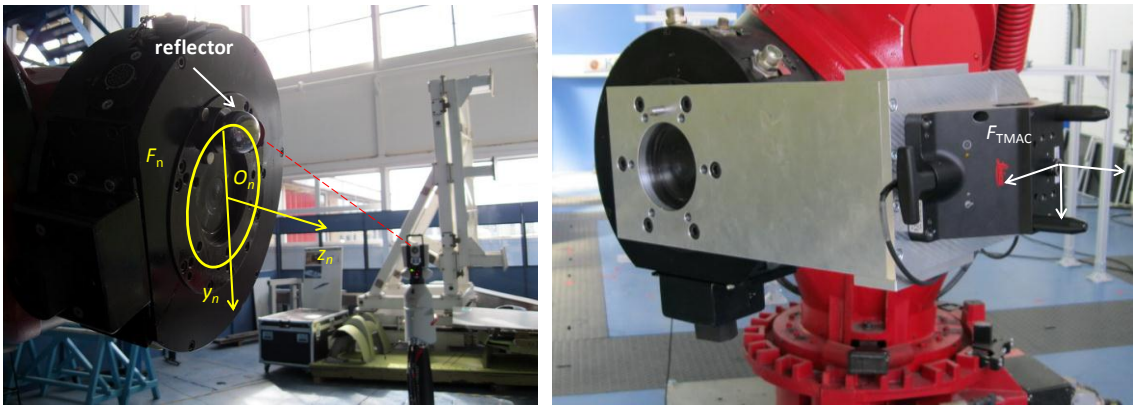


Figure D-2 Procedure to determine the PROBE transformation

The laser tracker visualization service developed in this work contains a wizard of the process and performs the above calculations automatically (Figure D-3).

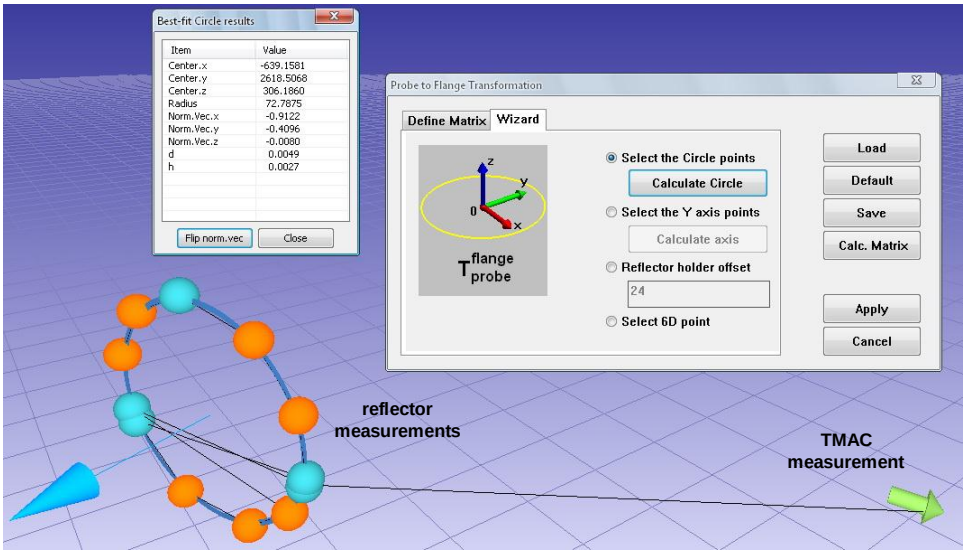


Figure D-3 The wizard for determining the PROBE transformation

D.2 Calibration of the BASE transformation

The procedure of calibration the $BASE$ transformation is carried out in a similar way to that of the $PROBE$ above:

- Firstly, fix the reflector on the outer surface of joint 1 of the robot. The reflector positions measured during the rotation of this joint form a circle. By fitting a best-fit circle through the measurements, the offset position O_0 and z_0 axis of frame F_0 are obtained.
- Secondly, fix the reflector on the base surface of the robot. Measure some positions to find the base plane (z=0) of frame F_0.
- Thirdly, move the robot along its x axis, and measure TMAC positions. The axis x_0 of frame F_0 is found by fitting a best-fit line through the measurements; its y_0 is solved by the right hand rule.

The frame F_0 determined via this procedure is expressed w.r.t. to the F_{LT} frame. Therefore, it is also the $BASE$ transformation. The procedure is depicted in Figure D-4.

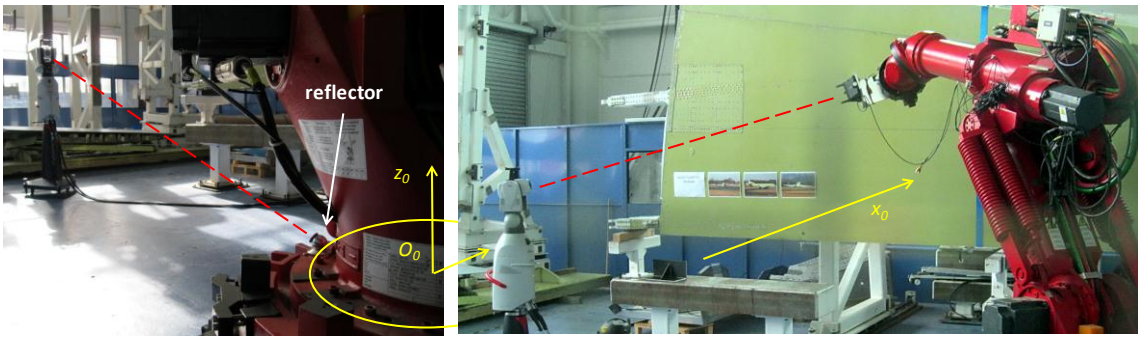


Figure D-4 Procedure to determine the BASE transformation

The laser tracker visualization service developed in this work contains a wizard of the process and performs the above calculations automatically (Figure D-5).

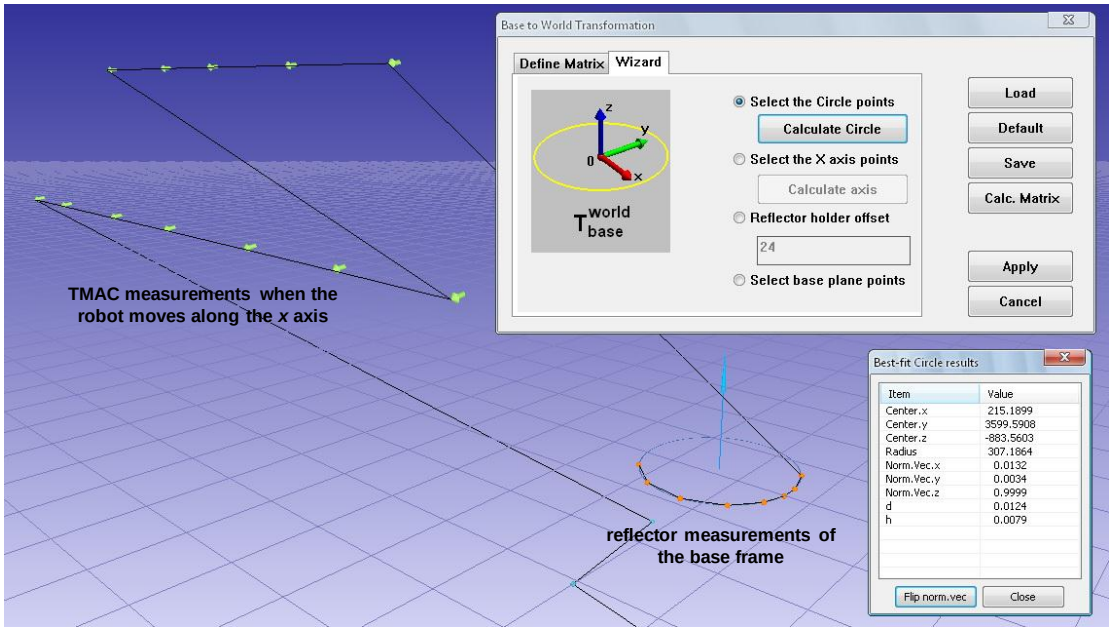


Figure D-5 The wizard for determining the BASE transformation

The manual calibration processes presented in sections D1-2 are only carried out when the relative positions between the laser tracker and the robot and between the TMAC and the flange have been altered completely from their original ones. When they are only slightly deviate, e.g., when the laser tracker is used for another activity then brought back to the place, re-calibration is not necessary because they will be automatically corrected by the error models presented in Chapter 6.

D.3 Jacobians of the BASE parameters

Recall equation (6.24) that the $BASE$ transformation is defined as:

$$BASE = Tran(x,a_0)Tran(y,b_0)Rot(x,\alpha_0)Rot(y,\beta_0)Rot(z,\theta_0)Tran(z,d_0) \quad \text{(D.4)}$$

of which the parameters $a_0, b_0, \alpha_0, \beta_0$ are identified to a higher degree of accuracy whilst d_0, θ_0 are identified through d_1, θ_1 of the robot (section 6.2.2). The Jacobians of these parameters are obtained by using symbolic Matlab as follows:

$$J_{a0} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}, J_{b0} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}, J_{\alpha 0} = \begin{pmatrix} 0 \\ 0 \\ -b_0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, J_{\beta 0} = \begin{pmatrix} b_0 s\alpha_0 \\ -a_0 s\alpha_0 \\ a_0 c\alpha_0 \\ 0 \\ c\alpha_0 \\ s\alpha_0 \end{pmatrix} \quad \text{(D.5)}$$

In order to calculate these Jacobians, the initial values of a_0, b_0, α_0 must be known in advance. When the matrix $BASE$ is determined from the procedure described in section D.2, the initial values of $a_0, b_0, d_0, \alpha_0, \beta_0, \theta_0$ can be obtained as:

$$\begin{aligned} \alpha_0 &= \text{atan}2(-BASE(2,3), BASE(3,3)) \\ \beta_0 &= \text{atan}2(BASE(1,3), -s\alpha_0 BASE(2,3) + c\alpha_0 BASE(3,3)) \\ \varphi_0 &= \text{atan}2(-BASE(1,2), BASE(1,1)) \\ d_0 &= BASE(3,4)/(c\alpha_0 c\beta_0) \\ a_0 &= BASE(1,4) - d_0 s\beta_0 \\ b_0 &= BASE(2,4) + d_0 s\alpha_0 c\beta_0 \end{aligned} \quad \text{(D.6)}$$

Appendix E Main components of the CCR

E.1 Port and PortSet

Port and *PortSet* are just FIFO message queues. Messages simply are data of a specific type. A *Port<T0>* can process only a single message type *T0* while a *PortSet<T0,T1,…,Tn>* aggregates several (max. 20) different types of messages. In fact, a *PortSet* is just a bunch of ports that can all be treated as a single entity. The main port of a service actually is a *PortSet* that accepts different messages defined in the service interface.

Messages can be sent to a local *Port* (or *PortSet*) of a service by the command *Post* or to the main port of another service by function calls described in section 5.1.1.2.

```
Port<int> intPort = new Port<int>();
intPort.Post(5);
intPort.Post(12);
```

In the above code snippet, a *Port* object *intPort* is created. Its data type is integer numbers. Two integer values, 5 and 12, are sent to the port and remain in the queue in that order until they are read by receivers.

E.2 Arbiter

The *Arbiter* static class provides a bunch of methods for creating receivers and other coordination models, such as the *Join*, *Choice* and the likes.

E.2.1 Receiver

Port and *PortSet* were just memory stacks if CCR did not implement receivers. These receivers register callback functions that are automatically activated on the arrival of messages at the ports. A receiver must be constructed and attached to a port as follows:

```
Arbiter.Receive<T0>(persist, port, Handler(T0));
```

where the *persist* flag indicates whether the receiver is persistent (a non-persistent receiver will be dismissed after processing the first message) and *Handler* is the callback function that will be executed when the port contains a value of type *T0*.

For example, a non-persistent receiver for the *intPort* created in the preceding example can be constructed as follows:

```
// Create a receiver and activate it
Activate(
    Arbiter.Receive(false, intPort,
    delegate(int n)
    {
        // Processing the received integer
        …
    })
);
```

In this example, the output will be 5. The value 12 still remains in the port until de-queued by another receiver. If the receiver were persistent, the output would be in the same order that the messages are queued in the port, 5 and 12. The *Handler* in this example is an anonymous method, which is a block of code passed to the C# delegate. Anonymous methods are normally used when the code block is relatively short. The *Activate* method is use to queue the delegate into the default *DispatcherQueue* of the service for execution.

Persistent receivers are vital to service-oriented architectures. Service handlers are simply persistent receivers on the main service port waiting for messages sent from other services. Therefore from this from onward, the terms receiver and handler will be used interchangeably without creating any confusion.

E.2.2 Choice arbiter

The *Choice* arbiter is effectively a logical OR. It waits on two ports until one of them receives a message, then shuts down the unused receiver.

```
Arbiter.Choice<T0,T1>(portset, Handler1(T0), Handler2(T1));
```

For example:

```
void Choice()
{
        PortSet<int, double> ps = new PortSet<int, double>();

        // Create the Choice and activate it
        Activate(
              Arbiter.Choice (ps,
              delegate(int n)
              {
                  Console.WriteLine("Integer value: " + n.ToString());
              },
              delegate(double d)
              {
                  Console.WriteLine("Double value: " + d.ToString());
              })
        );

        ps.Post(1000);
        ps.Post(3.14);
}
```

When this code is executed, the Choice arbiter is registered with the port `ps`. The usual output is 1000, but it's also possible for a value of 3.14 to be displayed depending on subtle timing variations in the CCR. Choice does not guarantee which port in a **`PortSet`** it will choose if both messages are already available.

The *Choice* is commonly used to handle response from service. The response can either be the successful data or an SOAP *Fault*, based on which the service can take appropriate actions (e.g. fault handling). An example of using *Choice* was shown in section 5.1.1.2.

E.2.3 Join arbiter

The *Join* arbiter is similar to a logical AND. It waits until messages arrive at both ports.

```
Arbiter.JoinedReceive(persist, port1, port2, Handler);
```

For example:

```
void Join()
{
        // Declare the Ports
        Port<bool>      p1 = new Port<bool>();
        Port<int>       p2 = new Port<int>();
        Port<string>    p3 = new Port<string>();

        // Create a Join on port 1 and 2
        Activate(
              Arbiter.JoinedReceive (false, p1, p2,
              delegate(bool b, int n)
              {
                  Console.WriteLine("Join 1: {0} {1}", b, n);
                  p2.Post(n+1);
              })
        );
        // Create a Join on port 2 and 3
        Activate(
              Arbiter.JoinedReceive (false, p2, p3,
              delegate(int n, string s)
              {
                  Console.WriteLine("Join 2: {0} {1}", n, s);
                  p2.Post(n-1);
              })
        );

        // Post values to the ports
        p1.Post(true);
        p3.Post("Hello");
        p2.Post(99);
}
```

When the function *Join* given above is executed, two *JoinedReceive* are activated immediately in two independent threads. The ports *p1*, *p2* and *p3* are then posted with their corresponding message types. Because the three messages arrive almost simultaneously at the ports, which are monitored by two concurrent receivers, the output could either be:

```
Join 2: 99 Hello
Join 1: True 98
```

or

```
Join 1: True 99
Join 2: 100 Hello
```

E.2.4 Interleave arbiter

Interleave is CCR's mechanism for multi-task synchronization. The *Interleave* consists of three groups of receivers: *ConcurrentReceiverGroup*, *ExclusiveReceiverGroup* and *TeardownReceiverGroup*. The *Concurrent-*

ReceiverGroup contains the receivers (handlers) that can run concurrently. The *ExclusiveReceiverGroup* only allows one receiver running at a time. If more exclusive receivers are triggered before the first one finished, they will be queued and executed one after another. The working mechanism of concurrent and exclusive groups of receivers is conceptually similar to the classical reader/writer lock paradigm in thread-based programming but is more elegant and less error-prone. Finally, the *TeardownReceiverGroup* contains the receivers that should be called before the whole *Interleave* is supposed to be shutdown, disposing other groups. Teardown receivers take the highest priority.

```
protected override void Start()
{
    base.Start();

    camPort.Subscribe(camNotify);
    camPort.Connect();

    // Create an Interleave for handlers of notifications from the camera service
    Activate(
        Arbiter.Interleave(
           new TeardownReceiverGroup(
              Arbiter.Receive <camera.Shutdown> (false, camNotify,
              CameraShutdownHandler)),
           new ExclusiveReceiverGroup(
              Arbiter.Receive <camera.MovieUpdate> (true, camNotify,
              MovieUpdateHandler)),
           new ConcurrentReceiverGroup()
    );
```

In the example given above, the robot service creates an *Interleave* for handlers of notifications from the camera service. The *MovideUpdateHandler* function is placed in the *ExclusiveReceiverGroup*, meaning that if the function spends more time for processing the image than the movie frame rate, the next triggered function will be queued. This *Interleave* will be disposed if the camera service is terminated (when the *Shutdown* notification is received).

E.3 Iterators

Another key concept is the *iterator*, a C# 2.0 feature that is used in a novel way by the CCR. When it is required to perform a sequence of asynchronous operations, the *iterator* allows the writing of code in a sequential fashion instead of having to use nested arbiters. An *iterator* is just a function (handler) declared with type *IEnumerator* <*ITask*>, meaning it will iterate over tasks. The compiler

will recognize that this type of function can contain *yield return* and *yield break* statements, which are not valid in a normal handler. The following code snippet will demonstrate the use of these statements.

```
private IEnumerator<ITask> JoinIterator(bool b, string s, int n)
{
        Port<bool>      p1 = new Port<bool>();
        Port<int>       p2 = new Port<int>();
        Port<string>    p3 = new Port<string>();
        bool continue = false;

        p1.Post(b);
        p3.Post(s);
        p2.Post(n);

        // Create the first Join on port 1 and 2
        yield return
              Arbiter.JoinedReceive (false, p1, p2,
              delegate(bool b, int n)
              {
                  Console.WriteLine("Join 1: {0} {1}", b, n);
                  continue = b;
              });

        if (continue == true)
        {
              p2.Post(n+1);

              // Create the second Join on port 2 and 3
              yield return
                  Arbiter.JoinedReceive (false, p2, p3,
                  delegate(int n, string s)
                  {
                        Console.WriteLine("Join 2: {0} {1}", n, s);
                        p2.Post(n-1);
                  });
        }
        else
              yield break;
}
```

In the *JoinIterator* given above, the *JoinedReceiver* on *p1* and *p2* ports is created first then depending on the *boolean* value posted to the *p1*, the second *JoinedReceiver* on *p2* and *p3* ports will be created later. Without using the *iterators*, the whole *if…else* branch must be placed inside the delegate function of the first receiver, making the code less readily. By using *iterators*, the code can be suspended each time the *yield return* in the code snippet is executed and resumed at some point later when the messages arrive. This effect is referred as *continuation*, which is very complex to achieve in ordinary event-based or thread-based programming (Lu *et.al.*, 2008). Therefore, the *iterators* in CCR provide outstanding advantages over the conventional asynchronous programming models.

Finally, to launch an *iterator*, e.g. the above *JoinIterator* routine, one can use the CCR's method *SpawnIterator*:

```
SpawnIterator<bool, int, string>(true, "Hello", 99,  JoinIterator);
```

which passes the parameters (*true, Hello, 99*) to the *JoinIterator* routine and execute it in an independent thread along with the main thread that executes the command *SpawnIterator* itself.

The output on the screen of the above code snippet is:

```
Join 1: True 99
Join 2: 100 Hello
```

Appendix F Predefined classes and enumerators

F.1 Device States

The enumeration *DeviceStates* represents the status of the device. It may have one of the following values:

Table F-1 The enumeration *DeviceStates*

Value	Descriptions
Ready	The device is ready to perform a task.
Busy	The device is busy executing a task.
Error	The device has some failures and cannot perform a task.

F.2 Connection States

The enumeration *ConnectionStates* represents the status of the connection between the service and the device controller. It may have one of the following values:

Table F-2 The enumeration *ConnectionStates*

Value	Descriptions
Online	The service is connected with its controller.
Offline	The service is disconnected or unable to connect with its device.
Error	The connection has some failures.

F.3 Command Types

The enumeration *CommandTypes* represents the type of a command. It may have one of the following values:

Table F-3 The enumeration *CommandTypes*

Value	Descriptions
NoData	The command does not return any data.
SingleData	The command returns single data. The data may be an array of items but are returned at once.
MultipleData	The command returns multiple data.
StopData	The command stops the one that sends multiple data.

F.4 Data Types

The enumeration *DataTypes* indicates the type of data that the command returns. It may have one of the following values:

Table F-4 The enumeration *DataTypes*

Value	Descriptions	Type conversion
Boolean	Boolean, Binary	C# type: boolean, .NET type: System.Boolean
Byte	8-bit unsigned integer	C# type: byte, .NET type: System.Byte
Integer	32-bit signed integer	C# type: int, .NET type: System.Int32
Double	64-bit floating point	C# type: double, .NET type: System.Double
String	Array of characters	C# type: string, .NET type: System.String

F.5 Command

The class *Command* contains information about the command that will be sent to the device controller for execution.

Table F-5 The class *Command*

Properties	Type	Descriptions
Name	string	Command name
Option	string	Optional parameter
SuccessCode	string	The feedback code of the device API indicating the command has finished without any errors from the controller side. Any other feedback means failure.
Type	CommandTypes	The type of the command.
InputDataType	DataTypes	The type of input data to the command, if any.
InputDataSize	int [ ]	The length of input data, if any.
OutputDataType	DataTypes	The type of output data from the command, if any (when the field Type is SingleData or MultipleData).
OutputDataSize	int [ ]	The length of output data, if any.

The class *Command* supports most common data types exchanged between electronic and software devices. Moreover, the length of data, represented by the fields *InputDataSize* and *OutputDataSize***,** are not restricted. Data can be a scalar, a vector or a matrix of *n*-dimensions of type *DataTypes*:

(a) If no data, *DataSize* is an array of one value: 0

(b) If the data is a scalar, *DataSize* is an array of one value: 1.

(c) If data is a vector, *DataSize* is an array of one value: n, which is the length of the vector.

(d) If data is a 2D matrix, *DataSize* is an array of two values: m and n, in which m is the number of rows, n is the number of columns of the matrix.

(e) If data is an n-D matrix, *DataSize* is an array of n values: $m_1 \ldots m_n$, in which m_i is the number of items in an i dimension.

F.6 Device

The class *Device* contains information about the represented device. It consists of the following members:

Table F-6 The class *Device*

Attribute	Type	Descriptions
Name	string	The unique name of the device.
Vendor	string	The manufacturer of the device
Info	string	Description on the device's functionality, model, accuracy etc.
Connection	ConnectionStates	The connection status between the service and the device controller
State	DeviceStates	The status of the device
Commands	Command [ ]	List of the commands that the device can perform
LastUpdated	DateTime	Time stamp of the latest update

F.7 Tag

The class *Tag* contains names of the sender and recipient of the request.

Table F-7 The class *Tag*

Value	Descriptions
Sender	Name of the service that sends the request
Recipient	Name of the service that receives the request

F.8 Process States

The enumeration *ProcessStates* represents the status of a process. It may have one of the following values:

Table F-8 The enumeration *ProcessStates*

Value	Descriptions
Queued	The process has been scheduled for execution.
Running	The process is being executed.
Completed	The process has completed without failures
Failed	The process has failed to complete

F.9 Process

The class *Process* contains information about the process one service generates at a remote service in order to invoke a command on this service. It has the following members:

Table F-9 The class *Process*

Member	Type	Explanation
Command	Command	The request command.
Tag	Tag	The sender and recipient.
Identifier	uint	The process's unique identifier.
State	ProcessStates	The process status.
Data	Object	The input data of the command, if required

F.10 ProcessUpdateNotification

The class *ProcessUpdateNotification* is the return data type of the notification *ProcessUpdate*. It has the following members:

Table F-10 The class *ProcessUpdateNotification*

Member	Type	Explanation
Tag	Tag	The sender and recipient.
Identifier	uint	The process's unique identifier.
State	ProcessStates	The process status (as in Process.State).
Data	Object	The output data of the process, if any

F.11 ConnectionUpdateNotification

The class *ConnectionUpdateNotification* is the return data type of the notification *ConnectionUpdate*. It has the following members:

Table F-11 The class *ConnectionUpdateNotification*

Member	Type	Explanation
Name	string	Name of the device (as in Device.Name)
Connection	ConnectionStates	Connection status (as in Device.Connection).

F.12 StateUpdateNotification

The class *StateUpdateNotification* is the return data type of the notification *StateUpdate*. It has the following members:

Table F-12 The class *StateUpdateNotification*

Member	Type	Explanation
Name	string	Name of the device (as in Device.Name)
State	DeviceStates	Device status (as in Device.State).

F.13 ShutdownNotification

The class *ShutdownNotification* is the return data type of the notification *Shutdown*. It has the following members:

Table F-13 The class *ShutdownNotification*

Member	Type	Explanation
Name	string	Name of the device (as in Device.Name)