The University of
Nottingham

UNITED KINGDOM · CHINA · MALAYSIA

Ryll, Marco (2011) Towards a software framework for reconfigurable and adaptive fixturing systems. PhD thesis, University of Nottingham.

Access from the University of Nottingham repository:

Towards a software framework for reconfigurable and adaptive fixturing systems

Marco Ryll, Dipl.- Inf. (FH)

Thesis submitted to the University of Nottingham for the degree of
Doctor of Philosophy

June 2010

Abstract

There is an ongoing trend towards advanced fixturing systems that can be automatically reconfigured for different workpieces and dynamically adapt the clamping forces during the manufacturing process. However, the increased utilisation of computer technology and sensor feedback currently requires a significant amount of programming effort during the development phase and deployment of such fixtures which impairs their successful industrial realisation.

This research addresses the issue by developing the core concepts of a novel software framework that facilitates the deployment and operation of reconfigurable and adaptive fixturing systems. This includes a new data model for the representation of the fixturing system, using object-oriented modelling techniques. Secondly, a generic methodology for the automatic reconfiguration of fixturing systems has been developed that can be applied to a plethora of different fixture layouts. Thirdly, a flexible communication infrastructure is proposed which supports the platform-independent communication between the various parts of the fixturing system through the adoption of a publish/subscribe approach. The integration of these core knowledge contributions into a software framework significantly reduces the programming effort by providing a ready-to-use infrastructure that can be configured according a given fixture layout.

In order to manage the complexity of the research, a structured research methodology has been followed. Based on an extensive literature review, a number of knowledge gaps have been identified which were the basis for the definition of clear research objectives. A use case analysis has been conducted to identify the requirements of the software framework and several potential middleware technologies have been assessed for the communication infrastructure. This was followed by the development of the three core knowledge contributions. Finally, the research results have been demonstrated and initially verified with a prototype of a reconfigurable fixturing system, indicating that the utilisation of the software framework can eliminate the need for programming, thereby drastically reducing deployment effort and lead time.

List of Publications

Journal Publications

Marco Ryll, Thomas Papastathis and Svetan Ratchev, "Towards an intelligent fixturing system with rapid reconfiguration and part positioning", Journal of Materials Processing Technology, Volume 201, Issues 1-3, pp. 198 – 203, 2008.

Marco Ryll and Svetan Ratchev, "A publish/subscribe approach for a software framework for reconfigurable fixturing systems", International Journal of Advanced Manufacturing Systems, Volume 11, Issue 1, pp. 7-14, 2008.

Book Sections

Marco Ryll and Svetan Ratchev, "Towards a publish/subscribe control architecture for precision assembly with the Data Distribution Service", In: Micro-Assembly Technologies and Applications, ISBN: 978-0-387-77402-2, pp. 359-369, Springer Boston, 2008.

Thomas Papastathis, Marco Ryll, Stuart Bone and Svetan Ratchev, "Development of a reconfigurable fixture for the automated assembly and disassembly of high pressure rotors for Rolls-Royce aero engines", In: Precision Assembly Technologies and Systems, ISBN: 3-642-11597-7, pp. 283-292, Springer Heidelberg, 2010.

Peer Reviewed Conference Papers

Marco Ryll, Thomas Papastathis and Svetan Ratchev, "Towards an intelligent fixturing system with rapid reconfiguration and part positioning", 10th International Conference on Advances in Materials and Processing Technologies (AMPT'07), 7-11 October 2007, Daejeon (South Korea).

Thomas Papastathis, Marco Ryll and Svetan Ratchev, "Rapid reconfiguration and part repositioning with an intelligent fixturing system", ASME International Conference on Manufacturing Science & Engineering (MSEC2007), 15 - 18 October 2007, Atlanta (USA).

Marco Ryll and Svetan Ratchev, "A publish/subscribe approach for a software framework for reconfigurable fixturing systems", International Conference on Agile Manufacturing (ICAM 2008), 16 – 18 July 2008, Kalamazoo (USA).

Marco Ryll and Svetan Ratchev, "Application of the Data Distribution Service for flexible manufacturing automation" 5th International Conference on Control, Automation and Systems (ICCAS'08), 25-27 July 2008, Prague (CR).

Acknowledgements

I would like to thank a number of individuals for their support and contributions without which this work would not have been possible.

First of all I would like to thank my supervisor Svetan Ratchev for having given me the opportunity to accomplish this research in a supportive atmosphere. His technical comments, guidance and encouragement have been invaluable not only for the success of this research, but also made the past 3 1/2 years a challenging, yet enjoyable experience.

I would also like to thank my colleagues in the Precision Manufacturing Centre at the University of Nottingham. Special thanks go to Thomas Papastathis for the outstanding team work and inspiring discussions on the AFFIX research project. Furthermore, I would like to express my appreciation to Colin Astill whose support and helpful hands were instrumental for the practical verification of the research results. I would also like to thank Rachel Watson for proofreading the final thesis.

An dieser Stelle möchte ich mich bei meinen Eltern Renate und Richard Ryll für ihre jahrelange Unterstützung und Liebe bedanken und für das Gefühl immer ein zu Hause zu haben.

Finally, but most importantly, I would like to express my deepest gratitude to my lovely girlfriend Silke Pohl for making me smile every day and for giving me the feeling of being loved. Without her and our cat Monsta I would not be complete.

Table of Contents

List of Figures

List of Tables

List of Appendices

Appendix A: Listings of Module Configuration Files in XML-Format

Appendix B: Datatype Definitions in IDL-format

Appendix C: Source Code for the Device Libraries used in the Prototype Application
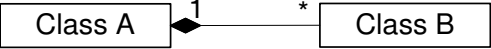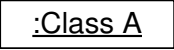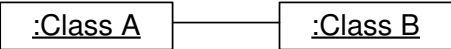
Appendix D: Diagrams for the Force Profiles Followed by the Fixture Modules during the Tests
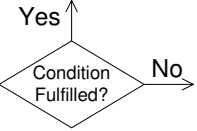
Symbology

Symbol	Name	Description
Unified Modelling Language (UML) Use Case Diagrams		
Actor	Actor	An actor is a role outside the system under study which interacts with the system.
Use Case	Use Case	A use case defines a certain functionality that a system provides to actors.
UML Class Diagrams		
Class A Class A / attributes Class A / methods Class A / attributes / methods	Class	A class is a formal description of a set of objects that have the same structure, constraints and semantics. A class defines attributes to encapsulate the state for its objects. Methods are defined to encapsulate the behaviour of the objects. A class can be depicted by any of the four variations shown on the left, depending on the required level of detail.
Class A ↑ Class B	Inheritance Relationship	Class B inherits from Class A, i.e. Class B is called a child class of Class A. The child class inherits the attributes and methods from its parent class

		and may add more specialised attributes and methods.
Class A —1———*— Class B	Undirected Association	Class A and Class B are associated with each other. That means, instances of each class have access to one another. The numbers specify the multiplicity of the association. This defines how many objects of class A, an object of class B can be associated with (and vice versa). In the picture on the left, Class A can be associated with many objects of Class B whereas Class B can be associated with exactly one object of Class A.
Class A —1———→*— Class B	Directed Association	Class A and Class B are associated with each other. The arrow head indicates that Class A has access to Class B, but Class B has no access to Class A.
Class A ◁—1———*— Class B	Aggregation	An aggregation is an association, semantically expanded by the comment that the participating classes represent a whole-parts relationship (also called "has-

		a"- relationship).
Class A ◆ 1 * Class B	Composition	A composition is a strict form of an aggregation, where the existence of the parts depends on the existence of the whole.

UML Object Diagrams

:Class A	Object	The instantiation of a class is called object. An object is depicted as a square box with the underlined class name, preceded by a colon.
:Class A ── :Class B	Link	An instance of an association is called a link. Thus, while associations are used for relationships between classes, a link exists between two objects. It is depicted by a line between the two objects.

UML Activity Diagrams (Flowcharts)

●	Initial Node	An initial node is represented by a filled circle and marks the entry point to an activity. It has outgoing edges, but no incoming edges.
◉	Final Node	A final node represents the end of an activity. This means, if the final node is reached, the activity terminates. A final node can have incoming edges, but no outgoing edges.

Activity	Activity Node	Represents a tasks to be carried out
Yes No Condition Fulfilled?	Decision Node	Is used to represent decisions.

1. Introduction

1.1.   Background and Motivation

Manufacturing practices are significantly affected by worldwide socio-economic trends such as high labour cost, increased quality expectations by the customers and the global competition. As a consequence, companies are forced to manufacture a great diversity of customised products within short time frames in order to be more competitive. In responding to these market requirements, the current manufacturing needs are characterised by increasing product diversity, shorter product lifecycles and higher quality requirements. To realise these goals, the concepts of automation and reconfigurability are widely acknowledged as the key factors in production and in the past decades a significant amount of research has been conducted in the field of reconfigurable manufacturing systems. The aim is to develop systems that are able to respond quickly to changing product requirements by adapting their equipment structure.

An essential part of almost any manufacturing system is the fixturing solution used to immobilise the workpiece during the process. Fixtures are devices to support, locate and hold a workpiece in a desired position during the manufacturing process. As a result of the direct contact with the workpiece, fixtures play an important role in guaranteeing the quality of the final product in both machining and assembly processes. Potential problems caused by a sub-optimal fixture device include deformation due to over-clamping, slippage and workpiece lift-off as a result of under-clamping, as well as geometric and dimensional deviations of the final product due to inaccurate positioning of the part. In addition to the influence on the workpiece quality and process performance, fixtures are a significant cost factor for the development of a manufacturing system. Indicatively, Bi and Zhang [1] estimated the cost of designing and fabricating fixtures at 10-20% of the total manufacturing system cost, while Consalter and Boehls [2] reported that fixtures and cutting tools may represent up to 25% of the initial investment cost for flexible machining processes. Additionally, Perremans [3] stated that fixturing may consume up to 25% of the total process planning time. Finally, the reconfigurability of the fixturing system determines

to a large extent the degree of flexibility of the overall manufacturing system. However, traditional fixturing and workholding methods are often a key bottleneck in a truly automated and reconfigurable manufacturing environment. Designed for specific products and lacking reactivity, they can be regarded as highly inflexible to changes in product and process specifications. Consalter and Boehls [2] described this problem as a "technological gap" separating fixtures from the advances achieved in the production systems they are a part of. In other words, while modern production systems are increasingly automated, fixtures are lagging behind, thereby becoming true obstacles to further automation and cost reduction. Therefore, Bi et al. [4] concluded: "If flexible manufacturing and assembly systems are to be truly flexible then the fixturing must also be flexible".

Due to the immense impact on the manufacturing process, fixturing has attracted extensive research effort. As revealed by the literature review in chapter 2, a large percentage of the research concentrates on automated fixture design, fixture verification methodologies and optimisation techniques. In addition, a large amount of research has been dedicated towards the development of modular fixtures which can be reconfigured to accommodate a variety of workpieces. However, in general these approaches appear to be restricted to purely mechanical passive devices with limited or no reactive capabilities. Other approaches focus on automated fixture reconfiguration, but these systems lack generality and are restricted to specific hardware setups as a result of using customised software routines. In general, the reactivity of these fixtures is limited to the reconfiguration phase while during the manufacturing process the fixture acts like a passive system with no adaptation of the clamping forces. To further improve the fixturing performance, a few researchers have recently worked towards the development of so-called adaptive fixtures which can actively control the clamping forces in response to external stimuli such as varying machining forces. While it has been shown that these approaches can lead to increased product quality, adaptive fixturing systems currently neglect the problem of reconfigurability. Additionally, the increased use of sensor feedback and computerised equipment leads to new challenges for the reconfiguration process, since the software of these systems must also be adaptable. To summarise, there is a clear trend towards automatically reconfigurable fixtures on one hand, and adaptive fixturing solutions on the other hand. The ultimate goal for the future is

to combine these two trends and develop fixturing systems that are both, automatically reconfigurable and adaptive, thereby becoming a truly flexible part in modern manufacturing systems. However, due to the increased utilisation of computerised components and the resulting need for software programming, the development of such systems can be described as time and cost-intensive, requiring skilled personnel with backgrounds not only in manufacturing but also in computer science. Supporting this statement, Mohamed [5] reported that the software development cost for flexible automation systems is typically 40% or more of the initial investment.

Apart from the mechanical fixture design phase and the physical assembly of the device, a large amount of the development effort must be dedicated to the programming of the system, leading to both, long lead times for the initial system development and long reconfiguration times. This is a significant difference to the development of traditional modular fixtures which typically consist only of passive metal blocks and therefore do not require any software layers. The programming effort includes the development of the software routines for the various sensor and actuator devices, but also the realisation of the overall software architecture of the system. While the former is mostly concerned with the programming of simple libraries for the hardware access, the latter deals with the more complicated integration of the different software modules into a working system. This includes the development of the reconfiguration sequence, the implementation of the force adaptation as well as the communication between the fixture and the rest of the manufacturing system. In current systems which rely on software routines customised to specific fixture hardware, a large part of the programming effort has to be repeated whenever the structure of the fixture is changed.

Figure 1-1 illustrates a typical lifecycle for the development of an automated fixture and indicates where the research presented in this thesis aims to reduce the development effort. The scenario outlines the main phases of the initial development of such a fixturing system until the production phase. Additionally, a reconfiguration scenario is shown where the structure of the existing fixture is changed to respond to new requirements. Examples for such changes would be the addition or removal of clamps, the replacement of a sensor

device with equipment from a different vendor or changes in the structural arrangement of the various fixture components.
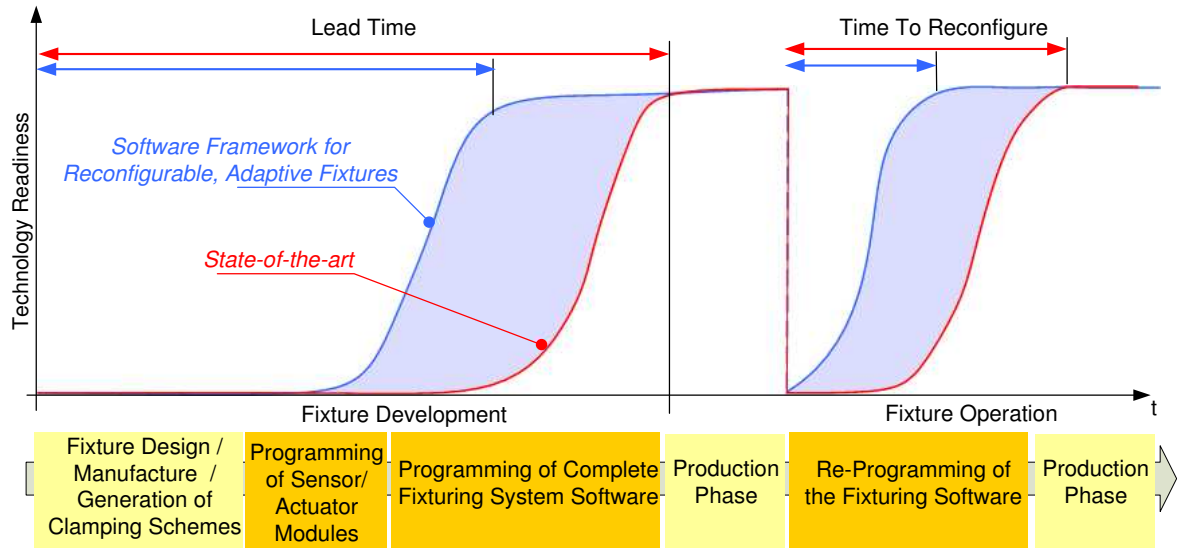
Figure 1-1: Schematic Representation of Potential Time Reductions for the Development of Reconfigurable, Adaptive Fixtures

The red curve indicates the increase of the system readiness during the various development phases without the utilisation of software framework whereas the blue curve depicts the expected improvements, resulting from this research. As can be seen, the software framework significantly reduces the programming efforts in both, the initial development phase and potential reconfiguration scenarios, thereby shortening the lead time and time to reconfigure.

1.2. Research Objectives

The aim of this research is to reduce the programming efforts through the development of a software framework for the operation of reconfigurable and adaptive fixtures. The English Oxford Dictionary defines the term 'framework' as "a structure composed of parts framed together, especially designed for inclosing or supporting anything" [6]. More specifically, in computer science a software framework provides a reusable design and code implementations to clients in order to realise applications of particular domain [7]. Software frameworks can be distinguished from libraries by the so-called "inversion of control". This means, the framework dictates the overall program control flow, whereas libraries are typically passive entities that are called by an application [8]. The software

framework, developed in this research study simplifies the fixture development task by providing a flexible communication infrastructure, a data model to represent the fixture capabilities and a reconfiguration method, applicable to a plethora of different fixture designs. Unlike existing approaches which focus on the automation of the fixture design procedure, the proposed framework reduces the lead times in two aspects:

- The provision of a software framework will eliminate the need to program or re-invent tasks like the automated recognition of equipment and their capabilities, information exchange between devices and the programming of the reconfiguration sequence.

- The platform-independent definition of the library interfaces for common types of devices used in adaptive fixtures such as force sensors, linear actuators and others can lead to an arsenal of ready-to-use software libraries which can be reused in different systems.

As a consequence of this shift from programming effort to configuration effort, engineers will be able to focus on their core competences (e.g. force control strategies, mechanical fixture design, simulation) rather than concentrating on programming and integration issues. Additionally, the fixture development task and the system reconfiguration can be realised by less skilled personnel. In the long term, the configuration of the framework can be automated even further through the utilisation of software tools, thereby further reducing cost, time and effort.

In order to achieve these aims, the following primary research objectives have been identified:

- To define a data model for the representation of the capabilities of reconfigurable and adaptive fixturing systems;

- To formalise a generic reconfiguration methodology that is independent of a specific fixture design and can be applied to a wide range of different fixture layouts;

- To develop an open and flexible communication infrastructure that allows platform-independent device access and communication between the fixturing components.

Additionally, a number of secondary objectives have been identified:

- To identify the user requirements a software framework for the operation of reconfigurable and adaptive fixtures must satisfy;
- To review available communication infrastructure approaches and identify a suitable technology for the adaptation to the fixturing domain;
- To experimentally prove the research results with a novel fixture device that is both, automatically reconfigurable and adaptive

1.3.   Thesis Structure Overview

The remainder of the thesis is structured as follows. Chapter 2 provides an extensive literature review and identifies a number of knowledge gaps. Based on this, chapter 3 defines the research domain by describing the knowledge contributions of this research and outlines the general research approach. Chapters 4, 5 and 6 contain the detailed descriptions of the core concepts of this research, each of them corresponding to one of the identified knowledge contributions. The illustration and verification of the developed software framework with an exemplary laboratory test bed is explained in chapter 7. Finally, the conclusions and the outlook to future work are presented in chapter 8.

2. Literature Review

2.1.   Introduction

Fixtures are commonly regarded as devices to hold and immobilise a workpiece in a desired position during the manufacturing process. As a result of this functionality, fixtures are composed of two main parts, namely clamps and locators. The former are used to exert a certain amount of force against the workpiece, thereby holding it firmly into position. The latter are usually passive elements which limit the degree of freedom of the workpiece and determine a specific position and orientation of the workpiece during the manufacturing process. Additionally, the stability of the system can be increased by the introduction of support elements. Like locators, these are passive elements that prevent the workpiece from moving when the clamps are actuated. The described functional and structural characteristics distinguish fixtures from other workholding devices, such as chucks and vices. These devices typically consist of a number of jaws which are used to hold a workpiece during the manufacturing process. In order to limit the scope of the research, the concepts presented in this thesis are focused on fixtures.

As a result of the significant impact on the manufacturing process fixtures have attracted extensive research effort over the past decades. In particular, a vast amount of work has been focused on the development of fixture design and optimisation methodologies. Also, a number of approaches are available on reconfigurable fixturing systems and recently few researchers have concentrated on the development of active clamping schemes using sensor feedback.

This chapter aims to give an overview on the recent developments in fixturing with a focus on flexible fixturing systems. Section 2.2 classifies the different fixture types and presents relevant research works in the respective categories. Section 2.3 presents existing fixture reconfiguration methodologies and related methods from other areas in manufacturing. Closely related to these are fixture representation concepts and data models that are used as the basis for the various reconfiguration methods. These will be covered in section 2.4. In

section 2.5, relevant communication architectures and middleware technologies will be described. Finally, section 2.6 will identify the current knowledge gaps which are addressed by this research.

2.2.   *Flexible Fixturing Concepts*

The term "Flexible Fixturing" subsumes fixturing systems that present some form of adaptability. This can either be the ability to be reconfigured for various workpieces or to adjust certain parameters of their behaviour like the clamping force. As opposed to flexible fixtures, the term "dedicated fixture" refers to systems that are designed for one particular workpiece and have no means to reconfigure or adapt. Consequently, dedicated fixtures are not the subject of this research work, although they are widely in use for mass production schemes where reconfiguration and adaptability are not considered to be important.

Overviews on the concepts for flexible fixturing have been presented by Shirinzadeh et al. [9], Lin and Du [10], and Bi and Zhang [1]. The main approaches are summarised in Figure 1-1 and include (1) modular fixtures; (2) phase-change fixtures; (3) conformable clamps; (4) programmable fixtures for automated reconfiguration and (5) adaptive fixtures. These technologies are presented in the following sections with a focus on programmable fixtures for automated reconfiguration and adaptive fixtures.
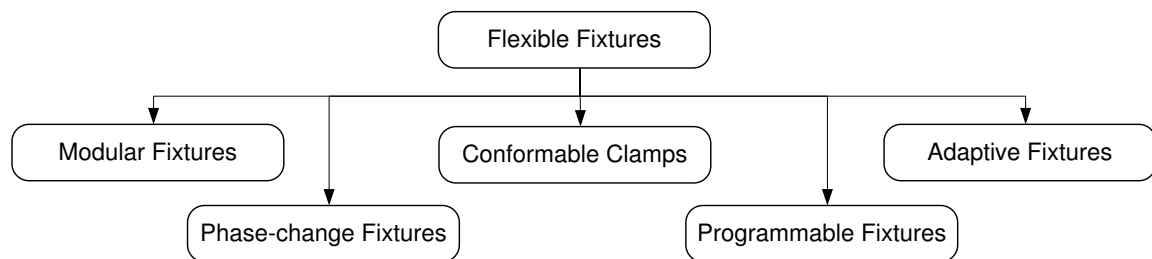
Figure 2-1: Overview on Flexible Fixturing Technologies

2.2.1. Modular Fixtures

Modular fixturing systems consist of a number of standard elements that can be combined in order to accommodate a certain workpiece. These elements include various forms of clamps, locators, supports, base plates and connections. An exhaustive review of modular fixtures can be found in [11] and a large proportion of research has concentrated on either

developing modular fixture equipment or automated design methodologies for this fixture category [12-15].

Sela et al. [16] presented an adjustable modular fixturing system for the assembly of flexible thin walled objects, such as free form metal sheets. The device uses a number of locators and clamps, which can be manually adjusted in all three Cartesian coordinates and locked in position on a T-slot base plate as shown in Figure 2-2.
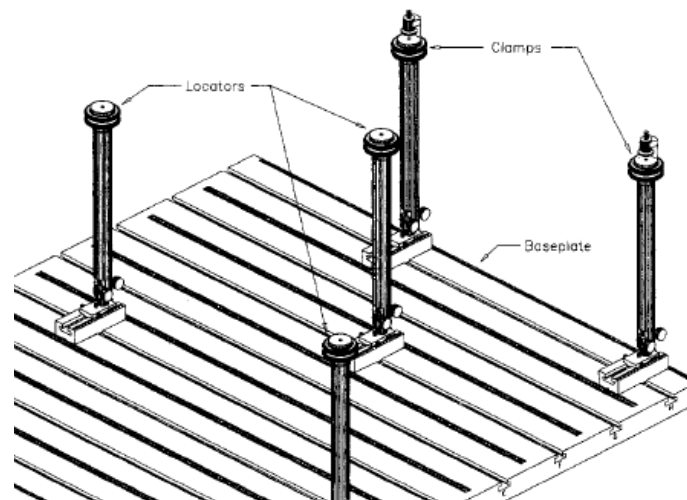
Figure 2-2: Modular Fixture proposed by Sela et al. [16]

Recently, Zheng and Qian [17] addressed the problem of holding workpieces with complex geometries by proposing a modular fixture which can be arranged in 3-D space. The system consists of three base plates with multiple holes on which clamping and locator pins can be mounted. One base plate is fixed and mounted horizontally, while the other two base plates are movable.

The main advantage of a modular fixture is that standard elements can be re-used to build a large variety of different setups. This leads to lower warehouse costs compared to dedicated fixtures and lower maintenance costs as damaged elements can be replaced. Hoffmann has reported that the capital cost of modular fixtures is approximately 25% of an equivalent dedicated fixture [11]. A more detailed analysis of the cost benefits from adopting the modular fixturing solution can be found in [18]. However, the setup of the modular elements leads to larger tolerance stack-up [19]. Also, with the increasing complexity of the

processes and workpiece shapes the planning, design and construction of a modular fixture becomes more difficult and hence time-consuming even for experienced engineers [20]. Finally, these systems are usually designed for manual assembly and are too complex to be automatically assembled and disassembled by robots.

2.2.2. Phase-change Fixtures

Phase-change fixtures exploit the ability of certain materials to change phase from liquid to solid and vice versa. This may be induced by temperature, electric impulses or magnetic fields. Normally, the workpiece is immersed in a container filled with the fixturing material in liquid form. The material solidifies in response to an external influence (catalysts or cooling) and firmly secures the workpiece in the desired position. After the machining process the material is again subjected to catalyst actions and changes its phase back to liquid releasing the workpiece.

Fixtures using phase-changing materials are appropriate for irregular workpieces which are difficult to hold [21] and have been widely used in the aerospace sector for holding turbine blades during the milling process [22]. An example for the application of phase-change fixtures for aerostructures has been presented by Aoyama [23] which utilises electrorheological fluids for the clamping of aerostatic sliders, while Rong et al. [24] exploit the phase-change behaviour of magnetorheological materials. Aoyama and Kakinuma [25] proposed a hybrid phase-change fixture using a low melting point alloy enclosed in a chamber with multiple movable pins to hold thin-walled parts. A heating source triggers the melting of the alloy which results in the repositioning of the locator pins. This system combines phase-change fixtures and locator elements found in modular fixtures. Further examples on phase-change fixtures can be found in [26, 27]. A comprehensive overview on phase-change fixtures has been published by Lee and Sarma [28].

The advantage of this technique is that there is no limitation to the shape or geometry of the workpiece as long as there is sufficient phase-change material to encapsulate it [29]. However, no sufficient solutions have been found to precisely position the workpiece in the

liquid material. Hence, these types of fixtures provide support but no localisation of the part and an additional mechanism must be used to align the workpiece [21].

2.2.3. Conformable Fixtures

Conformable fixtures consist of a number of independently adjustable clamping and locator elements that are arranged in an array to conform to the shape of the workpiece. This results in a more distributed load profile and allows the clamping of parts with complex geometries [30].

Englert and Wright [31] have developed a conformable clamping system for turbine blades. It consists of a hinged octagonal frame with a number of pneumatically controlled plungers. When the plungers have conformed to the shape of the workpiece they are locked with socket screws. Cutkosky et al. [32] have enhanced this approach with plungers that can be actively controlled with a computer programme. Al-Hababeih et al. [33] introduced a hybrid system for the clamping of complex aerospace components which consists of a conformable pin-array and a low-melt alloy whose phase-changing behaviour is exploited to immobilise the pins.

The main disadvantage of conformable fixtures is the limitation of the accessibility of the workpiece due to the large amount of pins. Secondly, reconfiguration times can be considerably longer in the case of passive pins that must be manually adjusted and often a master template workpiece is required to reconfigure the system [32].

2.2.4. Programmable Fixtures

The approaches described in the previous sections are generally based on passive devices with limited or no intelligence in the form of sensor feedback, programmability and automation. Consequently, the reconfiguration process of these systems involves manual adjustments which result in longer reconfiguration times. Programmable fixtures aim to overcome these disadvantages by incorporating sensor-feedback and NC-controlled actuators to automate the reconfiguration process of the fixture. Since this research study

addresses automated fixture reconfiguration, the research results on programmable fixtures are particularly relevant.

As an early example, Tuffentsammer [34] presented two alternative solutions for an automated machining fixture that can be controlled with a CNC interface. The first solution is called the "double revolver" and arranges locators, clamps and supports on servo-controlled turntables as shown on the left in Figure 2-3. The system can be configured for different clamping positions by combining the rotations of the primary and secondary revolvers. In this way, various workpieces of the same product family can be held. The clamping operation is divided into several steps. First, the locators are moved to their pre-programmed positions. When the workpiece is loaded, repositionable hydraulic cylinders which are located over the part provide pre-determined, small clamping forces to hold it in place. After, the supporting elements are set against the workpiece, the full operating hydraulic clamping force is applied to the workpiece and machining can commence.
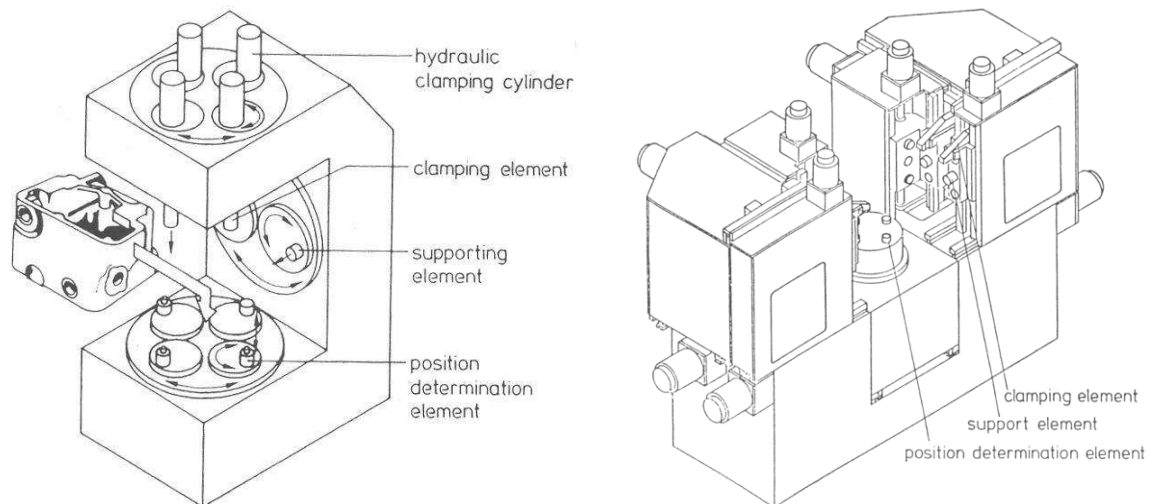
Figure 2-3: Double Revolver and Translational Movement System ([34])

The second system developed by Tuffentsammer is the "translational movement system" illustrated on the right in Figure 2-3. It incorporates repositionable toe clamps and supports on one or two translational axes to secure the part. As can be seen in the picture the system uses a sliding mechanism to adjust the position of the clamps and to be able to hold a wide spectrum of workpieces. Both systems are designed to be integrated in horizontal milling centres and are only targeted to hold bulky parts like castings. Although, both systems

appear to have a certain level of mechanical reconfigurability, it relies on dedicated software routines customised to the specific hardware setups. In other words, if these systems are subject to structural changes like the integration of an additional clamp, the software has to be reprogrammed.

Inspired by Tuffentsammer's double revolver, Lin and Du [35] presented a modular fixture consisting of two types of modules. The first module contains two fingers which can be repositioned with a double revolver mechanism to locate a workpiece precisely. The second module consists of two pneumatic cylinders and is used to clamp the workpiece. Although, these modules can be combined in various ways to secure different shapes of workpieces, no considerations have been mentioned on how this would affect the software architecture of the system. Based on this approach, the same authors proposed an automated flexible fixture for planar objects which can be seen in Figure 2-4 [36].
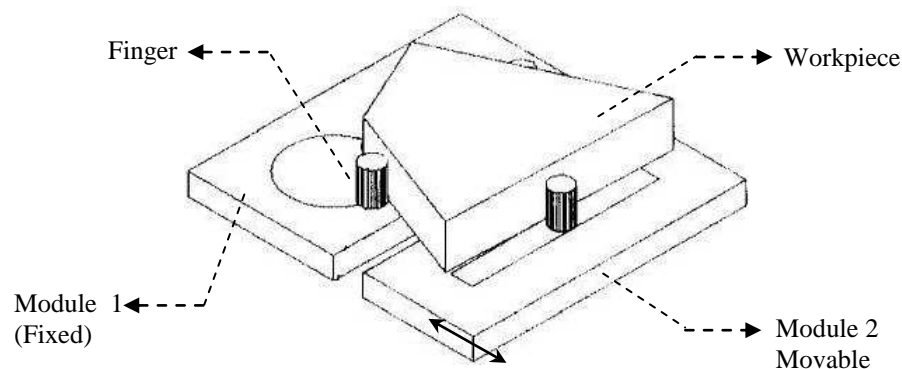
Figure 2-4: Three-fingered Programmable and Reconfigurable Fixture Concept by Du and Lin [36]

The system also consists of two CNC modules and is based on the idea of the minimum number of fingers needed to immobilize a planar workpiece. The locating module is fixed on the base plate and incorporates two fingers mounted on the module according to the double revolver principle. The second module is movable and has only one finger. It is moved towards the workpiece to provide the clamping force. Due to the simple design, it can only hold simple, planar objects and its structure is limited to exactly two modules.

Youcef-Toumi and Buitrago [37] presented a robot-operated modular adaptable fixture. Each module consists of a conformable surface element, a control unit and a locking interface. The conformable surface incorporates a number of pins that can conform to the surface of the workpiece. Therefore, this approach combines the advantages of both

modular fixturing and conformable clamps. The control unit includes the necessary sensors and actuators and the locking interface provides a means of connecting modules with each other or with the base plate. Another robot-operated flexible fixture approach was introduced by Chan et. al. [38, 39]. The system incorporates sensor-integrated horizontal and vertical locators, sensor-integrated V-Blocks, sensor-integrated horizontal and vertical clamps and a hole-type base plate. The sensing scheme is based on Y-guide proximity sensors which can verify if a component is mounted properly in a hole or not. Although, the system is programmable it is mentioned that dedicated software routines are needed for the fixturing process that are customised to the number of elements currently incorporated into the system. Secondly, robot assembly of fixtures has a number of disadvantages like tolerance stack-up. Additionally, the accuracy of the fixture is limited by the accuracy the robot can achieve.

Another automated fixture device was built by Kurz et al. [40, 41] consisting of two hydraulic cylinders, which are connected to the base by revolute joints. The pistons of the two cylinders are also connected with a revolute joint, achieving an accurate 2DOF positioning mechanism. Hence, the device can be incorporated in a fixture for positioning of a workpiece. However, it is not a complete fixturing solution as it lacks clamps for instance. Furthermore, Lu [42] described an automated fixturing system for two-dimensional clamping which has a similar structure as a vice. However, its jaws are fitted with rotatable half-cylinders whose flat surfaces act as clamps. Sensors are used to feed back the position of these clamping surfaces and the vice opening. Additionally, an algorithm is proposed to determine the location of the workpiece in the fixture. The obtained data is then fed back to the NC machine control.

Finally, Chan and Lin [43] developed a CNC controlled modular fixture according to the all-of-a-kind principle. The system comprises only one type of standard multi-finger CNC modules which can provide locating, clamping and supporting functions. Each module consists of four fingers controlled by one motor including two transmission and clutch systems. To clamp a workpiece several of these modules are combined on the platform. The finger positions can be adjusted to hold a variety of similar workpieces. This approach

simplifies the control problem of the system as all modules are of the same type. Also, in terms of flexibility the modules can be arranged in various ways and therefore secure a large number of different workpieces. However, the impact of the reconfiguration on the control software of this fixture has not been taken into consideration and the configuration of these modules cannot be achieved automatically [36]. It is assumed, that this system requires the reprogramming of the control software whenever the fixture setup changes.

2.2.5. Adaptive Fixtures

Adaptive fixtures can be characterised as a comparatively recent development in fixturing and consequently only little research is available on this subject. Similar to the previous category, adaptive fixtures utilise sensor feedback and automation to achieve a certain level of "intelligence" for the fixturing system. However, while programmable fixtures concentrate on reconfigurability, adaptive fixtures aim to improve the fixturing process by actively changing the clamping force in response to external influences during the manufacturing process. In conventional fixtures there is a major discrepancy between constant fixturing forces and dynamic machining forces acting on the workpiece throughout the duration of the process. Most research approaches regard clamping forces to be constant throughout the machining process and hence, clamping forces must restrain the maximum external force that is predicted for the machining process. This leads to over-clamping for the situations when the external load is lower than the maximum. As shown by Tao et al. [44, 45], clamping loads and workpiece deformation can drastically be reduced if the clamping forces are dynamically adapted during the machining process.

As one of the pioneers in adaptive fixturing, Gupta et al. [46] reported on the integration of sensing capabilities in a fixturing system for drilling operations. The device consists of standard vice with two V-blocks, each of them equipped with a dynamometer. With this setup, the system was capable of monitoring the clamping forces, the thrust force and the torque acting on the workpiece during the drilling procedure. Based on the collected data, Gupta et al. was able to define safe and unsafe clamping force regions, depending on the spindle speed and feed rate. However, the system was not able to change the clamping forces during the manufacturing process.

Arguably, the most promising approach has been proposed by Nee et al. [47]. The system consists of six locators which are equipped with piezo-electric force sensors and two clamps. The clamps consist of a DC Servo motor which is coupled with a linear actuator to achieve the clamping force. The position and speed of the servo motor are controlled by a servo driver. The servo creates torque to the actuator which transforms it into a linear movement. The actuator operates with a high reduction worm gear that has a self-locking capability in order to maintain its position and the force. At the end of the actuator a force sensor is embedded to feed back the force that is imparted to the workpiece. For maintaining high accuracy of the motion control, an encoder is attached to the servo motor to feed back the current position and speed to the control unit, forming a local closed-loop. An overview of the dynamic clamp can be seen in Figure 2-5.
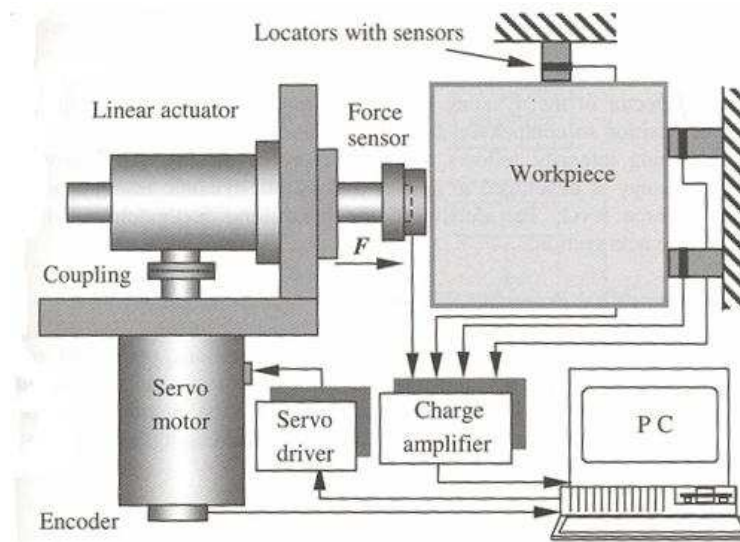
Figure 2-5: Schematics of the Dynamic Clamp ([48])

The whole system is controlled by customised software routines running on a PC. The signals from the sensors go through a charge amplifier and an analogue-to-digital converter (ADC) before entering the PC. The control programme processes the input signals from the sensors according to the clamping force control strategy and sends output signals to the servo controller to adapt the clamping force. Mannan and Sollie [49] have shown that the electro-mechanical clamp is able to adapt the clamping force with an accuracy of +/- 1 N over a range of 700N with a response time of 200ms. However, in the experiments only steps of +10 Newtons have been reported.

Details of the control strategy are described in [50]. It is based on the concept of "control-clamps" and on an optimisation model aiming at the reduction of the clamping and reaction forces in the system. A control clamp of a locator is defined as the clamp with the greatest impact on this locator. In the proposed system, each locator must be assigned with a control clamp. When the reaction forces on a locator fall under a specified lower bound value, the clamping force of its control clamp is increased. Likewise, if the reaction forces of a locator exceed a certain upper threshold, the software commands the related control clamp to minimize its clamping force. A prototype of the system has been successfully tested for a slot-milling operation of a simple-shaped workpiece and for a finish pocketing operation on a box-shaped, thin-walled workpiece. The results show that dynamic force adaptation improves the workpiece stability and that clamping forces can be decreased significantly compared to a conventional system with constant forces. This leads to less deformation and higher accuracy of the finished workpiece. Workpiece deformation could be decreased by 20% in one experiment [50]. A variation of the above system has been reported in [51] which uses hydraulic instead of electro-mechanical actuators. Inspired by the results of the aforementioned system, Rashid and Nicolescu [52] have recently applied the approach of adaptive fixturing to actively dampen vibration in a palletised workholding device. The proposed system consists of a rectangular frame with integrated force sensors and piezo-electric actuators, fast enough to counter-act vibration.

2.2.6. Discussion

The review of the available literature on fixturing shows an ongoing trend towards automation. This is particularly reflected by the recent developments in the fields of programmable fixtures and adaptive fixtures. However, the reviewed programmable fixtures are based on dedicated software routines, customised to specific hardware setups. Hence, these systems offer the capability of mechanical reconfiguration, but the adaptation of the underlying software has been widely neglected. This becomes even more important with the advent of adaptive fixtures. The presented examples for adaptive fixtures do not provide the ability to be reconfigured for a variety of workpieces. Hence, the ultimate objective in the future is to combine the benefits of programmable and adaptive fixtures. However, it appears that the ever-growing integration of sensor-feedback and automated

actuator units requires a greater level of flexibility of the fixturing software than current systems have.

2.3.   Reconfiguration Methodologies

While the previous section has focused on available mechanical concepts used for flexible fixturing, this part aims at exploring the available literature on methods for the fixture reconfiguration. Additionally, reconfiguration approaches from other areas in manufacturing are reviewed.

2.3.1. Fixture Reconfiguration Methods

A large percentile of the methodologies addressing the reconfigurability of fixturing systems target the design and planning process of modular fixturing systems. Traditionally, this has been a task relying on the experience and intuition of skilled engineers. To automate this time-consuming and hence cost-intensive process, researchers have applied a number of techniques from computer science and artificial intelligence. The research activities can be categorised in (1) Fixture Design and Verification Methodologies and (2) Fixture Optimisation Methodologies.

2.3.1.1.   Fixture Design and Verification Methodologies

Fixture design methodologies aim to automate the decisions made in the fixture development process. This involves activities like describing the requirements and the constraints, the selection of appropriate fixture elements and the positioning of the clamping points. Fixture verification is closely related to this and tries to evaluate a certain fixture configuration according to the design criteria, such as stability, workpiece deformation and the minimisation of the clamping forces. The information from the verification can be fed back to the system in order to generate a better design.

Expert systems have commonly been used which expect the description of the workpiece and the process as inputs and generate a fixture design by interpreting a set of rules. Nnaji and Lyu [53] presented such a system for the automatic layout of flexible fixture models on a CAD/CAM system. The proposed rules are based on the 3-2-1 locating scheme which is commonly used for prismatic workpieces. According to this, three locators are placed

against the largest planar surface, two locators are placed on the surface perpendicular to the previous plane which has the longest edge and the remaining locator is placed on a mutually orthogonal plane [21]. The methodology is implemented in the logic-based programming language PROLOG and was demonstrated for the surface milling of polyhedral workpieces. As part of the IDEFIX project Perremans [3] developed an expert system for the design and planning of modular fixtures for the machining of prismatic workpieces. The inputs are the faces on which positioning, clamping and supporting should be done and the system automatically generates the necessary assembly of modular fixturing elements. Gaoliang et al. [54] have proposed a hybrid method using rule-base reasoning and fuzzy logic to capture the geometric constraints of modular fixtures in a virtual reality system to automate fixture design.

However, there are some disadvantages in the use of expert systems. Firstly, the complexity of the design process makes it difficult to formulate rules. Secondly, even experienced experts struggle to explain their knowledge in simple rules. To overcome the disadvantages of rule-based systems, some researchers have applied case-based reasoning (CBR) techniques. In CBR knowledge is stored as experience in the form of cases. When the system is confronted with a new case it retrieves the most similar case from its case base and modifies it to meet the new requirements, thereby extending the data base by a new case. Sun and Chen [55] proposed such a system. In order to find the similar cases the authors introduced an index method for the features of a fixture. However, the proposed index considers only workpiece geometry and is quite superficial. A similar system was proposed by Li et. al. [56] which is based on a hierarchical decomposition of the fixture structure into layers of function units, components and elements. It is mentioned that this layered approach facilitates fixture reconfiguration, because the components and elements can be replaced in response to changing requirements. When confronted with a new case the system retrieves the most similar case from its knowledge base by calculating the "degree of similarity". However, to accomplish this calculation, the system relies on weigh factors whose values appear to be rather arbitrary. Recently, the same research group has enhanced this concept for a welding fixture design system [57].

Finally, a number of geometry-oriented approaches have been published where design information is mainly extracted from CAD systems. Based on the shape of the workpiece, appropriate clamping, support and locating elements are determined. Trappey et al. proposed a method that projected the geometry of a workpiece to find a feasible fixture configuration based on the 3-2-1 locating principle [58]. With a focus on design verification, Kang et al. [59, 60] presented a computer-aided fixture design verification (CAFDV) framework which is based on geometric and kinematic models to confirm locating accuracy, fixturing stability and the determination of the minimum clamping force. Wu et. al. [61, 62] presented a method addressing the geometric analysis and verification for the planning of modular fixturing systems. It is capable of determining the fixturing surfaces and locating points to provide suitable geometric constraints. The approach has been applied to various types of workpiece in 2D and 3D.

2.3.1.2.   Fixture Optimisation Methodologies

The high complexity of fixture design implies that in most cases there is a large number of possible solutions. Optimization techniques are used to identify the best solution in respect to a particular design objective. Hence, these systems require an objective function and search for the best solution by varying certain input parameters.

King and Hutter [63] proposed a theoretical approach that utilised kinematic, force and robotic grasp analysis to generate optimal fixturing location points that secure the workpieces ideally with respect to maximum stiffness, resistance to slip and stability. Menassa and DeVries [64] incorporated the Finite Element Method (FEM) to analyse the expected deflections of the workpiece. On this basis their system determined the ideal positions of the fixture supports in order to minimise workpiece deflection. The problem of these approaches is that they require complex and time-consuming computations. This is why these models were restricted to simple prismatic workpieces [65].

A number of researchers applied evolutionary programming techniques such as genetic algorithms (GA) and artificial neural networks to find the optimal fixture configuration for a set of requirements. Genetic algorithms are based on Darwin's Survival-of-the-fittest

theory, which states that only the most suited individuals in a population are likely to survive and generate offsprings. A genetic algorithm emulates the evolution theory by changing parameters in the system and measuring the "fitness" of the resulting system against a "fitness-function". The most promising solutions are chosen to generate offsprings and in this way the optimisation problem is solved. Wu and Chan [65] used this technique to find the most statically stable fixture configuration from a large number of candidates. Unlike earlier approaches this method is not limited to specific workpiece geometries and is free from frictionless assumptions. Krishnakumar et al. [66, 67] used a similar approach to optimise the fixture layout and clamping force intensity. Their objective function is the minimisation of the workpiece deformation during the cutting process. Other systems based on genetic algorithms have been developed by Vallapuzha et. al. [68], Kaya [69] and Aoyama et al. [70].

2.3.2. Reconfiguration Methods for Manufacturing Systems

The previous section indicates that the methodologies for fixture reconfiguration are mainly addressing the fixture design and optimisation phase. However, frameworks focussing on the reconfiguration issues that occur during the operation of the fixture are widely missing. The reason for this is that until recently fixtures were widely treated as passive mechanical elements without any intelligence. At the same time, the reported examples for reconfigurable fixtures (see section 2.2.4) lack a general methodology and are restricted to specific fixture layouts. The ongoing trend towards intelligent adaptive fixtures leads to a demand for generic methods that focus on realising the reconfiguration during the fixture operation. For this reason, it makes sense to review reconfiguration methods applied in other manufacturing areas which have progressed further on the transition to fully automated systems. In particular, there are a number of approaches available for the automated reconfiguration of assembly systems.

A widely acknowledged concept is the holonic approach which is a distributed control paradigm, based on autonomous and cooperative entities called "holons"[71]. As a key feature a holon can be part of another holon, which builds up an open-ended hierarchy, called the "holarchy" [72]. Further details about holonic manufacturing systems can be

found in [73]. Sugi and Maeda [74] presented a holonic assembly system comprising three manipulators, one belt-conveyor and two warehouses. The system consists of two layers, an upper management layer which is responsible for the task planning and a lower execution layer with holons corresponding to the assembly devices. Holons of the upper layer issue orders to those of the lower layer, while entities on the same level negotiate with each other who executes this task. Thus, for the upper layer it is transparent how a particular task is accomplished and therefore the assembly sequence can be generated dynamically according to the actual setup. Leitao and Restivo [75, 76] proposed a holonic architecture for agile and adaptive manufacturing control, called ADACOR. The system is based on a set of operational holons with self-organizing and learning capabilities. Additionally, a supervisor holon is introduced which coordinates the subordinate entities and allows for global optimisation of the process. Other holonic approaches can be found in [77-79].

Closely related to holonic approaches are agent-oriented systems. According to Ferber, an agent is defined as a physical or virtual entity which is capable of acting autonomously in an environment, can communicate and has its own goals which it tries to achieve [80]. A multi-agent system is characterised by the cooperation, communication and even competition between multiple agents. Due to the distributed nature of these systems, multi-agent approaches can react flexibly to changes and have therefore been introduced for reconfigurable manufacturing systems. In fact, most of the holonic manufacturing systems described in the previous section have been implemented as multi-agent systems. Tang and Wong proposed a flexible assembly cell based on several reactive agents [81]. Reactive agents do not maintain the status of their environment. They rather react to stimuli. Hence, they are particularly effective for systems with limited memory resources. The proposed system incorporates material-handling agents that control the conveyor line of the cell and robot agents representing the manipulators. Additionally, a supervisory agent coordinates the actions of the subordinate agents. This structure is also referred to as the subsumption architecture. Each of the agents acts autonomously according to its own "local" goals. The authors introduced a coordination model which allows a team of self-interested, reactive agents to achieve a global goal by the means of exchanging messages. Similar to the

holonic approaches, the system can flexibly adapt the assembly sequence when the physical resources change.

2.3.3. Discussion

The previous sections show that a significant amount of research is available on fixture reconfiguration methods. However, these methods concentrate on automating fixture design and optimisation. In some cases, the presented methodologies only apply to a small number of fixtures, lacking generality. Generally, the presented research is focused on the design issues of traditional modular fixtures composed of passive metal blocks. It appears that the trend towards adaptive fixtures, incorporating sensor and actuator devices, requires new methodologies to address the reconfiguration issues during the operation of the fixture on the shop floor. Other areas in manufacturing with a higher level of automation show that there is a number of potential approaches to achieve dynamic reconfiguration. Some researchers have proposed holonic architectures and agent-based systems. However, these approaches rely heavily on time-consuming negotiation algorithms. Although negotiation between agents is an adequate method for assembly lines where events typically happen in the ranges of seconds, it does not appear to be the right solution for fixturing systems which need to react much faster in order to adapt the clamping force. On the other hand, the proposed hierarchic control methods and the delegation of commands from one layer to another are regarded as key technologies for the development of a fixture reconfiguration methodology.

2.4. Data Models and Representation Concepts

A fundamental part of any framework for a reconfigurable system is a data model which is able to represent common aspects of the underlying systems and model the relationships between the various entities. This section aims at reviewing existing models for both fixturing systems and reconfigurable manufacturing systems.

2.4.1. Fixture Representation Concepts

A number of researchers have tried to conceptualise modular fixturing systems as a basis for the previously reviewed fixture reconfiguration methodologies. Perremans [3] proposed

a feature-based data model which describes a number of modular fixture elements in terms of geometry, type of contact, tolerances and other aspects. His model is based on three feature types, namely (1) "Contact Features"; (2) "Assembly Features"; and (3) "Tightening Features". The first feature type represents elements that are in contact with the workpiece such as different forms of locators. Assembly features are used to combine various types of modular elements, while the third feature type is used to tighten an assembly of modular elements. The author has expanded this model to a catalogue consisting of 26 contact features types, 10 assembly features and 7 tightening features. The model showed acceptable results for two commercially available modular fixturing systems (Norelem[@] and Bluco[@]), however the concept is limited to passive elements and is therefore inadequate for the representation of reconfigurable, adaptive fixtures. Other feature-based concepts have been proposed by Nee et al. [82], Shirinzadeh [83] and Jeng and Gill [84]. The latter defines a hierarchy of fixturing elements in terms of high-level, functional entities such as base plates, clamps, locators and supports as shown in Figure 2-6.
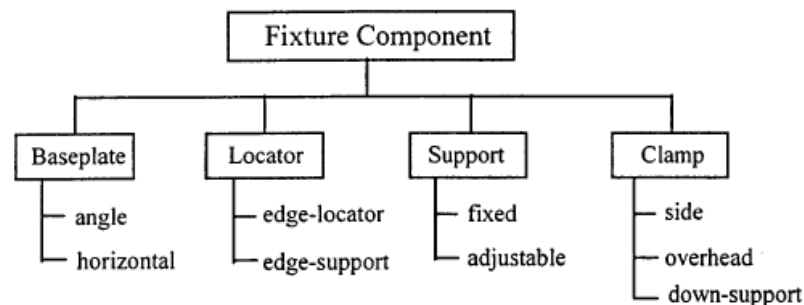
Figure 2-6: Hierarchical Classification of Fixture Components [84]

For each component type a data structure is proposed that contains some qualitative attributes of the functional properties, geometrical constraints and the constituent components. The feature-based approach of Subrahmanyam [85, 86] also distinguishes clamping, locating and support features, however these definitions refer to the workpiece surfaces rather than the fixturing system itself.

The hierarchic modelling approach can also be found in Li et al. [56] who have decomposed the fixturing structure into several functional units such as top-clamping, side-clamping or bottom locating. Each functional unit is further decomposed of so-called functional components which are in contact with the workpiece and assistant components.

The bottom of this tree-like structure consists of function elements and assistant elements which are the basic building blocks for the component layer. This layered hierarchy is the key for the reconfigurability of the system, because the entities in each layer can be replaced in response to different requirements without affecting other layers. Similarly, Wang and Rong [57] utilised multi-level data abstraction to generate a hierarchic model of the fixture structure. In contrast to the previously mentioned approaches this system enhances the hierarchic structuring idea with an object-oriented model to represent the relationships between the various entities in the system. Figure 2-7 illustrates an example for the capturing of fixture design information, used in this system.
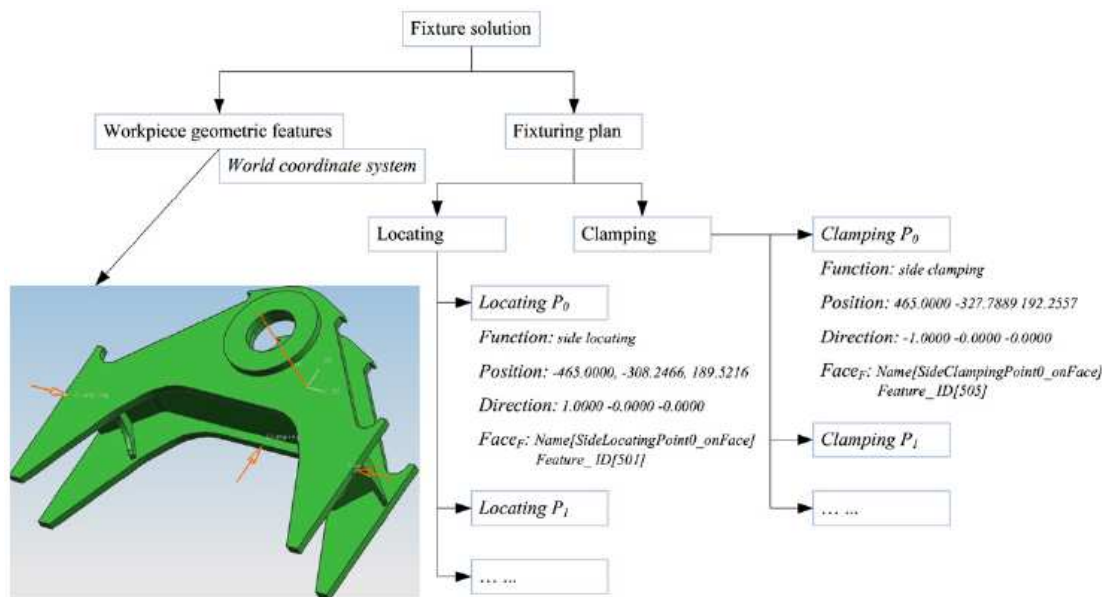
Figure 2-7: Example for Capturing Fixture Design Information as Objects [57]

An advantage of using objects is that they can be easily described with computer-readable, platform-independent languages such as XML. In this context, Liquing and Kumar [87] utilised XML and object-orientation in a case-based reasoning system for automated fixture design of modular fixtures. The description of the cases consists of the part representation, the fixture design representation and the setup representation which links the design information with various workpieces. In this conceptualisation, a modular fixture is a subclass of a fixture which is composed of multiple fixture elements, namely clamps, locators, supports, base plates and accessory equipment. The implementation of this system as a client-server application using Java Remote Method Invocation (RMI) has been described in [88].

Recently, Hunter et al. [89, 90] presented a functional approach for the formalisation of fixture design information as a part of a design methodology for modular fixtures. Object-oriented modelling techniques are used which are represented with the Unified Modelling Language (UML). The main entities of the model are non-functional fixture requirements like cost and functional requirements such as clamping forces and locating points. Additionally, the model contains design rules and so-called fixture functional elements in terms of clamps, locators and support elements. The methodology relates the requirements with suitable fixture functional elements which are mapped to specific commercially available components.

2.4.2. Representation Models for Reconfigurable Manufacturing Systems

Similar to the reconfiguration methods, the majority of the fixture representation models appears to concentrate only on passive modular fixtures. Additionally, these models focus on the issues of fixture design and cannot be directly applied to the operation of a reconfigurable, adaptive fixturing system. This section aims at highlighting some related research on other fields in manufacturing where researchers have tried to generate data models addressing the need of automated reconfiguration.

A number of approaches have been proposed for the formalisation of the process capabilities of equipment modules in automated assembly systems. Based on knowledge-intensive Petri nets, Zha et al. [91] generated a function–behavior–structure model for the automated design of such systems. According to Lohse et al. [92], the behaviour of a module is an objective description of how the module reacts to stimuli and transforms inputs to outputs. The functions are a subjective abstraction of the behaviour and express the capabilities of a module, based on the purpose or the intention of the designer. The structure describes the physical model of the modules with objects, attributes and relations. Based on this, Lohse [93] has described an ontology framework which is able to capture the capabilities and requirements of modular assembly systems. Other related research was reported in Meijer et al. [94], Zhang et al. [95] and Prabhakar and Goel [96].

Due to its flexibility, object-orientation has been widely used by a number of researchers. Kovács et al. [97] commented on the merits of object-oriented methods for the reconfigurability of the control software, both during the design phase and the low-level management of hardware changes. Schäfer and López [98] proposed an object-oriented model for the control of flexible manufacturing systems with robotic manipulators. The model defines a number of equipment resources and their capabilities, as well as control parameters and coordinate frames. Each resource is defined by two classes, one resource class and corresponding control class. Further, Bruccoleri et al. [99] reported on an object-oriented high-level control structure for the real-time error recovery in reconfigurable assembly systems. In a related article [100], the same authors described a reconfigurable system for robotised manufacturing cells. The underlying model for this approach is based on an object hierarchy as illustrated in Figure 2-8.
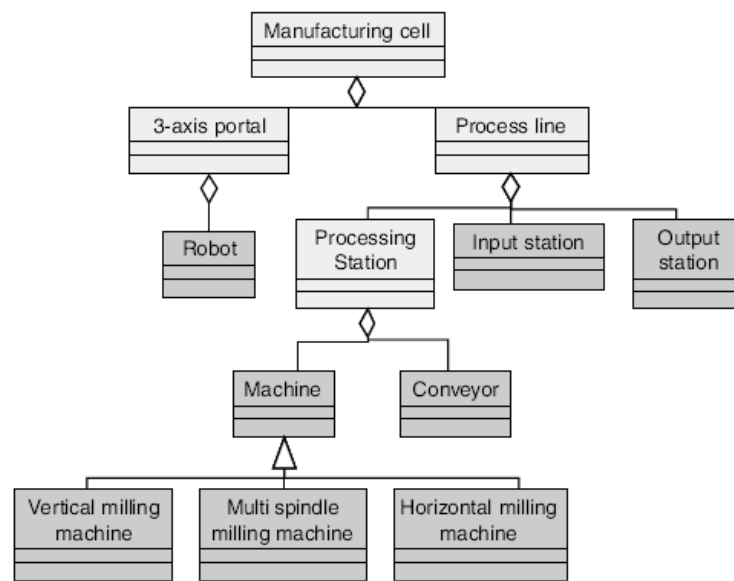
Figure 2-8: Class Diagram for the Control System of a Robotised Manufacturing Cell [100]

As a further extension of the object-oriented paradigm, a few researchers have exploited the benefits of object-oriented design patterns in their models. The concept of design patterns goes back to Alexander's "The Timeless Way of Building" [101] which describes them as generic solutions to recurring problems and therefore they allow the reuse of expertise acquired in previous designs. Later, design patterns have been introduced to software engineering for the reuse of generic object structures in the design of software applications [102, 103]. Gamma et al. [104] formalised the description of patterns and published a

standard catalogue of 23 design patterns that are widely considered as the standard work in this field.

Thiry et al. [105] applied a number of design patterns from Gamma's catalogue to the field of robot control. In more detail, the "Command" pattern was adopted to allow dynamic upgrade of a system with new behaviour. An illustration of the class structure is provided in Figure 2-9. A system, in this case a legged robot, can be attached with a variable number of behaviours, each of them modelled as own classes. To invoke a certain behaviour, the generic function "Do" is called on the System with the identifier of the behaviour and an optional parameter list. The request is then delegated to "Do"-function of the corresponding behaviour object.
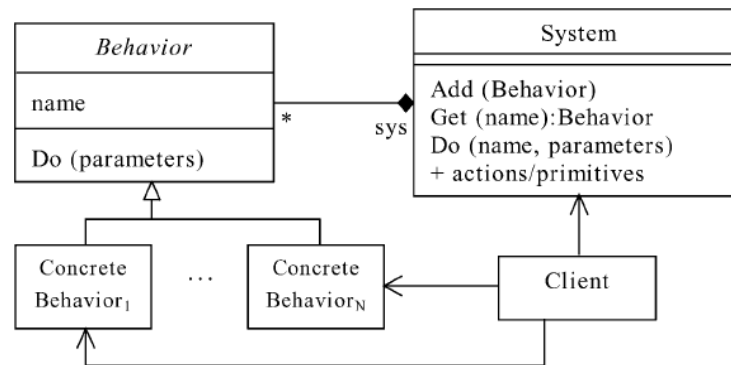
Figure 2-9: Class Structure of the Polymorphic Behaviour Pattern [105]

Recently, Soundararajan and Brennan [106, 107] have adapted the "Proxy" design pattern for a distributed real-time control system. The pattern proved particularly useful in distributed systems where clients invoke server requests on a local representative who is responsible for the information exchange and makes the rest of the application independent from the implementation details of the server. Further examples of the use of design patterns can be found in Pont and Banner [108] (embedded systems control), Sanz and Zalewski [109] and Buschmann et al. [110].

2.4.3. Discussion

The presented literature shows that a number of researchers have tried to generate data models for the representation of fixturing systems. Most of the models concentrate on capturing the structural characteristics of a fixturing system. While this information is

crucial in the mechanical design phase, these models lack the means to represent the behavioural information of the fixture which is required during the operation of the device. Moreover, the presented models are limited to traditional modular fixtures composed of mechanical blocks and do not provide mechanisms to represent more intelligent modules, used in today's adaptive and programmable fixtures. Nonetheless, these approaches still indicate that hierarchical modelling approaches proved useful in conceptualising a system. In particular, a number of researchers have successfully applied object-orientation techniques and highlighted the merits of using platform-independent standards like XML and UML to support their models. Research effort towards the automation and dynamic reconfiguration of systems in other manufacturing areas shows a clear trend towards the modelling of system capabilities using object-oriented techniques. However, the existing models do not address the specifics of the fixturing domain and must therefore be adapted accordingly. In general, it was observed that although a lot of models are based on object-orientation, they appear to be limited to basic inheritance relationships while not mentioning the use of method delegation. The importance of delegation for the reconfigurability of systems has been emphasised by Gamma et al. [104]. In this context, several examples have been presented which adopt object-oriented design patterns which are usually characterised by the heavy use of method delegation.

2.5. Communication Infrastructures for Information Exchange

The trend towards adaptive fixturing systems, composed of a variety of sensor and actuator modules will lead to an increased information exchange between the fixture components. Additionally, the fixture needs to communicate with other parts of the manufacturing environment, such as Human Machine Interfaces (HMI), Machine Control Systems or Resource Planning Systems. Consequently, there is a need for a communication infrastructure that allows information exchange in a heterogeneous network environment consisting of different hardware architectures, operating systems and communication requirements. Although, traditional field bus technologies provide robust communication of cyclic process data, the existing technologies are notoriously hard to integrate with other networks. At the same time, ethernet has emerged as the most widely used communication

technology in other domains, such as e-commerce. For this reason, Neumann [111] states that in recent years, there is a significant trend towards Ethernet-based communication systems in the manufacturing arena.

To support Ethernet-based data exchange, a number of middleware approaches are available which rely on a variety of different communication paradigms. The term middleware refers to an additional software layer between the application software and the operating system, shielding the former from low-level tasks for the data distribution. The fundamental communication paradigms for these middleware solutions can be classified in multiple ways, depending on which aspect is of interest. Hurwitz [112] distinguishes between Message-Oriented Middleware (MOM), Remote Procedure Call (RPC) and Object Request Broker (ORB) systems. Recently, Amoretti and Reggiani [113] proposed a similar classification and added service-oriented architectures (SOA). In their categorisation, the term Distributed Object Architecture (DOA) is used for ORB-approaches which is subsequently adopted. The following sections provide a brief overview on the most significant architecture paradigms and highlights examples for their use in the manufacturing domain. Additionally, a further category is introduced, namely data-centric architecture. Remote Procedure Call can be regarded as a forerunner model of the Distributed Object Architecture and is therefore not described in detail.

2.5.1. Distributed Object Architecture

Distributed Object Architecture systems allow clients to invoke remote methods of server objects in the same way as local function calls. Based on the formal description of the method interface, a client can instantiate a proxy of an object on which it calls a certain method. Internally, the request is forwarded to the actual server-object which implements the method.

An example for this category is the Java Remote Method Invocation (RMI) standard. Mervyn et al. [88] utilised RMI for the implementation of an internet-enabled fixture design system. However, the Common Object Request Broker Architecture (CORBA) [114], developed by the Object Management Group has arguably attracted most attention

over the years. CORBA has been specifically designed for distributed systems in heterogeneous environments and enables applications to communicate with each other regardless of the operating system, programming language and computer architecture. This is achieved through the definition of the communication interfaces in a platform-independent format called Interface Definition Language (IDL) [115]. Based on these definitions the source code for the data transfer is automatically generated and can be linked with the application source code. In order to communicate, an application needs to instantiate a local object which represents the remote application. When the functions provided by its interface are called, the middleware internally cares for all data format conversions across different platforms and routes the request to the remote application through a so-called Object Request Broker. The latter acts as a mediator, routing requests and responses between the distributed objects. As a result of this architecture, it makes no difference for the software developer if an application is distributed over a large network or if the communicating peers run on the same computer, or even as parts of the same process. However, a disadvantage is that the ORB can become a single-point-of-failure and a potential performance bottleneck. Furthermore, as CORBA is based on the client/server principle, it creates tight couplings between the interacting applications and therefore makes the implementation of decoupled many-to-many communication comparatively difficult. To address the needs of real-time applications, a special CORBA profile has been released as a standard, namely RT CORBA [116]. This standard shares most characteristics with the full CORBA profile like the client/server principle or platform-independence, but extends it with features to have better control over timing and resource usage to allow deterministic data exchange. Key to this is the Quality-of-Service (QoS) model. The term Quality-of-Service (QoS) refers to a general concept used to specify and control the behaviour of the communication service. It offers the advantage that the application developer only needs to indicate 'what' is required rather than 'how' this behaviour is achieved [117]. In particular, QoS provides the ability to manage the use of resources like network bandwidth or memory as well as reliability, timeliness and persistence of the data transfer. Examples for CORBA-based systems in manufacturing have been reported by Shin et al. [118], Sanz [119] and Haber et al. [120].

2.5.2. Data-centric Architecture

According to Joshi [121] the data model is the most stable part in a system of loosely-coupled applications and is therefore less likely to change over time than the method interfaces. Following this observation, data-centric architecture approaches aim to decrease the interdependencies in distributed applications by exposing the data model, instead of the method interfaces. Based on the platform-independent definition of the data model, the source code for sending and receiving data can be generated automatically for the various target platforms. Secondly, data-centric architecture systems typically follow the publish/subscribe communication paradigm. According to this model, the applications do not communicate directly with each other. Instead, data is shared among the applications by the means of topics. Processes that want to send data become "publishers" for a topic while other applications can subscribe for contents of a topic if they require data from it. Consequently, the data topics form a so-called "global data space" that is accessible to all interested applications [117]. Figure 2-10 illustrates the global data space with three topics and five participants. The arrow directions indicate if an application is a publisher or a subscriber for a certain topic. Specifically, an ingoing arrow marks the application as a subscriber while an outgoing arrow declares it as a publisher. As a result of the publish/subscribe concept, communication is decoupled through the topics and flexible many-to-many communication between a large number of participants is supported.
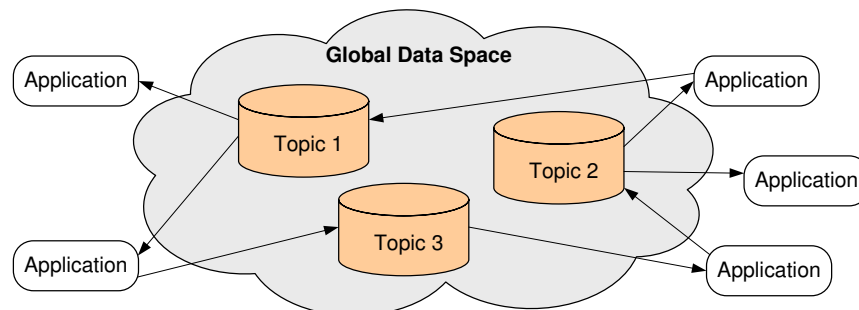
Figure 2-10: Overview of the Publish/Subscribe Concept

The Data Distribution Service [117] is an example for a platform-independent data-centric middleware standard, released by the Object Management Group. Like CORBA it utilises the Interface Definition Language as a basis for the automatic generation of communication source code for a large variety of operating systems, programming languages and computer architectures. The middleware is able to automatically detect new participants in the system

and establishes connections between the publishers and subscribers for a matching topic. Additionally, the standard is specifically designed for the needs of real-time applications and provides a rich set of Quality-of-Service (QoS) parameters to configure the communication behaviour for each topic according to the application requirements. There are a number of commercially available DDS solutions on the market, including those of Real-time Innovations, Inc. and PrismTech. Additionally, the open-source community provides a free version of the standard, called OpenDDS [122]. The aerospace and defence industry is widely using the DDS standard for intelligent weapon systems and flight control. Other industrial examples have been reported for flight simulation systems and traffic control systems [123]. Schneider Electric are using DDS-based communication bus for their range of Programmable Logic Controllers (PLC) [111, 124], while ALSTOM Schilling Robotics have developed a remotely operated robotic system, used for underwater installation and repair works [123].

2.5.3. Service-oriented Architecture

The service-oriented architecture (SOA) paradigm aims at minimising the interdependencies between the communicating software entities by defining independent "services" which can be accessed through a stateless request/reply scheme [113]. Thus, the use of SOA results in an environment of loosely-coupled service providers and service consumers. Key to the concept is the unambiguous, computer-interpretable description of the service interfaces and their location. SOA systems have mainly been implemented using Web Services. This technology uses the Web Service Description Language (WSDL) to define the interface of a service for its consumers. However, there is still no standard for the distributed publication and discovery of Web Services. The messages are typically transferred in a standardised protocol, such as Simple Object Access Protocol (SOAP). In this case, the use of service-oriented architectures introduces a significant communication overhead as a result of the message interpretation. Veiga et al. [125] compared two different SOA platforms for the integration of equipment in robotised assembly cells and concluded that the use of such frameworks can reduce the complexity of the development of modern manufacturing systems, since engineers can concentrate on their expertise (machine vision, force control, mechanical design) instead of dealing with device

interconnection and cross-platform communication problems. Other applications of SOA platforms in the manufacturing area have been reported by Ahn et al. [126], Estrem [127] and Ha et al. [128].

2.5.4. Message-oriented Architecture

Similar to the previous category, Message-Oriented Middleware (MOM) aims at the decoupling of applications. The difference is the use of a message broker which acts as a mediator, passing messages to and from the distributed applications. This allows the decoupled and asynchronous information exchange between a large number of applications.

The Java Messaging Service (JMS) is a message-oriented middleware that provides an Application Programming Interface (API) for the sending and receiving of messages in JAVA programs [129]. It has become the de facto industry standard for JAVA-based messaging [130] and is supported by most commercially available MOM platforms [131]. The standard provides two general mechanisms for communication, referred to as message domains. The point-to-point domain is used for the synchronous communication between possibly multiple senders and exactly one receiver. Additionally, JMS allows asynchronous many-to-many communication via data topics according to the previously described publish/subscribe paradigm. Industrial applications of JMS have been reported in Urdaneta et al. [132] and Mervyn et al. [133].

While JMS is an application-neutral middleware standard, another MOM system exists that is tailored to the manufacturing domain. As a result of the National Electronics Manufacturing Initiative (NEMI) for a plug & produce environment in the electronics industry [134], Computer-Aided Manufacturing using XML (CAMX) has been proposed as a message-oriented middleware which defines an event-based conversational framework based on exchange of standardised XML messages [135]. These messages are distributed via a central entity, the message broker, whose general architecture was specified in the IPC2501 standard [136]. The XML messages are exchanged according to the publish/subscribe paradigm, allowing many-to-many real-time communication between an arbitrary number of processes. Additionally, the framework aims at supporting platform-

and vendor-independent communication of a wide range of equipment. For this reason, the syntax and semantics of a large number of messages associated to manufacturing events on the shop floor are specified in the IPC-254x standards [137-139]. As a difference to the other middleware candidates, CAMX is not application-neutral, since it is designed for the assembly of printed circuit boards. Like DDS, CAMX provides a number of Quality-of-Service parameters to adjust the behaviour of the communication [140, 141].

2.5.5. Discussion

The previous sections have shown that there is a myriad of different communication platforms available that allow efficient data exchange in distributed applications. In the field of manufacturing, some researchers have proposed communication platforms for various applications, in particular robotic systems and reconfigurable assembly systems. However, the application of these communication models in the fixturing area has not been reported, yet. The reason for this appears to be once again that fixtures have been only recently accepted as intelligent or automated components. Consequently, the development towards a communication infrastructure, tailored to the fixturing domain is identified as an important step towards next-generation intelligent workholding.

2.6.   Knowledge Gaps

Despite the significant developments in the reported research areas, the available systems do not yet fully address the needs of reconfigurable and adaptive fixtures. The results of the literature review show an ongoing trend towards adaptive fixturing systems that utilise sensor feedback and programmable actuators to introduce reactivity in the clamping process. However, currently these systems are not reconfigurable. Existing approaches for reconfigurable fixtures on the other side appear to be limited to specific setups and rely on dedicated software routines, tailored to a particular configuration. Additionally, these systems lack a software framework that supports the platform-independent integration of devices. As a consequence, automatically reconfigurable fixtures have not been properly adopted by industry up to now. The following knowledge gaps have been identified as current barriers for the successful transition from traditional fixtures as passive devices to automatically reconfigurable and adaptive parts of modern manufacturing systems.

Lack of a generic data model for the representation of the capabilities of adaptive fixturing systems.

Current data models for fixture representation concentrate on the fixture design of traditional, modular fixtures comprising passive elements. Consequently, these approaches address only the structural aspects of the fixture. However, to allow automated reconfiguration of adaptive fixtures, a data model is required that is able to capture the capabilities of the fixturing systems and their devices, including sensors and actuators. This must also allow for the combination of elements and their related capabilities. Further, whilst a number of researchers have applied object-oriented techniques in other manufacturing areas, these models are domain-specific and cannot be directly applied to fixtures. Additionally, the existing models appear to utilise only basic object-oriented techniques such as inheritance and are consequently limited to a merely hierarchical representation of the system in question. While such a model is an important requirement for any automated system, it does not necessarily allow for the exchange of software methods during the operation of the system which is the key to achieve dynamic reconfiguration and vendor-independent device access. Thus, for the development of a truly reconfigurable software framework that can support the dynamic reconfiguration of adaptive fixtures, other techniques such as object-oriented design patterns and method delegation are required.

Lack of a fixture reconfiguration method defining the decision-making sequence for the automated reconfiguration of a wide range of different fixture setups

A number of automatically reconfigurable fixturing systems have been proposed in the literature. However, the reconfigurability of these systems is limited to specific fixture setups and lacks general applicability for other fixture layouts. The reason for this is that the adaptability of the software is not sufficiently taken into consideration. The software routines that are utilised to achieve the reconfiguration of these systems are customised to a particular fixture design comprising a set of vendor-specific hardware devices. For example, the system proposed by Lin and Du [35] only works with specific finger modules, whereas the system presented by Chan et al. [39] is restricted to a base plate with mounting holes. Other existing fixturing methodologies (see 2.3.1) only address the design phase

while neglecting the challenges of the reconfiguration during the manufacturing process, such as dynamic discovery of fixture modules, replacement of modules and the combination of capabilities. In general, a large part of the research effort has been restricted to purely mechanical passive devices with limited or no reactive capabilities. Further, the existing approaches on rapidly reconfigurable manufacturing systems do not address the fixturing domain and can therefore not be directly applied. As a result, the decision-making for automated fixture reconfiguration must be formalised in a methodology, independent of a particular fixturing system or design. In general, customised algorithms need to be replaced by a generic decision-making software architecture that can dynamically adapt to structural changes of the fixture setup.

Lack of a communication infrastructure for reconfigurable, adaptive fixturing systems that allows to dynamically establish communication channels and flexible information exchange

The advent of adaptive fixtures brings new challenges for the reconfiguration and operation of fixturing systems, such as the need for data exchange between the sensors and actuators. Additionally, to be an interactive part of the manufacturing system, future fixtures need to be able to communicate with other manufacturing equipment, too. Today, the communication channels in the reported fixturing systems are predefined during the development phase and cannot be changed dynamically during the operation of the device. For this reason, the presented examples for automated and adaptive fixtures do not provide sufficient mechanisms to dynamically change the fixturing layout by adding or removing equipment. To make fixtures truly reconfigurable, the communication links have to be established dynamically between the various devices whenever new modules are discovered. The literature review has shown that there is a myriad of different communication platforms available that support efficient data exchange in distributed applications. Some of these approaches have been utilised for reconfigurable manufacturing systems, such as robotics and reconfigurable assembly cells. However, an efficient, yet flexible communication architecture tailored to the fixturing domain is still missing.

2.7.   Chapter Summary

This chapter presented the results of a detailed literature review which provides the theoretical background for the research study. First the different fixturing developments of the last decades were highlighted, showing a continuous trend towards intelligent and adaptive fixturing solutions. After this, the available literature on fixture reconfiguration methods, fixture representation models and communication infrastructures for distributed, modular systems have been critically reviewed. It is concluded that the currently available methods do not sufficiently address the needs of reconfigurable, adaptable fixturing systems. Consequently, three main knowledge gaps have been identified, namely (1) the lack of an adequate data model; (2) a fixture reconfiguration methodology that is applicable for a wide variety of different systems and (3) a flexible communication infrastructure.

3. Research Methodology

3.1.   Introduction

The knowledge gaps identified in the literature review indicate a general lack of formalised software models and methods to support the reconfiguration of adaptive fixturing systems. For this reason the research addresses the reported gaps by the development of a sound software framework for the operation of reconfigurable adaptive fixturing systems. The complexity of this research requires the precise identification of the research domain and the definition of a detailed research methodology.

A systematic research methodology has been followed throughout the duration of the research. The main steps and phases of the methodology are illustrated in Figure 3-1, indicating also the relation to the other chapters. As it can be seen in the diagram the research methodology consists of four main phases. In the first phase an extensive literature review was carried out to get a detailed overview on the research available in the field of fixturing. This is the foundation to identify the state-of-the-art in flexible fixturing and define the knowledge gaps as described in chapter 2. The second phase focuses on the definition of the research domain and transfers the knowledge gaps into clear research objectives. Additionally, the system requirements for the software framework are identified in the form of a use case analysis. Based on this, the suitability of available technologies for the communication infrastructure are assessed. The third phase consists of the parallel development of the three core knowledge contributions of this research, namely the data model for reconfigurable and adaptive fixtures, the fixture reconfiguration methodology and the communication infrastructure. The three core contributions are highly interrelated and the main results of this work are described in the chapters 4, 5 and 6. In this context, the data model provides the definitions and interrelations of the main entities forming the system. The reconfiguration methodology uses the data model and defines the decision-making sequence that is carried out when a fixture needs to be reconfigured. The communication infrastructure realises the flexible communication of an arbitrary number of components in the framework. Finally, the proposed software framework has been applied

to a physical prototype of a reconfigurable fixture in order to demonstrate and prove the research results. In particular, a number of tests have been carried out to verify if the system meets the requirements and if the research has reached its declared objectives.
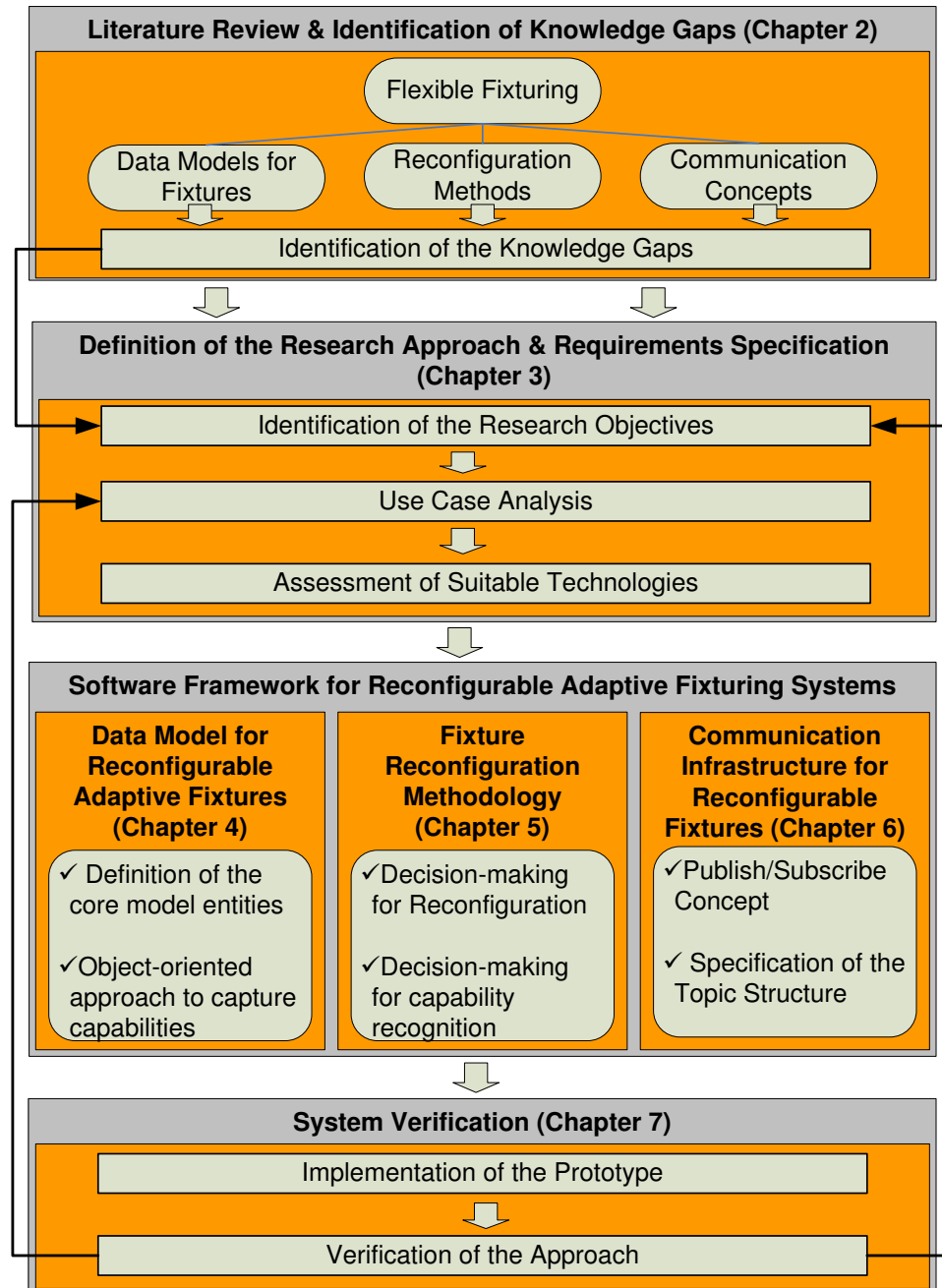
Figure 3-1: Overview on the Research Methodology

The following sections provide a comprehensive overview on the steps of the research methodology and the decisions made during the research. The description concentrates on the identification of the research domain (section 3.2), the requirement analysis (section

3.3) and the assessment of suitable communication technologies (section 3.4). Additionally, two conceptual fixture designs are introduced in section 3.5 which are used for illustration purposes throughout the rest of the thesis. The results of the literature review, the development of the core contributions and the system verification are covered in individual chapters and are therefore omitted here.

3.2.   Definition of the Research domain

3.2.1. Definition of the Knowledge Contributions

This research work addresses the identified knowledge gaps by the development of:

- **A data model for the representation of the capabilities of reconfigurable and adaptive fixturing systems**

  The model is based on an object-oriented approach which creates a hierarchic view of the fixture using generalisation and abstraction principles. However, unlike existing approaches [57, 84, 87], it is tailored to the operation phase of the fixture and not for the fixture design phase. For this reason, the model captures not only structural aspects of the fixture layout, but also provides the means to represent the changing capabilities of adaptive fixtures when components are added, removed or replaced. In addition, advanced object-oriented techniques such as design patterns and software delegation are used to allow the dynamic access and flexible substitution of the model elements during the operation of the fixture. Other research [104, 105] shows that these techniques are the key to create reconfigurable and re-usable software systems. The data model proposed in this research builds upon these approaches and applies them to the fixturing domain. For the formalisation and definition of the relationships between the model elements the Unified Modelling Language (UML) has been used which guarantees a platform-independent definition of the model.

- **A fixture reconfiguration methodology**

  The core of the methodology consists of two interrelated parts. The first part deals with the recognition and combination of the capabilities of the fixture elements as a

result of structural changes of the fixture layout. The approach is based on the formal description of capabilities with the Extensible Markup Language (XML) and describes the steps to instantiate the model elements in order to represent a fixturing system. The result is a layered object hierarchy where model elements of higher layers delegate requests to the model elements of subordinate layers. The principle of software delegation has been used for the development of reconfigurable systems in other areas [75, 76]. This research aims at transferring this principle to the fixturing domain. The second part defines the decision-making sequence to rapidly adapt a fixture for the next workpiece. The main idea is to dynamically link the software objects representing the physical setup with the objects representing the predefined fixture design parameters. Based on this assignment, the required reconfiguration sequence can be generated. While there are a number of reconfigurable fixturing systems available [34, 35, 37, 39], this method will provide a more general solution that is applicable not only for one particular setup, but for a variety of different fixturing systems. Additionally, it enhances existing adaptive fixturing approaches [47, 49] with a reconfiguration method.

- **A flexible communication infrastructure for the operation of adaptive fixturing systems**

  The methodology and the data model are integrated with a communication infrastructure which allows the flexible communication between the various parts of the fixturing system. In contrast to existing fixturing systems with hardwired connections between the devices, the communication infrastructure provides the means to dynamically establish communication channels when components are added, removed or replaced. The communication infrastructure uses an existing middleware standard [117] and applies it to the fixturing domain which so far lacks any standardised communication platform. Moreover, standardised library interfaces for adaptive fixturing equipment are defined which is the basis to achieve vendor and platform-independent device access.

These knowledge contributions are the fundamental cornerstones for the software framework, which is illustrated in Figure 3-2 with its major inputs and outputs. The main idea is a paradigm shift from programming effort towards configuration effort. In other words, instead of developing customised software routines for a specific fixture setup, engineera would configure the framework with the necessary information about the fixturing system. This includes the formal description of the capabilities of the fixture components, the device libraries for the hardware access as well as some information about the position and orientation of the fixture modules. As a result, the framework provides ready-to-use software applications for the operation of the fixturing system.
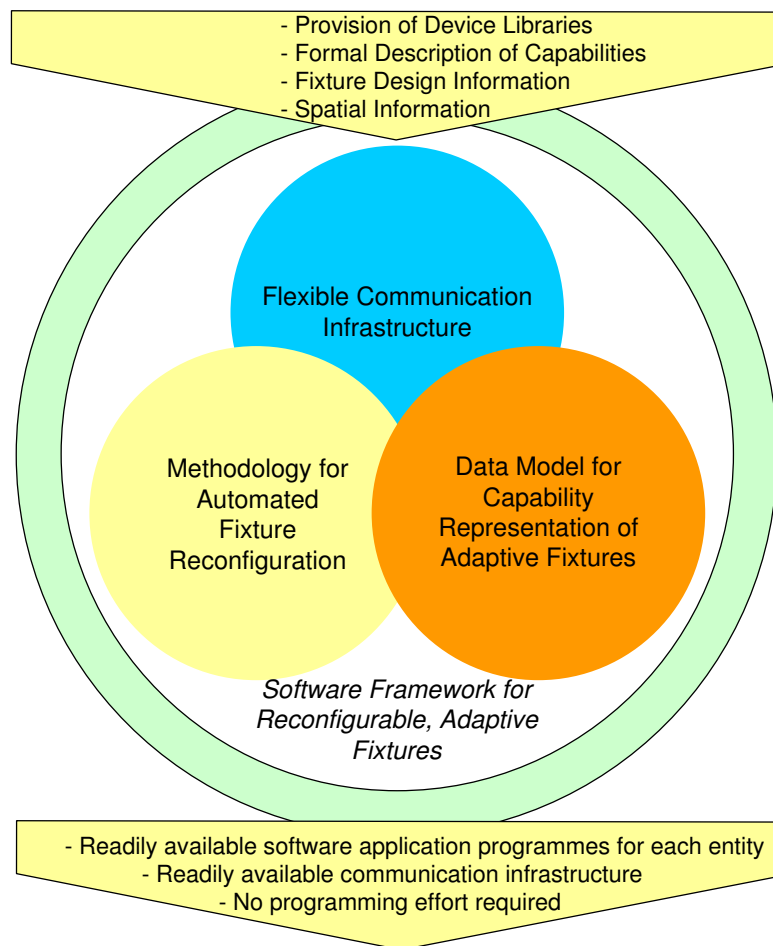
Figure 3-2: The knowledge contributions in the context of the software framework

3.2.2. Assumptions and Limitations

The final result of this research will be a software framework for reconfigurable adaptive fixturing systems that can be applied to a plethora of different fixture setups. For this

reason, the framework is a step towards the successful development of next-generation, intelligent fixtures. However, in order to limit the complexity of the research task, a number of general assumptions and limitations have been identified.

- **Predefined Fixture Design**

  This research work does not address the problems of the fixture design phase. Instead, it assumes that all fixture design parameters such as clamping positions or force profiles for each workpiece are readily accessible when the fixture needs to reconfigure.

- **Reconfigurable fixtures using independently controllable fixture modules**

  The framework is tailored to fixtures that have the ability to be reconfigured and have a modular structure. Consequently, the framework cannot be applied for the operation of dedicated fixtures. Additionally, the framework cannot be used for phase-change fixtures because they are not based on a modular structure. Within the scope of this thesis, the term "fixture module" refers to a physical component with an own software program that is in contact with the workpiece. The fixture modules communicate with the so-called fixture coordinator software which manages the overall fixturing process.

- **Degree of Automation**

  In order to generate the model elements and perform the reconfiguration process automatically, adequate computer technology is required for the fixturing system. This is the case for adaptive fixturing systems comprising actuator and sensor devices. On the contrary, traditional modular fixtures which consist of passive metal blocks typically lack this kind of computational power and can therefore not directly benefit from this research. However, passive elements can still be represented by the data model in which case the framework can assist the operator during the fixture reconfiguration.

- **Components with linear movements**

  To limit the scope for the definition of the data model, the repositioning of elements is limited to linear movements. This means, fixtures that reposition their elements with rotational movements, such as the double revolver fixture by Tuffentsammer [34] are currently not addressed by the research. However, due to the object-

oriented approach, the data model can be extended with classes to accommodate such systems.

3.3.   *Requirements Specification*

To capture the functional requirements of the software framework a use case analysis has been carried out. This is a standardised method for analysing the required functionalities of a system from a user's point of view. Hence, any technical details of how a certain functionality can be achieved is omitted. The results of the analysis are summarised in the use case diagram in Figure 3-3 which uses the notation conventions of the Unified Modelling Language 2.0 (UML 2.0) [142, 143]. According to this standard, the system (depicted as the large rectangle) is described in terms of actors, use cases and relationships. An actor, depicted as a stick man, is a role outside of the system under study that interacts with it [144]. This can be either a human being or another system. A use case refers to a certain functionality the system provides to actors. It is illustrated as ellipsoids in the diagram. Use cases can be specialised by other use cases which is represented by a line with an unfilled arrowhead pointing from the specialised use case to the more general use case. Additionally, the so-called "include" relationship is used to integrate one use case as a logical part into another use case. Even though the relationships between use cases may suggest a natural flow to the reader, use case diagrams do not indicate any sequences of actions or flows of events. Further information on use case diagrams can be found in [143].
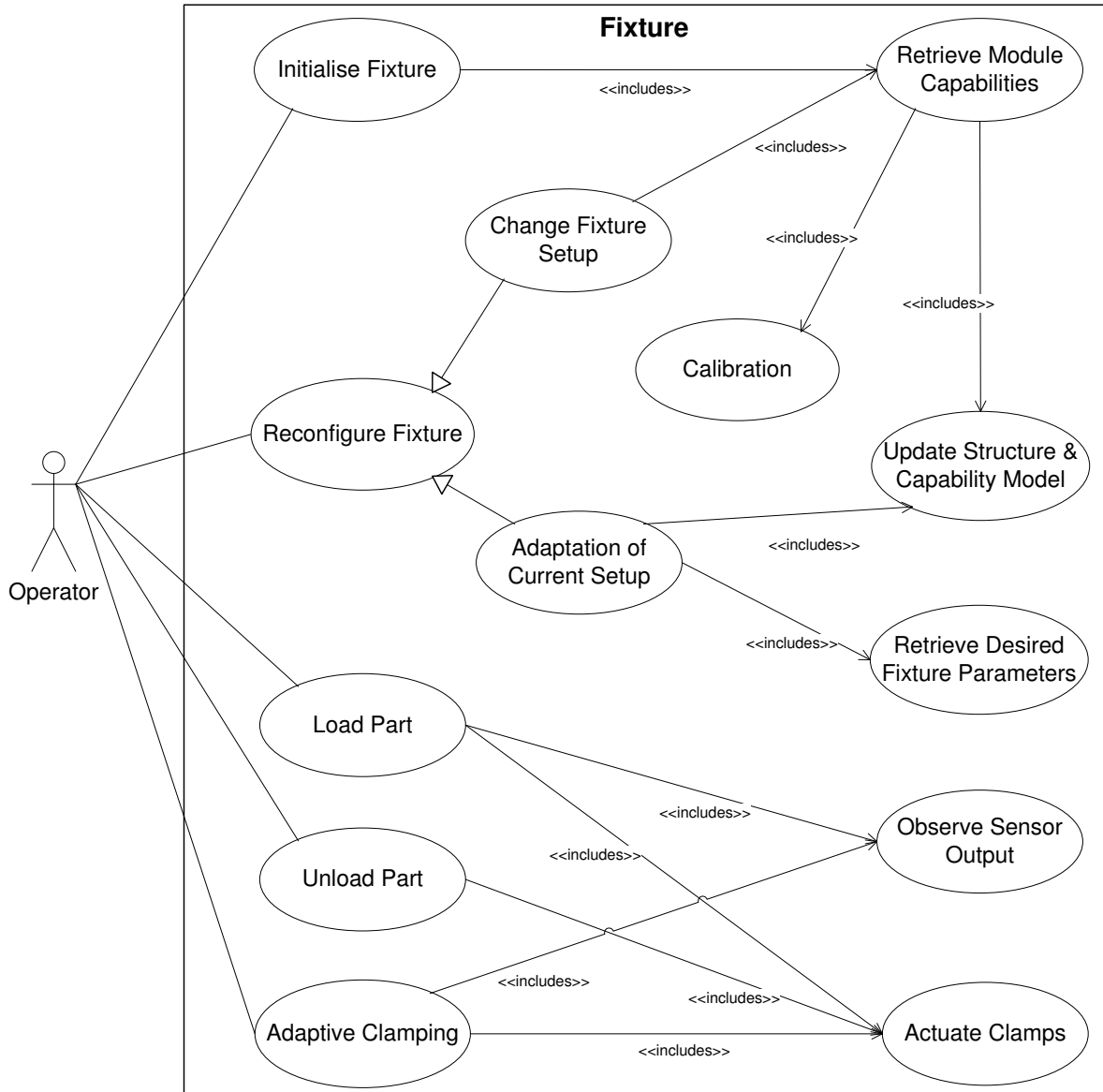
Figure 3-3: Use Case Diagram for the Fixturing System

In the context of this research, the operator has been identified as the main actor with regards to the fixture. Other subsystems that potentially interact with the fixture have not been modelled as individual actors because for the development of the knowledge contributions it is irrelevant if a certain functionality is invoked by a human operator, a robot or another part of the shop floor. Thus, the role operator represents any stakeholder that calls a service provided by the fixturing system. Furthermore, five top-level and six second-tier use cases have been identified which are described in the following sections. The top-level use cases are directly triggered by inputs from the user, while the second-tier

use cases are performed internally by the fixture software in order to satisfy a certain top-level use case.

3.3.1. Initialise Fixture

The first use case is the initialisation of the fixturing system. This requires the software framework to recognise the structure and the capabilities of the underlying fixture. In order to achieve this, the capabilities of each fixture module must be determined and communicated to the rest of the system which is represented by the second-tier use case "Retrieve Module Capabilities". This requires a communication infrastructure that is able to

- Recognise an arbitrary number of modules in the system
- Exchange capability information in a defined format

When all information about the fixture modules is collected, this information can be combined to form a complete view of the fixturing system which is part of the use case "Update Structure and Capability model". Additionally, a calibration step is required for the correct functioning of every fixture. Calibration, however, requires specific routines depending on the actual underlying fixture hardware. Therefore, the framework takes into account the necessity of a calibration step but does not define a specific algorithm. After these essential steps the fixturing system is ready to work.

3.3.2. Reconfigure Fixture

This use case addresses the functionality of a reconfigurable fixture to adapt itself in response to changing requirements. The reasons for fixture reconfiguration are typically the need to clamp several parts with one fixture or to process multiple surfaces of a workpiece using the same fixture. Two forms of fixture reconfiguration have been identified which are both addressed by the proposed framework.

The first form is concerned with the physical change of the structural layout of the fixture and the associated use case has been named "Change Fixture Setup". This typically includes the manual addition, removal or replacement of fixture modules as well as the modification of the internal device structure of an existing fixture module. The fixture must be switched off during these changes and the initialisation routine is required after the

reconfiguration procedure. The software framework must be able to recognise the structural changes and no reprogramming shall be required in order to allow the functioning of the new fixture. The second form of fixture reconfiguration is called "Adaptation of the current setup" and occurs more frequently than the previous use case. In contrast to the previous reconfiguration type, this use case addresses the ability to adapt the existing fixture configuration without the need of dismantling its current structure. Examples for this are the modification of the fixture with new clamping parameters such as initial clamping forces or the maximum allowable reaction forces. Additionally, the ability of fixtures to automatically reposition their clamping modules is addressed by this use case. For example, the prototype described in chapter 7 allows to relocate its modules on rail guides. In order to achieve the adaptation process automatically, the framework must retrieve the predefined fixture parameters for the new configuration from a data base. These need to be compared with its current structure and all necessary steps to transform the current fixture into its desired state must be determined and executed.

3.3.3. Load Part

This use case refers to the ability of the software framework to initiate the clamping of a part with the fixture. The procedure requires that the reconfiguration step has been completed and the workpiece is correctly positioned in the fixture working envelope. The repositioning of the workpiece in the fixture or the adaptation of the tool path is not the subject of this research. Upon a trigger signal, the clamps must be actuated towards the workpiece in order to exert a predefined initial clamping force. The execution of this use case requires the retrieval of the sensor data from the modules and adequate actuation of the modules under real time conditions. The actual determination of the correct clamping points and initial forces is part of the fixture design phase and therefore beyond the scope of this research.

3.3.4. Unload Part

Similar to the previous use case, this addresses the ability of the fixture to accurately release the part from the fixture. This procedure requires an input signal as a trigger and as a response each clamping module should retract to its home position, thereby releasing the

part. In order to do this, the framework must be able to retrieve the current position of the fixture modules and actuate the clamps until they have reached their desired position.

3.3.5. Adaptive Clamping

During the manufacturing of a part, particularly in machining processes, the external forces acting on the part change dynamically. The promising results of adaptive fixtures to improve the workpiece quality by reacting to the changing external forces has been highlighted in the literature review in chapter 2. For this reason, this use case refers to the ability of the framework to observe the sensor data and issue appropriate commands in order to adapt the clamping forces. It is assumed that the use case "Load Part" has been completed.

Rather than focusing on the development of the actual force profiles, the framework aims at providing the infrastructure to establish the communication in a flexible way. The term 'flexible' addresses the challenge of achieving information exchange in adaptive fixtures whose number of modules and their interrelations between each other can change over time. Additionally, since the fixture modules can incorporate different hardware devices, it is possible to implement same capabilities with different technologies. For example, a clamping module can realise the clamping capability by several types of linear actuators (e.g. electromechanical, hydraulic, pneumatic). Clearly, these technologies require different input signals to achieve a certain clamping force. For this reason, an additional layer of abstraction is necessary that makes the software framework independent from a certain platform or vendor. This includes a common data format for the communication between the fixture modules and the fixture coordinator. The exchanged information needs to be mapped into the platform-specific signals required for the device access, thereby rendering the framework compatible for a plethora of different hardware profiles.

In order to achieve this, the framework must have the ability to be parameterised with device libraries to correctly interpret the data coming from the devices as well as sending appropriate signals, the device hardware can understand. Additionally, the framework will utilise the module capability descriptions which are obtained during the initialisation

routine as described in the use case "Initialise Fixture". This is illustrated in Figure 3-4 which shows a simple fixture module communicating with the fixture coordinator.
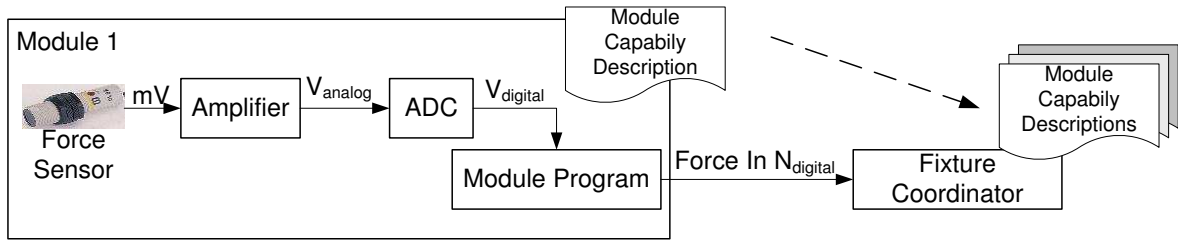
Figure 3-4: Simplified Scheme of Communication between a Module and the Fixture coordinator

The module consists of a force sensor which is accessed by the local module software. The latter is configured with the characteristics of the sensor and informs the fixture coordinator about its resulting capabilities during the initialisation routine. Among other details, this information declares how the force readings have to be interpreted. During the operation of the fixture, the module software calls the device library to retrieve the current sensor signal and converts the voltage signal to a force value in Newton. As a result, the local module program acts as a software facade which encapsulates the hardware access to the sensor device, while the fixture coordinator is able to interpret the received force values in order to process them.

3.4. *Assessment of Suitable Communication Technologies*

The specification of the user requirements shows that the envisioned software framework is characterised as a complex distributed system where an arbitrary number of modules need to communicate. To manage the complexity of the communication it was decided to utilise and adapt an available middleware technology. Apart from reducing the risks of failure, the development of the communication infrastructure on top of a recognised standard increases the acceptance of the proposed system and facilitates potential take-up from industry in the future.

3.4.1. Definition of Technical Requirements

As revealed by the literature review, a number of different middleware technologies are available for various application domains. Examples for these are the Common Object Request Broker Architecture (CORBA) [114], Data Distribution Service (DDS) [117], Java

Messaging Service (JMS) [129] and many more. In order to choose the most suitable middleware technology for the communication infrastructure a structured approach for the assessment was followed. In the first step, the technical requirements were defined that are imperative for the realisation of the software framework. These are summarised as follows:

- **Reactivity to Dynamic Network Topology Changes**

  As fixture modules can be added or removed to the platform, the infrastructure must be able to automatically recognise these changes in the network topology to invoke the reconfiguration process.

- **Platform-Independence**

  Since fixturing systems are provided by a variety of vendors, the framework needs to allow communication between a wide range of computer architectures, operating systems and programming languages.

- **Real-time communication**

  Fixtures are part of the production environment. As such, they are subject to timing constraints for the operation which are imposed by the process. This means, that the time between a sensor input and the system's response in form of actuator movements must be predictable and deterministic. In order to achieve this, the communication infrastructure needs to have full control over timing and resource usage.

- **Performance and Scalability**

  As described in [145], common middleware performance indicators are end-to-end latency and the throughput. The former refers to the time required to send a message from one communication end to another, whereas the latter is defined as the maximum amount of data that can be transferred per unit of time. For the fixture application, it is assumed that the end-to-end latency is more important than the throughput. Essentially, it determines how fast the fixture can react to sensor feed back. As a result of this, the latency determines for which processes the framework is applicable. Scalability is defined as the ability to maintain performance levels as more nodes are added to the system. Scalability issues can occur when more fixture modules are added to the system in response to more complex workpieces or when the fixture is connected with other subsystems via the communication infrastructure.

- **Overcoming Impedance Mismatch**

  As identified by Joshi [121], impedance mismatch is one of the fundamental challenges for the integration of distributed systems in heterogeneous environments. The term refers to the difficulties that arise when subsystems with disparate communication requirements in terms of data volume and data rates need to interact. For example, some applications produce data at higher rates than others are able to consume. Since the fixture is a part of a wider production environment, the software framework needs to interact with other subsystems of the factory like Human Machine Interfaces (HMI) or the machine control. For this reason, the communication infrastructure needs to offer a mechanism to fine-tune the data transfer individually for the requirements of the peers.

3.4.2. Selection of Middleware Candidates

In the second step a number of middleware technologies were selected for the assessment against these requirements. Due to the huge number of middleware solutions it is not feasible to assess all available technologies within the scope of this thesis. For this reason the assessment was limited to the most common solutions for each of the communication architecture paradigms discussed in the literature review (see section 2.5). The selected candidates are listed below. Further details on each of the candidates can be found in the literature review.

- **Common Object Request Broker Architecture (CORBA)**
- **Real-time CORBA (RT CORBA)**
- **Data Distribution Service (DDS)**
- **Java Messaging Service (JMS)**
- **Computer-Aided Manufacturing using XML (CAMX)**
- **Web Services (WS)**

3.4.3. Assessment of the Middleware Technologies

In the final step, the suitability of the presented candidates for the described technical requirements is compared. The aim of this step is to derive qualitative statements about the suitability of the technologies with respect to the requirements which results in a ranking.

For this reason an ordinal scale is introduced which classifies the support of a certain requirement in four categories:

- o Category I: No support
- o Category II: Weak support
- o Categroy III: Good support
- o Category IV: Very good support

Although it is arguable whether a certain technology offers weak, good or very good support, this classification indicates tendencies and at the end a conclusion can be drawn about the most suitable choice for this research study. The evaluation is based on available literature and on personal experience with these technologies. The results of the evaluation are summarised in Table 3-1. The category II, III and IV are illustrated by one, two or three stars, respectively while for category I a dash is used.

	Network topology Changes	Platform Independence	Real-Time Communication	Performance and Scalability	Impedance Mismatch
CORBA	-	★★★	-	★★	-
Real-Time CORBA	-	★★★	★★★	★★★	★
DDS	★★★	★★★	★★★	★★★	★★★
JMS	-	★★	-	★	★
CAMX	★★	★★★	★★★	★★	★★
Web Services	★	★★★	★	★★	★

Table 3-1: Assessment of Middleware Technologies

Support of network topology changes

In terms of the support of network topology changes, CORBA, RT CORBA and JMS do not offer off-the-shelf mechanisms to inform the application about other participants being added, removed or replaced. In order to achieve this functionality own proprietary protocols for the discovery of participants have to be developed which is cumbersome and error prone. An example for a discovery mechanism in a CORBA-based system can be found in [146]. A number of approaches have been published for the dynamic service discovery of

Web Services [147, 148]. However, these solutions are not yet readily available to use and Sun et al. [149] highlighted that the dynamic discovery of web services is still difficult to achieve. For CAMX, a method to optimise the allocation of clients to message brokers has been presented in [135]. Additionally, a number of event messages are defined to reflect the states of equipment [137], including aspects like liveliness. DDS automatically establishes an internal connection between participants with matching data topics and Quality-of-Service settings. Consequently, communication is automatically achieved when participants are plugged in. Additionally, DDS provides meta-information about the communication status of the participants in special data topics. Applications can subscribe to these topics and are notified by the middleware when other applications are plugged in, removed or replaced. To conclude, only DDS and CAMX satisfy this criteria off-the-shelf. If one of the other technologies is selected, this functionality needs to be developed.

Platform independence

In general, the second requirement is satisfied by all candidates. In particular CORBA, RT CORBA and DDS are defined as platform-independent standards which means they can be implemented on any kind of transport protocol and hardware. Additionally, when using these technologies the communication interfaces of the applications are defined in a platform-independent way which allows the integration of a large variety of different platforms and the automatic generation of source code. In CAMX the message transfer is accomplished with web-based communication using the Simple Object Access Protocol (SOAP). This also allows the collaboration of different platforms, since the SOAP protocol acts as a layer of abstraction. Web Services also use SOAP as well as other platform-independent protocols and therefore satisfy this requirement. JMS, is a JAVA-specific API-standard. Consequently, it is hard to establish communication with other applications that are not written in the JAVA programming language. However, since JAVA programs run in a so-called JAVA Runtime Environment (JRE), they are portable over different operating systems. Additionally, Sanchez et al. [150] have demonstrated how JMS can be accessed from other programming languages based on additional libraries. Although, their research shows that this introduces further performance losses in terms of latency and

throughput, the requirement of platform-independence can be satisfied. For this reason, JMS was classified into category III with regards to this criterion.

Real-time support

With regards to real-time suitability, JMS and CORBA do not provide sufficient means to ensure deterministic and predictable data exchange. For this reason these two candidates appear to be less suitable for the fixturing framework application. Web Services typically offer weak support for applications with real-time requirements. To overcome this gap, recently numerous researchers have tried to integrate the Quality-of-Service paradigm with Web Services [149, 151, 152]. However, there is still no uniform standard available and the solutions are not yet mature. The remaining candidates are specifically tailored to the needs of real-time applications and are therefore satisfying this requirement. In fact, the real-time support of all these candidates is based on a rich and mature implementation of the Quality-of-Service approach.

Performance and scalability

The described performance indicators (end-to-end latency and throughput) are influenced by a large number of factors, including the speed of the CPU, the operating system, the programming language, the message length, the number of communicating systems and others. Since there are extensive benchmark tests available, it is beyond the scope of this research to compare the performance of the different technologies in terms of quantitative measurements taken from own experiments. Instead, the assessment is based on information from literature and, more importantly, conclusions about potential performance differences are drawn based on the architecture characteristics of the candidates. Recent performance tests for CORBA and RT CORBA have been reported in [153, 154]. Additionally, large amounts of performance data have been gathered by the Open CORBA Benchmarking Project which provides an online database of benchmarks for a large number of CORBA systems [155, 156]. In this context, Gokhale and Schmidt [157] reported that most CORBA implementations do not sufficiently address the objective of low latencies. A performance comparison between Web Services and CORBA has been published by Gray [158] which concludes that despite recent improvements of the former, Web Services are

considerably slower with a higher consumption of network bandwidth and CPU cycles. Similarly, in the experiments of Juric et al. [159] Web Services proved to be 9 times slower than Remote Method Invocation which is based on similar principles as CORBA (see section 2.2.4). The main reason for the performance problems can be found in the overhead related to the SOAP message processing [159]. For DDS detailed benchmark tests have been carried out as part of the Real-Time DDS Examination & Evaluation Project (RT-DEEP) [160]. Results of this research have been reported in [161, 162]. According to this, end-to-end latencies can be lower than 50μs [163]. Compared to CORBA, DDS achieves potentially faster data exchange since it does not route data through a central message broker. Additionally, the publish/subscribe approach followed by DDS minimises the communication overhead when the number of nodes is increased. However, these issues can be overcome with RT CORBA when the so-called event service is used. Essentially, this service establishes publish/subscribe-like data channels. Compared to JMS and CAMX, the data transfer with DDS has the potential to be significantly faster. The reason for this is the data-centric approach of DDS while the other two technologies are message-oriented. This means, in these systems information is encapsulated in a message body which has to be parsed and analysed upon its receipt. This interpretation of messages requires additional processing time in each node. DDS on the other side shares information as user-defined data types. For this reason, there is less communication overhead because there is no need for message headers and the received data is immediately available for the application. Furthermore, JMS and CAMX use message brokers as centralised entities which are potential performance bottlenecks and failure points. DDS on the other hand establishes peer-to-peer communication between the participating applications. A more detailed comparison between JMS and DDS has been conducted by Joshi [145]. JMS is arguably the slowest option of all candidates, since it relies on the JAVA programming language. Such programs do not run as executables, but are interpreted by the run-time environment which slows the execution down.

Impedance mismatch

The impedance mismatch requirement is best addressed by DDS and CAMX. The reason for this is the loose coupling due to the publish/subscribe approach and the Quality-of-

Service concept offered by both middleware technologies. The QoS-parameters of CAMX primarily aim at satisfying the needs of real-time communication by grouping messages into four categories (closed-loop real-time control, supervisory control, operator control, other purposes) with different priorities [135]. In contrast, the QoS-concept offered by DDS has more configuration options. In addition to the parameters ensuring real-time communication, DDS allows to apply time-based and content-based filters to individual topics which prevents applications from being flooded with data. The client/server approach of CORBA and RT CORBA results in tightly coupled connections which makes it hard to integrate applications with disparate communication requirements. Therefore, CORBA is not supporting this requirement. However, the event service and the QoS-concept of RT CORBA alleviate this drawback. Although JMS and Web Services do not offer any specific features to address the problem of impedance mismatch, their approach of loosely coupled communication supports the integration of applications with disparate communication requirements.

To conclude, a number of middleware candidates have been assessed against technical requirements of the software framework for adaptive fixturing systems. As a result, CORBA , Web Services and JMS are less suitable for this application as they lack real-time support and do not satisfy other important requirements. Although, RT CORBA offers a fast, robust and platform-independent communication service, its client/server concept introduces tightly coupled communication channels which cannot adequately support many-to-many communication. Therefore, CAMX and DDS appear to be more appropriate for this kind of application. Overall, the assessment revealed that DDS is the preferred choice for the fixture framework. It is specifically designed for the needs of platform-independent real-time applications with low-latencies and addresses the challenge of impedance mismatch. Moreover, since DDS is an application-neutral standard it can be adapted to the fixturing domain. CAMX on the other hand is designed for assembly applications. This means, although the middleware allows the definition of extensions, the majority of the standardised CAMX messages cannot be applied directly to the fixturing domain. Consequently, DDS was chosen as the communication infrastructure of the software framework.

This decision has a high impact on the design of the software framework. In particular, the communication infrastructure described in chapter 6 needs to be based on the data-centric publish/subscribe paradigm. Data types for the exchanged information between the fixture modules, as well as an associated data topic concept have to be defined as part of this research. Additionally, the research shows how the Quality-of-Service concept offered by DDS can be utilised to address the challenges of reconfigurable fixturing systems.

3.5. Overview on Example Fixtures for Illustration Purposes

In this section the conceptual designs of two different fixturing systems are presented in order to facilitate the understanding of the concepts described in the following chapters. The first fixture is based on a rail frame which allows the automatic repositioning of clamps and locators in order to reconfigure for a variety of workpieces. This concept has also been realised as a physical prototype and was used for the tests which are described in chapter 7. The second design uses a base plate with mounting holes on which a set of fixturing elements can be arranged. This concept has not been implemented as a physical test bed. Instead, it is used to illustrate the general applicability of the methodology and the data model presented in the thesis.

3.5.1. Rail-based Fixturing System

The basic idea of this system is to utilise rail guides on which a set of clamping and locating elements are mounted. These elements can be moved continuously along the rails to achieve fixture reconfiguration. A variable number of rails can be arranged in 3D space, depending of the different part families. Figure 3-5 shows a design drawing of a configuration with four rail guides, forming a closed working envelope. Each rail consists of a pair of linear low friction guides which are mounted on a plate to provide adequate vertical and lateral support for the guides and also raise them in height. To allow the repositioning of the linear actuators and other fixturing elements, a number of carriers are attached to the rails that can slide along the linear guides. As shown in the detailed view in the bottom right corner of the drawing, the carriers consist of a runner element on each linear guide and a metal plate on which a clamp or other equipment can be mounted.
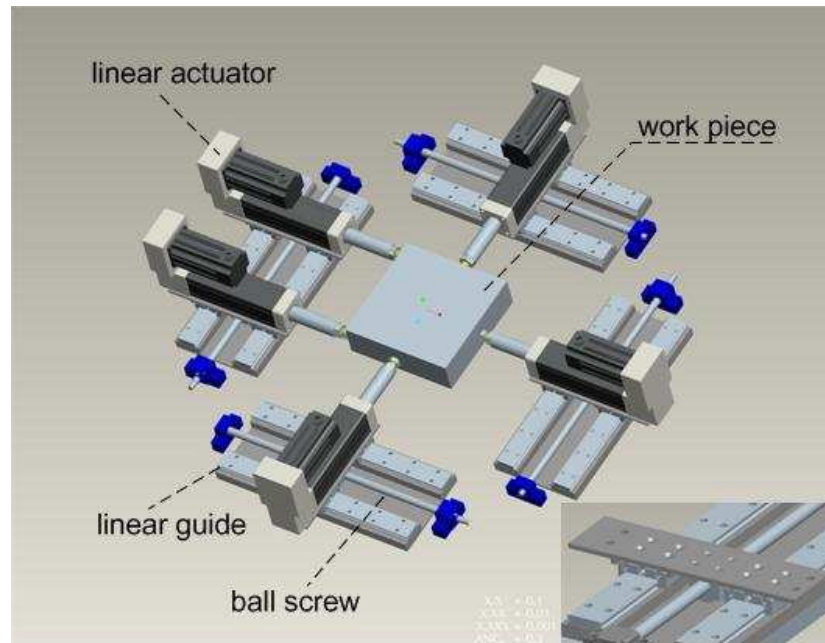
Figure 3-5: Conceptual Design of a Fixture with Four Rails

To realise the repositioning, a ball screw is mounted between the linear guides. The ball screw nut is mounted on the lower surface of the connecting plate and the ball screw shaft is held in place by means of ball bearings at the ends of the shaft. One of the ends is driven directly by a co-axially mounted servomotor with integrated positional feedback which is not shown in the drawing. The position of the runner pair on the linear guides is thus controlled through this motor. Different actuators or passive fixturing elements are mounted on top of the connecting plates. The actuators act as clamps, whilst the passive units act as locating or supporting points. Actuating units could be based on any available actuating technology (e.g. pneumatic, hydraulic, electromechanical) depending on the application requirements. The linear actuators shown in the drawing are each driven by individual AC servo motors and incorporate a displacement and a force sensor to provide feedback capabilities. The servo motor has a locking mechanism, granting the formation the ability to be used as a clamp and a locator. Detailed descriptions on the selected equipment for the physical test bed are provided in chapter 7.

The general concept can be adapted according to the application requirements. For example, the end user may choose to include more or less linear guides, runner pairs,

different types of active or passive fixturing elements and could choose between top or side clamping. Figure 3-6 illustrates three different variations of the rail-based concept.
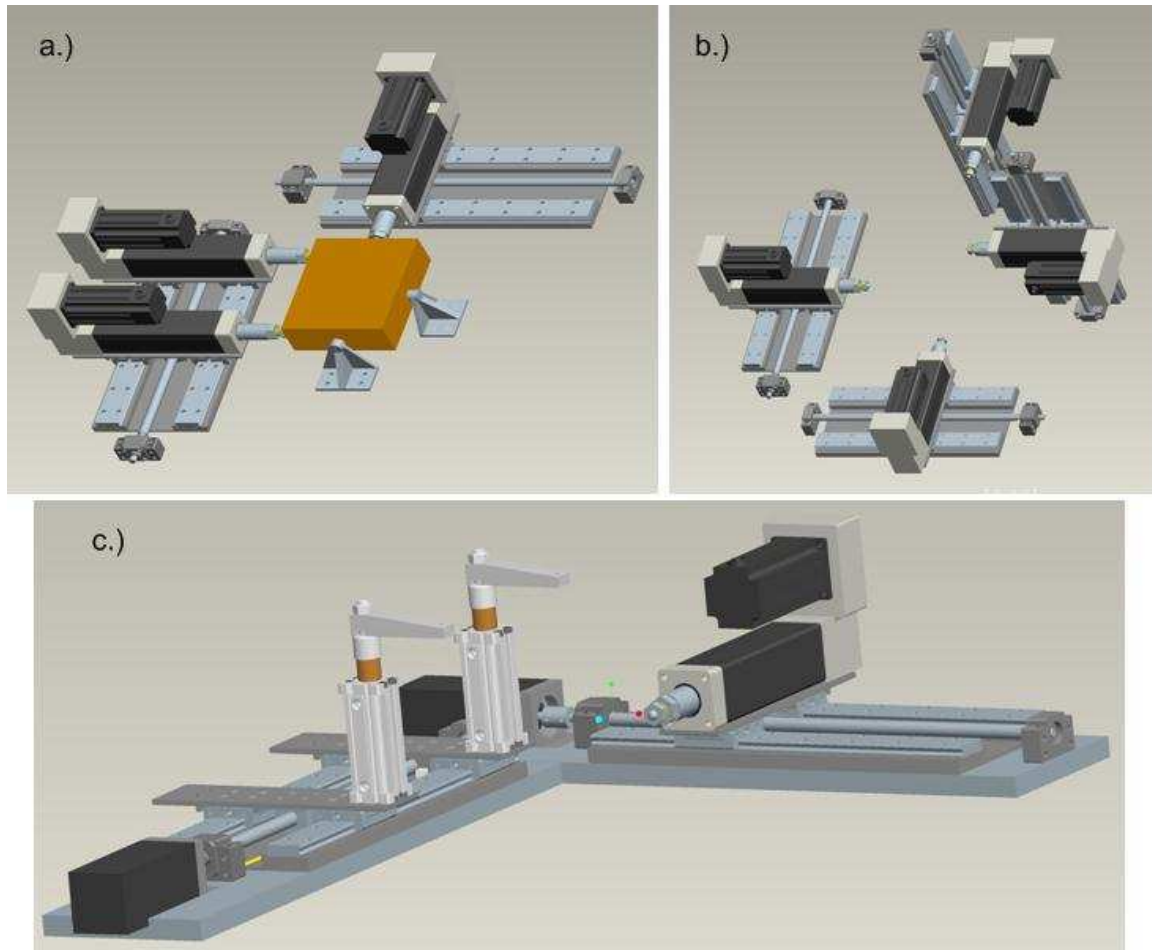
Figure 3-6: Variations of the Rail-based Fixture Design

The fixture in Figure 3-6.a shows a configuration with two rails and two rigid locators. Figure 3-6.b demonstrates how the concept can be scaled up to a 3D solution. Finally, Figure 3-6.c illustrates the use of different clamping elements. In this example, two swing clamps are mounted on the rails which can be used for top clamping of workpieces.

3.5.2. Fixture using a Base Plate with Mounting Holes

The second example consists of a different fixture design which uses a base plate with a set of mounting holes. The holes can be used to attach a variety of fixture modules like clamps or locators onto the plate. This approach is similar to the systems proposed by other researchers [37-39] and a design overview is provided by Figure 3-7.
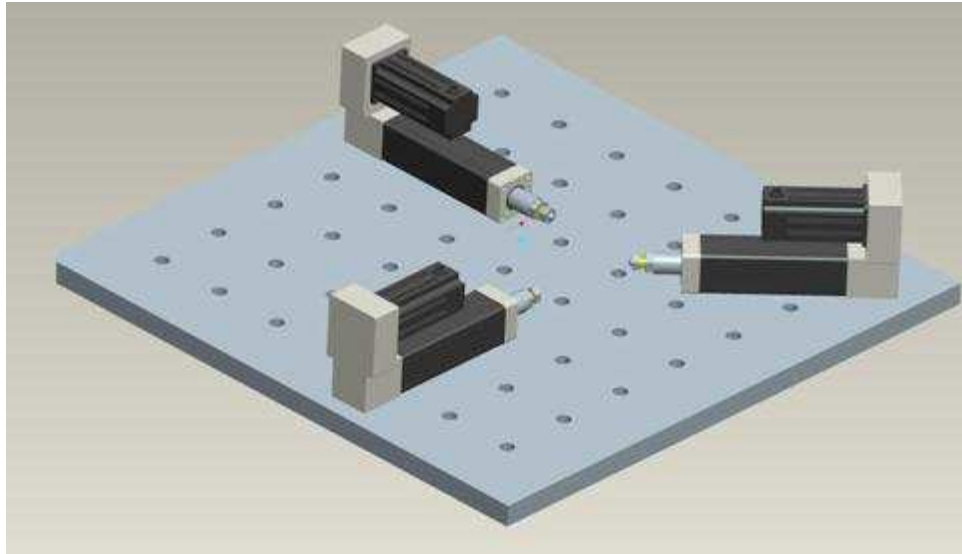
Figure 3-7: Conceptual Design of a Fixture Using a Base Plate with Mounting Holes

In contrast to the rail-based fixture design, this approach does not support the continuous movement of the mounted elements. Instead, a discrete number of mounting holes determines the possible positions and allows rotating the modules around the axis normal to the mounting hole. This concept requires an additional mechanism to reposition the fixture modules during the reconfiguration procedure which can be realised by a robot. Regardless of what repositioning mechanism is utilised, input information about the current position and orientation of the modules, their geometric dimensions and the desired positions is required in order to clamp the next workpiece.

Like the previous approach, the general design can be adapted to create different fixture layouts. For example, multiple base plates can be combined in 3D space with different hole patterns. Additionally, different types of fixturing elements can be added or removed and their positions can be changed on the base plate. Some of the variations are illustrated in the drawings provided by Figure 3-8. The design in Figure 3-8.a shows an arrangement of three linear actuators on a base plate with a 7 x 7 hole pattern. Figure 3-8.b illustrates an example where two base plates are combined and Figure 3-8.c demonstrates a fixture with swing clamps for top clamping and passive locator elements.
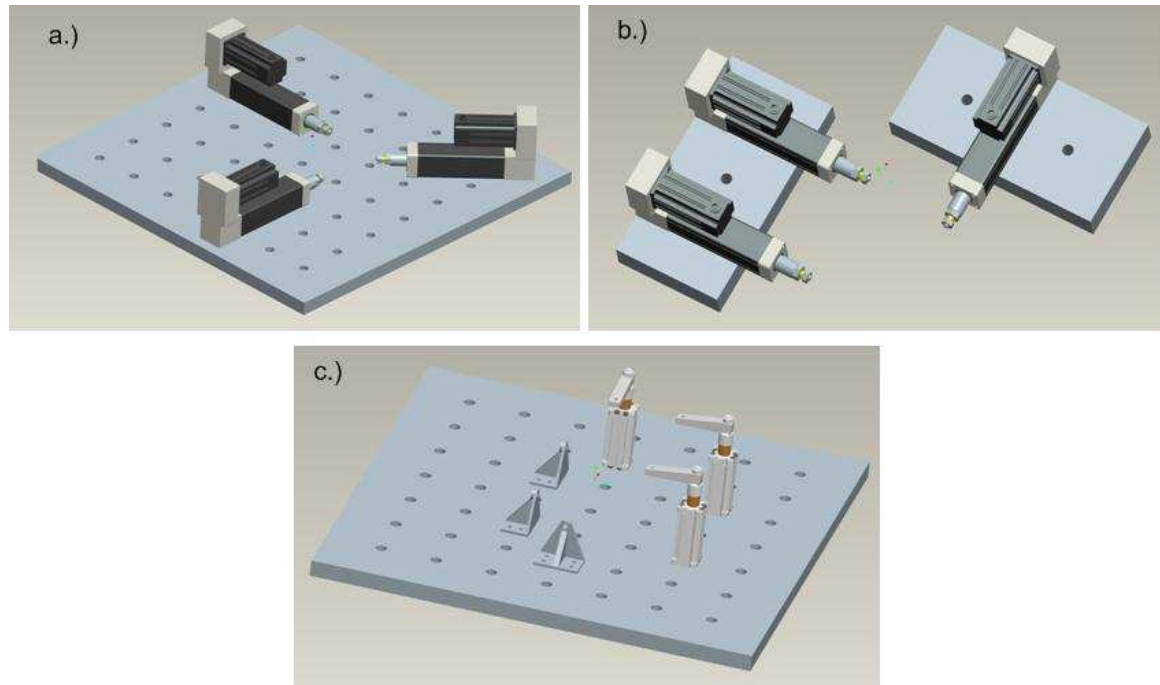
Figure 3-8: Variations of the Fixture Design with Base Plates and Mounting Holes

In the context of the presented examples, the fundamental aim of this research work is to develop a software framework which is applicable to any of the presented fixture design variations. This means, the data model must be able to represent the capabilities of the different fixture elements and the reconfiguration methodology must be formulated independently from specific design features like mounting holes or rail guides, thereby realising a concept with general applicability.

3.6.   Chapter Summary

This chapter has outlined the systematic research approach adopted in this study and described the main steps and key decisions taken in the development of the research. Based on the knowledge gaps identified in chapter 2, the research domain has been defined. This includes the definition of the key research objectives and the description of general assumptions for the software framework. A detailed requirement analysis has been carried out with the use case method to capture the necessary functionalities of the software framework. Furthermore, a suitable middleware technology has been selected as the basis for the communication infrastructure of reconfigurable, adaptive fixturing systems. The described selection process has resulted in the decision to adopt the Data Distribution

Service (DDS). To facilitate the understanding of the core knowledge contributions of this research, two exemplary fixture design concepts have been described which are subsequently used for illustration purposes in the following chapters.

4. Object-oriented Data Model for Reconfigurable and Adaptive Fixturing Systems

4.1. Introduction

To guarantee the general applicability of the software framework, a common data model has been formalised to represent the capabilities of a variety of fixturing systems. This is based on the observation that despite the structural differences of fixtures, common groups of functionalities can be identified. The purpose of this chapter is therefore to define the core model elements that serve as the foundation for the methodology described in chapter 5 and 6. Object-oriented techniques are utilised to logically group common aspects of fixtures that are subject to the reconfiguration procedure. On the other side, details that are irrelevant for the methodology are omitted. For example, for the automatic reconfiguration of fixture modules it is not necessary to capture the exact mechanical structure (e.g. the number of screws) below module level as these aspects are determined in the fixture design phase. Therefore, details that can be regarded as constant during the operation of the fixture are ignored by the model. In this way, the model provides a functional view of the fixture for the software framework.

To manage the complexity of the model, it has been divided into five logical parts. This has been done based on the package concept which is defined in the Unified Modelling Language (UML) standard [142, 143]. According to UML, a package is "a collection of model elements that can be of arbitrary types and that are used to structure the entire model in smaller, easily manageable units" [144]. Each package defines a number of model elements in terms of classes and data types. A class groups model elements with same specifications of features, constraints and semantics [144]. Data types are used by the classes for the specification of attributes. In contrast to classes, data types have no identity. This means, two instances of the same data type cannot be distinguished from each other if their values are identical. On the other side, two instances of the same class (called objects) can be distinguished at all times.

Section 4.2 provides a comprehensive overview on the data model and its package structure. Based on this, the subsequent sections describe the various model elements of each package. In more detail, section 4.3 describes the most fundamental elements which are used by other packages and therefore have been grouped together. Sections 4.4 and 4.5 focus on the classes related to the devices and the fixture modules, respectively. The details of transport components are explained in section 4.6. Finally, the classes needed for the reconfiguration methodology are described in section 4.7.

4.2. Model Overview

Figure 4-1 shows the package structure of the data model which consists of the five packages "Common Elements", "Transport Component", "Fixture Module", "Device" and "Reconfiguration". The package "Common Elements" defines the base classes and common data types used in other packages.
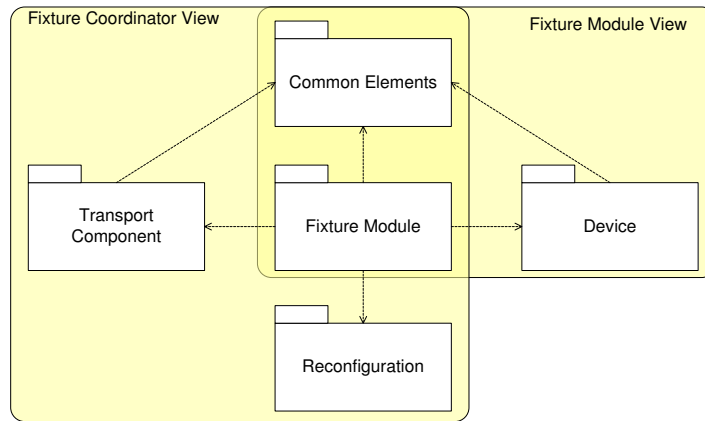
Figure 4-1: Overview of the Package Structure of the Data Model

The packages "Fixture Module", "Transport Component" and "Device" define the physical elements of a fixture as well as their capabilities. It therefore reflects the overall approach of this research work to decompose a fixture into these three categories. Essentially, these packages extend the classes Component and Capability from the "Common Elements" package. As a consequence, so-called dependency-relationships emerge between these packages which are depicted by dashed arrows pointing from the dependent to the independent package. For example, the package "Fixture Module" utilises the model elements defined in package "Common Elements" and further elaborates them. Finally, the

package "Reconfiguration" contains the elements that are required for the reconfiguration methodology.

The model elements of the packages are instantiated in both, the fixture coordinator software and software programmes representing each individual fixture module. As it can be seen in the picture, these software units utilise different parts of the data model, thereby creating two complementing views of the fixture with different levels of detail. The software of the fixture coordinator instantiates the model elements of the packages "Common Elements", "Fixture Module", "Transport Component" and "Reconfiguration". Consequently, it generates a global view of the entire fixture whilst remaining unaware of the internal devices and their functionalities within each individual module. These details are encapsulated in the software for the modules which provides each module with a local view of its own devices and capabilities. Both software units utilise the model elements defined in the packages "Common Elements" and "Fixture Module" which highlights the central role of the fixture modules in the data model.

4.3. Model Elements of the Package "Common Elements"

This package defines the two main classes Component and Capability which serve as the roots for the entire model. Both classes are abstract which means that they are not directly instantiated by the software framework. Instead, these classes encapsulate properties that are common for the child classes in other packages that inherit from them. An overview of the package contents is shown in Figure 4-2. A summary of the utilised UML notations is provided in the Symbology section in the beginning of the thesis.
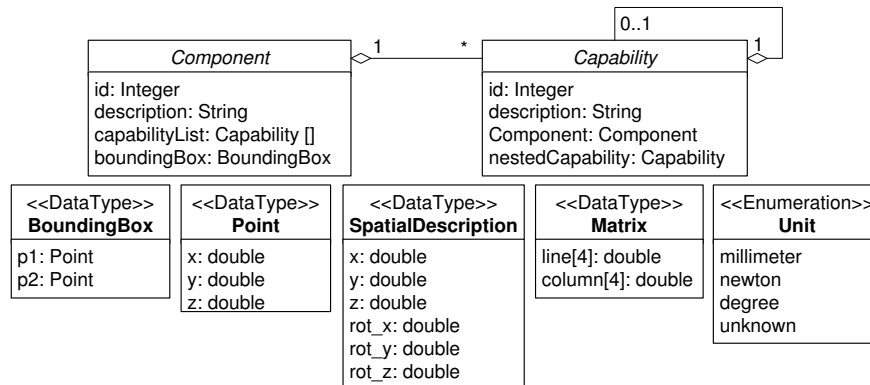
Figure 4-2: Model Elements of the Package "Common Elements"

In the diagram, the UML notations for classes and data types are used. For each class the attribute names and the data types are provided which are separated by a colon. Unless stated otherwise, the class methods are not shown in the UML diagrams in this chapter.

4.3.1. Data Types

UML defines the fundamental data types double, Integer and String. The data types double and Integer are used for numerical values, whereas String is used to retain text. Based on the former, this package defines a number of additional data types that are used throughout the model. The data type BoundingBox is used to approximate the spatial dimensions of a component. It is defined by the coordinates of two diametric corner points of the smallest box, enclosing a component. Both corners are defined as elements of the data type Point which specifies the x, y and z values of a point in the local coordinate system of a component. To define the measuring units, a number of classes of the model utilise the data type Unit. The latter is an enumeration data type which defines a set of enumeration literals for each physical unit. Furthermore, the data type SpatialDescription is used throughout the model to define the position and orientation of a component relative to another coordinate system. In essence, it holds the translational and rotational parameters to perform the coordinate transformation from one coordinate frame to another. Figure 4-3 shows an example of two such coordinate frames S_1 and S_2. Based on the spatial description of S_2, one can derive the matrices for the translation and rotation from S_1 to S_2.
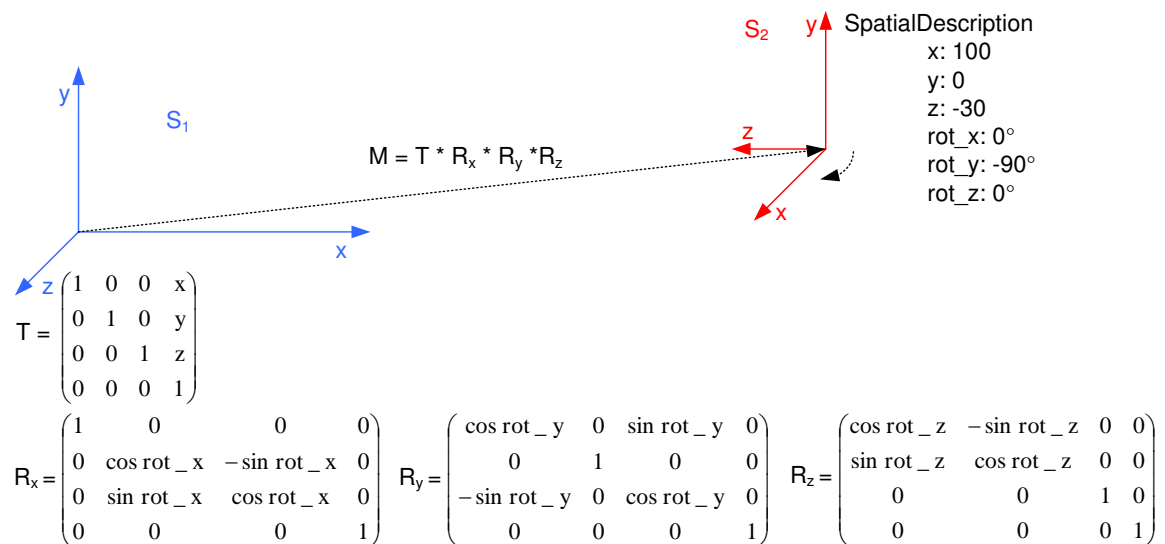
Figure 4-3: Homogeneous Coordinate Transformation Using the Data Type SpatialDescription

According to the homogenous coordinate transformation approach, these matrices are 4 by 4 matrices, which express the transformation between S1 and S2 as a matrix multiplication, resulting in matrix M. These matices are represented in the model by the data type Matrix. Based on this, a point P of the system S_2 can be expressed in coordinates of system S_1 by multiplying it with matrix M. For this, its Cartesian coordinates are converted to homogenous coordinates, using the following relation:

$$(x, y, z)^T \rightarrow (x, y, z, 1)^T \qquad \text{(Equ. 4-1)}$$

After the multiplication of matrix M with the homogenous vector of point P, the resulting 4x1 vector is converted back to Cartesian coordinates with the following function:

$$(x', y', z', 1)^T \rightarrow \left(\frac{x'}{1}, \frac{y'}{1}, \frac{z'}{1} \right)^T = (x'', y'', z'')^T \qquad \text{(Equ. 4-2)}$$

4.3.2. The Class Component

Every physical entity of a fixture that is known to the software framework is modelled as a subclass of Component, thereby providing a set of common properties. In this context, a force sensor, a clamp or an entire fixture module are represented as components. Each component has a unique numerical identifier and a description text. The most important characteristic at this abstraction level is however the association with a variable number of Capability objects. Additionally, for each component of the system its spatial dimension can be defined by setting the attribute boundingBox whose type has been described in the previous section.

4.3.3. The Class Capability

The class Capability represents a functionality of a component in the fixturing system. Its subclasses describe what a component is able to do and trigger the associated behaviour. There are matching capability subclasses for each component type. Similar to Component, the class Capability does not define any details of a particular functionality since this is modelled in its subclasses. Instead, it subsumes the commonalities among all capabilities of the data model. Firstly, the association between a capability and a component is defined in this class, thereby guaranteeing access to the component who owns the capability. Secondly, it provides the Capability subclasses in the other packages with a numerical

identifier and a textual description. Finally, as shown in Figure 4-2, a capability can own a so-called nested capability. This concept reflects the layered approach of the model and is a result of the hierarchical relationship between fixture modules and their internal devices. When a capability of a fixture module is triggered, it does not directly access the hardware to carry out the requested behaviour. Instead, it delegates the request to its nested capability which can access the associated hardware device.

4.4. Model Elements of the Package "Devices"

The elements of this package are used to represent the internal devices of a fixture module and their capabilities. For this, the module software must be configured with information about the devices, their capabilities and the software libraries to access the hardware. This can be provided in the form of XML files. Figure 4-4 presents an overview of the class structure of the package. The data types used for the class attributes are not shown in the diagram, but will be explained in the relevant sections.
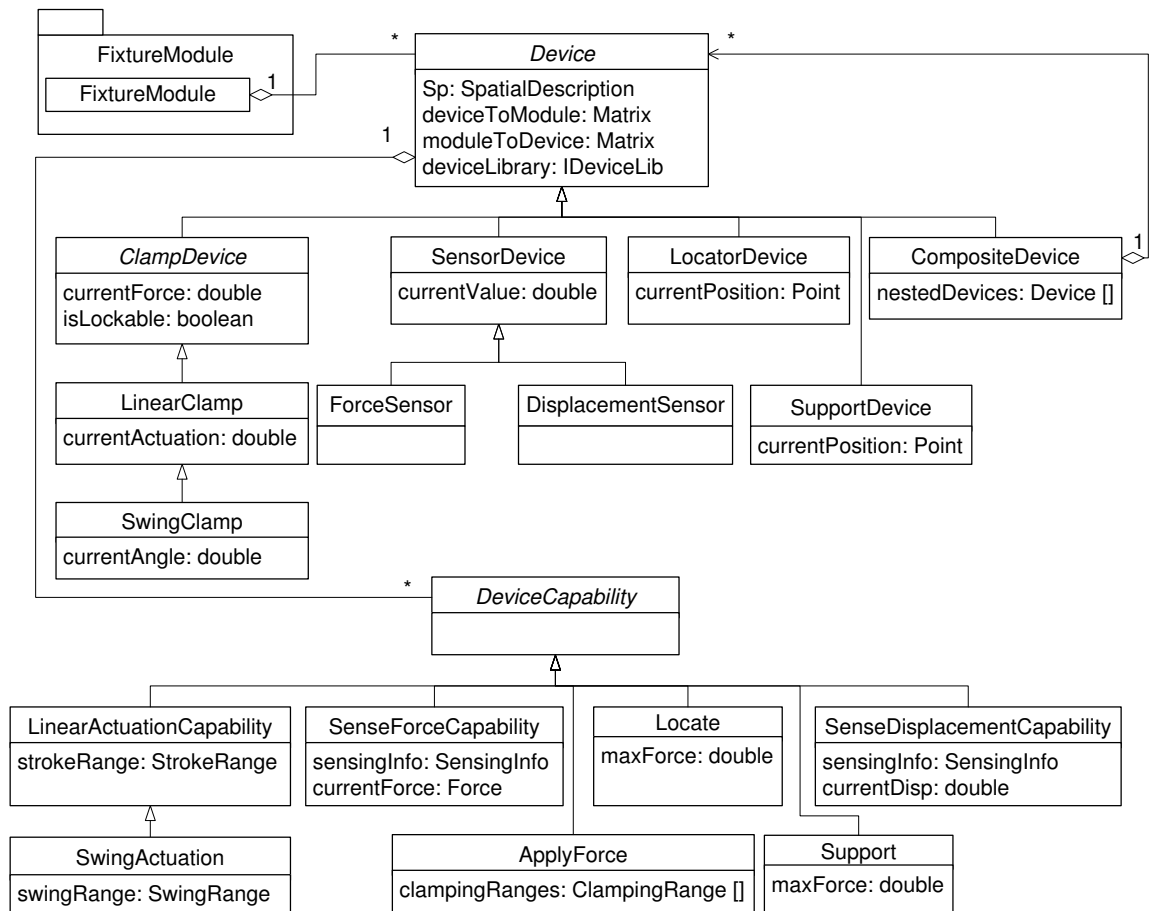
Figure 4-4: Class Diagram of the Package "Device"

4.4.1. Device Hierarchy

A device is defined as a subcomponent of a fixture module. In contrast to fixture modules, devices are not encapsulated by own active software programmes and they have no direct access to the publish/subscribe communication infrastructure described in chapter six. The class Device inherits the properties from the base class Component defined in section 4.3.2 In particular, this enables a device to be attached with an arbitrary number of capabilities which in this case inherit from the class DeviceCapability. Additionally, the Device class defines an aggregation relationship with one fixture module. In other words, one fixture module can consist of a variable number of internal devices. For the representation of the internal device structure of a fixture module, the object oriented "Composition" design pattern [104] has been adopted. According to this pattern, the parent class Device defines the class attributes and interfaces that are common to all devices. This includes a reference to the device library which contains the source code to access the hardware and the spatial description of the local coordinate frame, relative to the coordinate system of the fixture module. Based on the spatial description, the transformation matrices from the device's frame to the module's frame and vice versa can be generated and are stored in the properties deviceToModule and moduleToDevice, respectively. The common interface includes the methods to set and retrieve these attributes which are not shown in the diagram. The subclasses ClampDevice, SensorDevice, LocatorDevice and SupportDevice represent concrete device types, while the subclass CompositeDevice is used to group devices into composites. For this, the class allows to add a number of so-called nested devices which are in turn objects of the base class Device, thereby recursively creating a tree structure.

The semantics of this object hierarchy is used to express the links between devices. For example, when a force sensor is mounted on a linear clamp, the module software creates not only the objects for these devices, but also an object of the type CompositeDevice. The latter becomes the parent node of the sensor and the clamp, indicating the connection of both components. Moreover, the composite receives all capabilities of its children, thereby providing a combined view of its child nodes. The advantage of the composite pattern is that, from a software point-of-view, simple devices like an individual force sensor can be

treated the same way as complex devices which are composed of several sub-devices. This is illustrated in Figure 4-5.a which shows the object model of a fixture module consisting of a linear clamp with integrated sensors for force and displacement. Conversely, Figure 4-5.b shows a fixture module which only consists of a force sensor. In an object model diagram, an instantiated object is illustrated by a rectangle which contains the associated class name, preceded by a colon. Links between objects are depicted as lines between the rectangular frames.
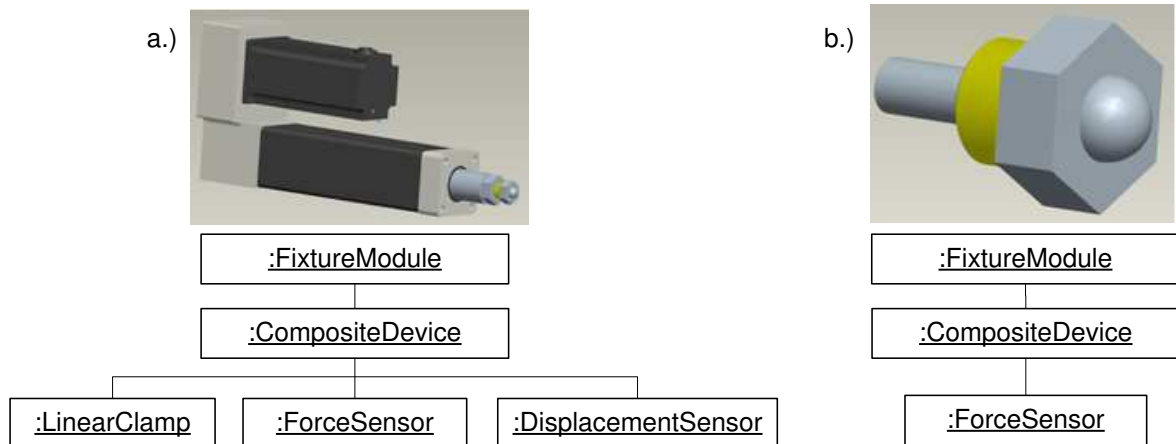
Figure 4-5: Examples for the Device Representation with the Composition Pattern

4.4.2. Device Types

The data model provides the classes for the most commonly used devices in adaptive fixtures. These include the classes SensorDevice, ClampDevice and LocatorDevice. The classes SensorDevice and ClampDevice have further child classes to reflect the variety of different kinds of these devices. Obviously, the framework does not intend to provide classes for all available device types. However, the object-oriented approach allows to enhance the data model by adding new classes. For example, for a rotary sensor an additional child class of SensorDevice can be attached, while other forms of clamping devices would require to add new subclasses of ClampingDevice.

4.4.2.1. Sensor Devices

Typical sensors used in adaptive fixturing systems are force sensors to measure reaction or clamping forces and displacement sensors. Consequently, the framework offers distinct classes for the representation of these hardware devices. To store the latest sensor reading, the base class SensorDevice provides the attribute currentValue. Since each device also

contains matching capability objects which are described in section 4.4.3, this value can be correctly interpreted by the fixture module before it is published to other subsystems. The base class also defines the interface for the method getCurrentValue() which is not shown in the class diagram in Figure 4-4, since this chapter concentrates more on the data structures. The description of the interfaces is instead the subject of chapter 6. The method is called to retrieve the current values from the sensor hardware. Internally, the classes ForceSensor and DisplacementSensor delegate the requests to the software library they are configured with, which ultimately accesses the hardware. The configuration of a device with a software library is already defined in the Device class. For this reason, the classes ForceSensor and DisplacementSensor do not add own attributes to the model. Instead, they are defined for semantic reasons.

4.4.2.2.   Clamp Devices

Similar to the sensor devices, the framework provides classes for the most common clamping types used in adaptive fixtures. The base class ClampDevice provides the attribute currentForce to store the currently exerted clamping force of the device, if the clamp is connected with a sensor to measure the force. Additionally, the Boolean attribute isLockable defines whether or not the clamp can be locked in position. If it can be locked, the clamp can also act as a locator. To model clamps based on a linear actuator the class LinearClamp is utilised by the framework. In order to store the current stroke of the linear clamp, the class defines the attribute currentActuation. As illustrated in Figure 4-6, the class SwingClamp represents clamps that perform an additional swing-in/swing-out movement during the clamping procedure.
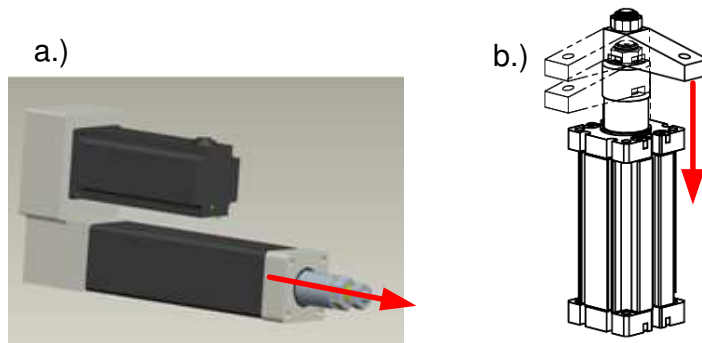
Figure 4-6: Examples for a Linear Clamp (a) and a Swing Clamp (b)

The class SwingClamp inherits from LinearClamp because the clamping process is nevertheless based on a linear actuation. It provides the attribute currentAngle for the position of the clamping arm. For the interpretation of this value, the capability class SwingActuation provides the attribute swingRange which defines the maximum swing angles in both clockwise and anti-clockwise direction (see section 4.4.3.1). Within the scope of the research, it is defined that clockwise rotations are expressed as negative angle values while rotations in counter-clockwise direction are positive. Consequently, the sign of the currentAngle attribute indicates the direction of the swing movement.

4.4.2.3.   Locator and Support Devices

The research study focuses on active devices which can be adapted before or during the clamping procedure. Passive devices like locators or supports which consist of purely mechanical structures without any kind of intelligence, do not actively participate in the clamping process. However, these devices can also be the subject of the reconfiguration procedure. For this reason, the framework provides model elements for the representation of these devices, in terms of their existence and position. Other mechanical details like material or the exact shape, are omitted as these aspects cannot be automatically reconfigured. The classes LocatorDevice and SupportDevice are used for the representation of passive devices. Figure 4-7 presents two devices that can be modelled with the described classes.
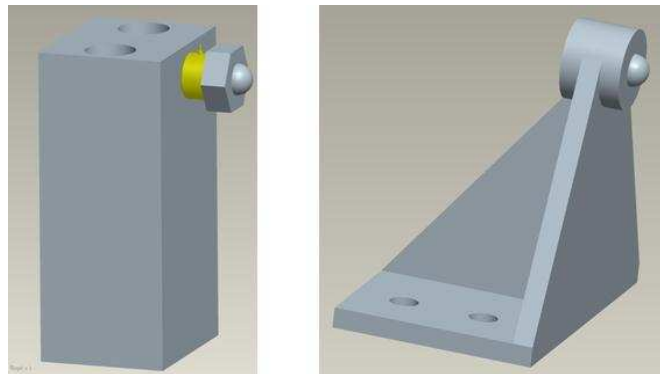
Figure 4-7: Examples for Locator Devices

Both classes have the same structure and extend their base class with an attribute for the current position of the locator/support tip. Thus, they are designed for passive devices with one contact point with the workpiece. However, locators or support elements with multiple

contact points or contact areas can also be represented, since the position property can be treated as a reference which indirectly determines the positions of other points. Furthermore, the object-oriented approach of the data model allows to add more detailed classes to capture specific locator/support devices.

4.4.3. Device Capabilities

The device capability classes are used to describe the data format and the limitations of the functionalities, a certain device provides to the fixture module. For example, a force sensor can be attached with the capability to sense force in Newton within a range of 0 to 1000N and with a resolution of 0.5N. As a consequence, clients are able to interpret the value for the current force attribute, defined in the device class. In addition to this descriptive purpose, the capability objects are used to trigger a particular functionality of a device. As described in section 6.4.2, all requests to the capabilities of the fixture modules are delegated to their nested device capabilities which have the knowledge about the interface of a particular device object for the hardware access. This delegation approach makes it possible to enhance the fixture module program with new capabilities and to exchange software objects in lower layers without affecting upper layers. Additionally, due the representation of the device capabilities as separate classes, a particular device object can be configured with exact capabilities the hardware offers. For example, some linear actuators have integrated force sensors which results in the ability to apply a certain target force while other actuators do not offer this feature. By separating the device structure from the capability classes, each device object can be linked with a list of required capability objects, based on the underlying hardware. The alternative to this approach would have been to represent the capabilities within the device classes. However, this approach would require the data model to define all theoretically possible capabilities of a device type, leading to a potentially large number of classes or obsolete class attributes. The following sections describe the device capability classes in more detail.

4.4.3.1.   Actuation Capabilities

The class LinearActuationCapability is used for clamping devices based on a linear actuator. It provides an attribute of the data type StrokeRange to describe the allowed travel

of the linear actuation in terms of the minimum and maximum stroke, the accuracy and the measuring unit.

<<DataType>> **StrokeRange**	<<DataType>> **SwingRange**	<<Enumeration>> **Axis**
min : double max: double unit : Unit resolution: double	axis: Axis cw_max : double ccw_max : double resolution : double unit : Unit	x_axis y_axis z_axis
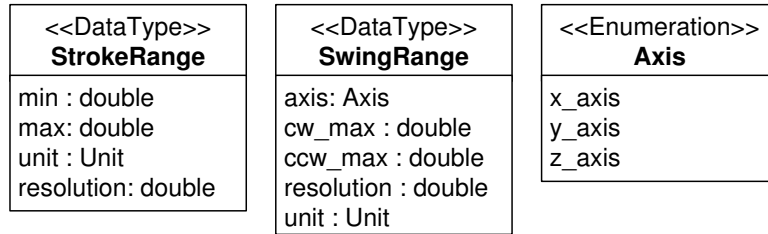
Figure 4-8: The Data Types StrokeRange, SwingRange and Axis

For swing clamps, the class SwingActuation is used which additionally provides an attribute of the data type SwingRange to describe the limitations of the swing movement of the clamping arm. The data type defines the axis around which the swing movement is performed, as well as the maximum angles in the clockwise (cw_max) and anti-clockwise (ccw_max) direction. Additionally, the accuracy and the measuring unit can be defined.

4.4.3.2. The ApplyForce Capability

This capability class is used to represent the ability of a clamp to apply a force in a certain direction. The model supports clamps that can pull, push or exert force in both directions. For this reason, the class contains a list whose entries are defined by the data type ClampingRange. This data type contains fields for the minimum and maximum achievable force, the accuracy and the measuring unit. Additionally, the field direction is used to specify whether the information accounts for the pull or the push direction. Figure 4-9 shows the UML definitions of these data types.

<<DataType>> **ClampingRange**	<<Enumeration>> **ClampingDirection**
direction: ClampingDirection minForce : double maxForce : double unit : Unit resolution : double	push pull both unknown
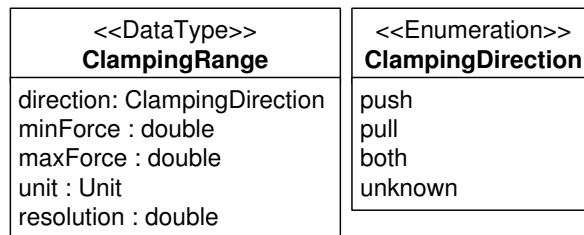
Figure 4-9: The Data Types ClampingRanges and ClampingDirection

For clamps that can act in both directions, the list contains two entries, one for the pull and one for the push direction, thereby allowing to specify different sets of information for both directions. A single entry is defined for a clamp that can exert force only in one direction. Further, to unambiguously express the clamping direction in terms of the local device

coordinate system, the definition of the local coordinate system is subject to the following restrictions:

- The clamp must act along or in parallel to the x-axis of the local coordinate system
- If the clamp can apply force in both directions, the push is defined in positive and the pull in negative direction of the x-axis
- If the clamp can only apply force in one direction (either pull or push), the clamping direction is defined in positive x-direction

The graphic below illustrates these rules. Figure 4-10.a shows a linear actuator that can push and pull. Consequently, the device coordinate system has been placed such that its x-axis defines the clamping direction when in push-mode. In Figure 4-10.b, the actuator is assumed to support only a single-acting pull-mode. Therefore, the local coordinate system has been placed such that the x-axis is pointing in the direction, the force is exerted.
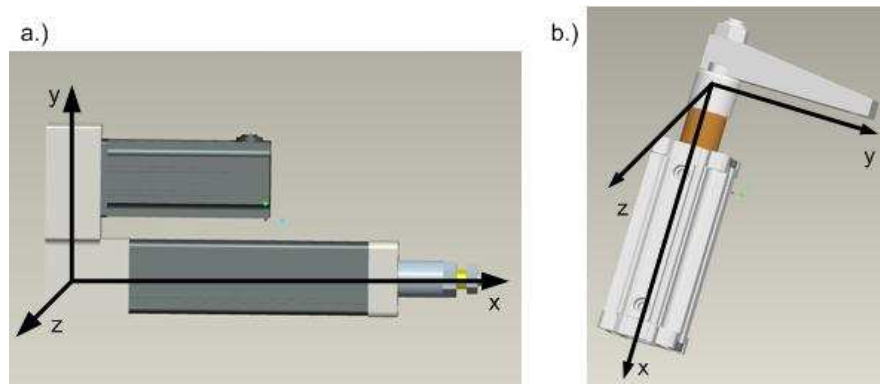
Figure 4-10: Coordinate System Definitions for Clamping Devices

4.4.3.3. The Capabilities Locate and Support

Passive elements like locators do not communicate with other devices since they lack the computational power. Nevertheless, their capabilities need to be represented by the software framework in order to assess the overall capabilities of the fixture. Additionally, self-locking clamp devices can also be used as locator or support elements. For this reason the capability classes Locate and Support have been defined. Both classes contain the attribute maxForce which specifies the maximum allowed reaction force in Newton the locator can receive without being damaged.

4.4.3.4. Sensing Capabilities

The class SenseForceCapability and SenseDisplacementCapability describe meta-information about the sensing capabilities of a device. They can be attached to device objects representing an individual force or displacement sensor. The data model is limited to force and displacement sensing. However, the model can be extended by further classes for other types of feedback. Both classes use the data type SensingInfo to describe the limitations of the sensing capability. This data type contains attributes to define the minimum and maximum measurable values, the resolution and the measuring unit. Additionally, both classes provide an attribute to hold the latest sensor sample. For the displacement sensing, the current sensor value is stored in the attribute currentDisp as a floating-point number. For the force sensing capability, the data type Force is used. This allows to store not only the current force value, but also the current clamping direction. If the capability is attached to a for force sensor that is connected with a locator, the clampDirection attribute of the data type is set to "unknown".

<<DataType>> **SensingInfo**	<<DataType>> **Force**
min: double max: double unit : Unit resolution: double	moduleId: integer clampDirection: ClampingDirection value: double
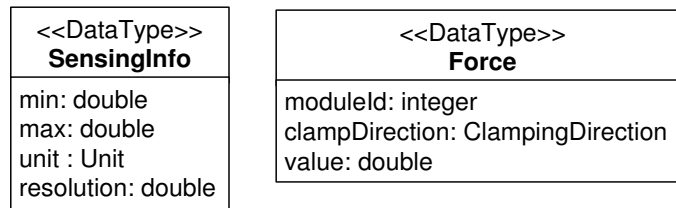
Figure 4-11: The Data Types SensingInfo and Force

4.5. Model Elements of the Package "Fixture Module"

The classes in this package are particularly important for the reconfiguration methodology and the communication infrastructure. Figure 4-12 provides an overview on the classes in this package and their relationships to other packages. The data types used for the class attributes are not shown in the diagram, but will be explained in the relevant sections.
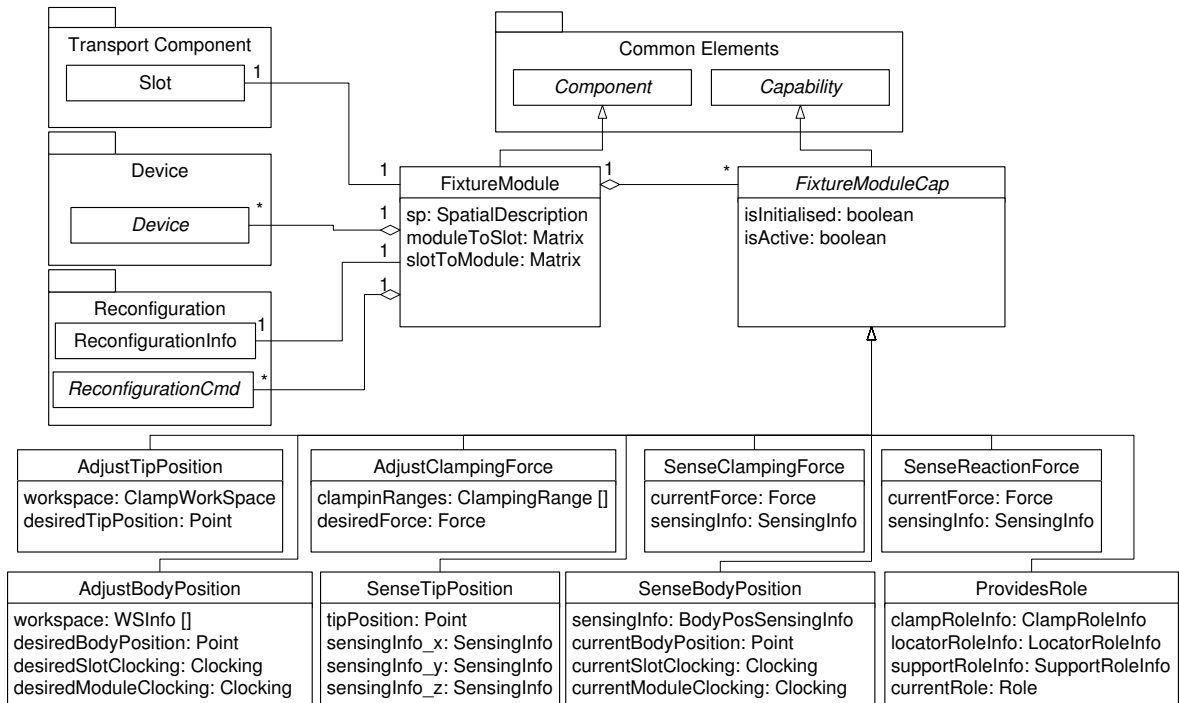
Figure 4-12: Model Elements of the Package "FixtureModule"

4.5.1. Fixture Modules

The class FixtureModule represents a component that interacts with the workpiece and is endowed with an own software program. They are regarded as the essential components in an adaptive fixturing system and are therefore addressed by the reconfiguration methodology described in chapter 5. As a result of the software program, fixture modules can actively announce their presence to the system and propagate their capabilities based on their internal devices according to the communication concept.

Physically, a fixture module is made up of sensor and actuator devices whose capabilities determine those of the entire module. This is represented in the diagram by the aggregation-relationship between the class FixtureModule and the Device-class. As a result of this hierarchy, the class FixtureModule is not limited to a specific mechanical structure and groups its devices into one functional unit which can communicate with the fixture coordinator. The classical example for a fixture module in this research is a smart clamp with integrated force and position sensors, but also a simple force sensor or a linear actuator without any feedback can be modelled as a fixture module if they are enhanced with an own local software routine that complies to the definitions of the communication

infrastructure. On the fixturing platform the fixture modules are mounted on transport components which determine their position and allow their movement when the fixture needs to be reconfigured. This is represented by the association between a fixture module and a "slot" defined in the package "Transport Component" (see section 4.6). To accomplish the reconfiguration process, a fixture module can be assigned with an object of the class ReconfigurationInfo and with a number of reconfiguration commands. These classes are described in section 4.7 while the reconfiguration methodology is described in chapter 5.

In addition to the relationships with other model elements, the class FixtureModule defines three more properties which determine its position and orientation on the fixture. This includes a property of the data type SpatialDescription that defines the translational and rotational parameters for the coordinate transformation between the module's local coordinate system and the coordinate system of its associated slot. Based on this, the transformation matrices from the module's frame to the slot's frame and vice versa can be generated and are retained in the properties moduleToSlot and slotToModule, respectively.

4.5.2. Capabilities of Fixture Modules

The capabilities of fixture modules are modelled as subclasses of FixtureModuleCap which in turn inherits from the class Capability, thereby redefining the general relationship between components and capabilities. Thus, a fixture module can only own capabilities that are subclasses of FixtureModuleCap. The reason for this restriction is that only the fixture module capabilities can communicate with the fixture coordinator while the device capabilities are exclusively visible to the fixture module.

The capability objects in this package serve three purposes. Firstly, by attaching them to the fixture module object, the latter can be enhanced in a flexible way with functionalities and additional properties like the ability to exert a clamping force or to feed back the current position of the actuator tip. Without attaching capability objects, the class FixtureModule is merely an empty shell. Consequently, the approach allows to reuse the class for a variety of different hardware setups by attaching it with different capability objects. Secondly, only

the fixture module capability classes contain the logic to communicate via the publish/subscribe concept. Consequently, the capability objects constitute the interface to trigger a particular behaviour of the module. Received requests are delegated to the nested device capability objects until the hardware is accessed. Thirdly, the capability classes describe the characteristics and limitations of the related functionality in order to allow other subsystems to utilise the fixture module in a meaningful way. When the fixture module capabilities are forwarded to fixture coordinator or to other subsystems, they utilise this information to interpret the data coming from the module.

The framework defines eight fixture module capabilities, reflecting the most common functionalities in a fixture. However, the object-oriented approach allows programmers to extend this hierarchy with other classes if required. For example, if a fixture module containing a temperature sensor is introduced, a new subclass SenseTemperature can be introduced without affecting the overall concept. In the following sections the fixture module capabilities are described in more detail.

4.5.2.1.   The Capability AdjustTipPosition

The tip position of a fixture module is defined as the point where it touches the workpiece. Thus, this capability is attached to the fixture module if it contains a clamp device that is able to actuate to a certain position. The coordinates of this point are relative to the local coordinate system of the fixture module.

<<DataType>> **ClampWorkspace**
strokeRange_x : StrokeRange strokeRange_y : StrokeRange strokeRange_z : StrokeRange swingRange : SwingRange
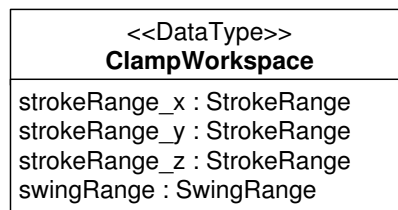
Figure 4-13: The Data type ClampWorkspace

The class provides an attribute of the data type Point for the desired tip position which can be set by other systems in order to trigger the actuation, as described in chapter six. Additionally, the property workspace specifies the area which can be reached by the actuator tip of the module, using the data type ClampWorkSpace. This is a structural data type containing the allowed stroke of the actuator along the x, y and z axis of the module

and the swing range. For this the data types StrokeRange and SwingRange are used which have been described in section 4.4.3. Below an illustrative example is provided of fixture module consisting of a linear actuator with a maximum travel of 60mm and a resolution of 0.01mm. Since the module does not allow any swing-movement, the clockwise and anti-clockwise value in the attribute swingRange is set to zero.


```
- workspace
    - strokeRange_x
            -min: 300
            -max: 360
            -unit: mm
            -resolution: 0.01
    - strokeRange_y:
            -min: 0
            -max:0
            -unit: mm
            -resolution:0
    - strokeRange_z:
            -min: 0
            -max: 0
            -unit: mm
            -resolution: 0
    -swingRange:
            -axis: unknown
            -cw_max: 0
            -ccw_max: 0
            -unit: degrees
            -resolution: 0
```
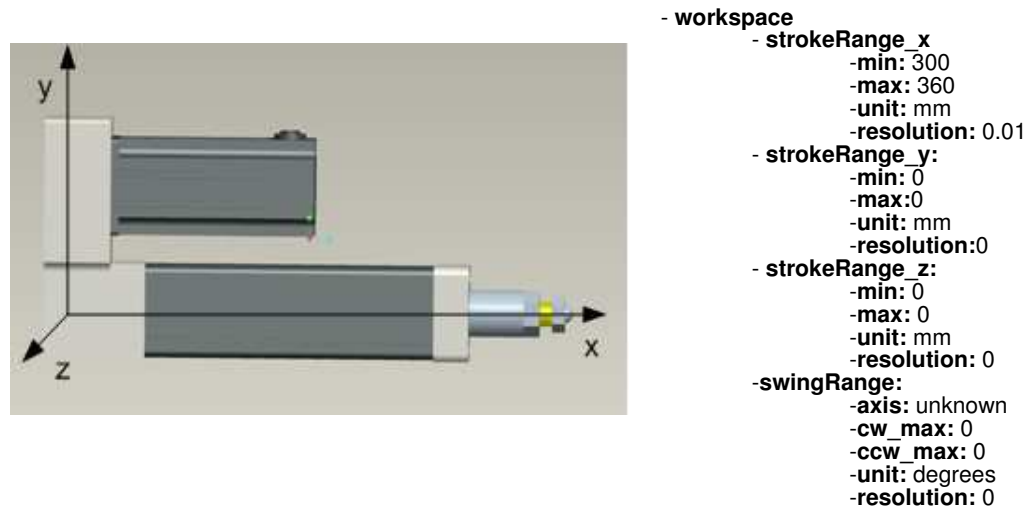
Figure 4-14: Example Instantiation of the AdjustTipPosition Capability

4.5.2.2.   The Capability SenseTipPosition

If a fixture module has a sensor device for the positional feedback of its tip position, this capability is instantiated. The coordinates of the currently measured tip position are stored in the property tipPosition and defined relative to the local coordinate system of the module. Further, the class provides three additional attributes of the data type SensingInfo to allow other systems to interpret the x, y and z component of current tip position value and to inform them about the limitations of the sensing capability. Details of the data type SensingInfo have been presented in section 4.4.3.

4.5.2.3.   The Capability AdjustBodyPosition

The body position refers to the position of the fixture module on the fixturing platform and is defined relative to the global coordinate system. Since fixture modules are mounted on the transport components, the body position and orientation depends on the:

- The position and orientation of the transport component relative to the global coordinate system
- The position and orientation of the slot on the transport component

- The position and orientation of the module relative to the associated slot

The fixture coordinator creates this capability class automatically when the operator links a fixture module with a slot on a transport component as described in section 5.2.3. Based on the capabilities of the transport component, the workspace of the fixture module is derived and represented in the AdjustBodyPosition class using the data type WSInfo. The latter defines the allowed linear movements of the fixture module in global coordinates using the data type StrokeRange.

<<DataType>> **WSInfo**	<<DataType>> **ClockingRanges**	<<DataType>> **ClockingRange**	<<DataType>> **Clocking**
slotId: Integer linearRange_x: StrokeRange linearRange_y: StrokeRange linearRange_z: StrokeRange slotClockingRanges: ClockingRanges moduleClockingRanges: ClockingRanges	clockingRange_x: ClockingRange clockingRange_y: ClockingRange clockingRange_z: ClockingRange	cw_max: double ccw_max: double unit: Unit resolution: double	rot_x : double rot_y : double rot_z : double

Figure 4-15: Data Types related to the AdjustBodyPosition Capability

Additionally, the element slotClockingRanges contains the allowed rotation of the associated slot around its axis. Similarly, if a module can be rotated on the slot, the attribute moduleClockingRanges defines the allowed rotation. The data type ClockingRange follows the same concept as the data type SwingRange (see section 4.4.3).

Apart from representing the workspace of the fixture module, the capability is used by the fixture coordinator to trigger the repositioning of the fixture modules during the reconfiguration procedure. In this context, the fixture coordinator can update the attributes desiredBodyPosition, desiredSlotClocking and desiredModuleClocking with the target values. For this, the data type Clocking is used to indicate the desired angles in clockwise and counter-clockwise direction. As described before, negative values indicate a clockwise rotation while positive angles signal a counter-clockwise rotation. These values are published by the capability according to the communication concept and are ultimately received by the software objects of the transport component. The transport component is responsible for the repositioning of its slots, thereby changing the position of the associated modules. The data exchange between fixture modules and transport components in described in section 6.3.1.

4.5.2.4. The Capability SenseBodyPosition

Similar to the previous class, the capability SenseBodyPosition is automatically created by the fixture coordinator software when the operator connects a fixture module with a slot on a transport component. The capability class is used to retrieve and represent the current position and orientation of the fixture module on the platform. For this, the class provides the attributes currentBodyPosition, currentSlotClocking and currentModuleClocking. To allow the correct interpretation of these values, the attribute bodyPositionSensingInfo is used whose data type definition is depicted in Figure 4-16.

```
<<DataType>>
BodyPositionSensingInfo
───────────────────────────
posX: SensingInfo
posY: SensingInfo
posZ: SensingInfo
moduleClockingX: SensingInfo
moduleClockingY: SensingInfo
moduleClockingZ: SensingInfo
slotClockingX: SensingInfo
slotClockingY: SensingInfo
slotClockingZ: SensingInfo
```
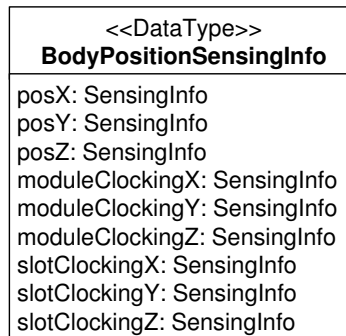
Figure 4-16: Relevant Data Types for the Capability SenseBodyPosition

4.5.2.5. The Capability AdjustClampingForce

If the fixture module contains an actuator device with the ability to apply a clamping force, this capability is created for the fixture module, based on the ApplyForce capability of the device. The class allows other subsystems to trigger the clamping behaviour of the module via the publish/subscribe communication infrastructure. To specify the target force and clamping direction, the class attribute desiredForce must be set with the desired values. Similar to the class ApplyForce, the limitations of the functionality are specified using the data type ClampingRanges (see section 4.4.3 for further details).

4.5.2.6. The Capabilities SenseClampingForce and SenseReactionForce

The capability class SenseClampingForce is attached to a fixture module which contains a clamping device and a force sensor to measure the force at its actuator tip. The class has the property currentForce to represent the current clamping force value and direction. The attribute is defined using the data type Force which has been described in section 4.4.3. Additionally, information for the interpretation of the sensor value is provided in the class

attribute sensingInfo whose data type has also been described before. The class SenseReactionForce has the same structure as the previous capability. However, it is attached to a fixture module that acts as a locator and contains a force sensor to measure the experienced reaction force at its locator tip. If a fixture module consists of a lockable actuator which can act as a clamp and a locator, both capability classes are instantiated for the module. During the operation, one of them is inactivated, depending on the current role of the module.

4.5.2.7. The Capability ProvidesRole

Based on the internal devices, a fixture module can support different roles during the clamping procedure, namely the roles clamp, locator and support. This classification is represented in the model by the enumeration data type Role which defines three enumeration literals for the roles. Moreover, the software framework allows modules to change their role for different fixture setups. For example, a fixture module can act as a clamp for one workpiece and as a locator for another workpiece, provided that it can lock in position and withstand the estimated reaction forces.

To indicate the supported roles the capability class ProvidesRole provides the three attributes clampRoleInfo, locatorRoleInfo and supportRoleInfo whose data types are listed below.

<<DataType>> **ClampRoleInfo**	<<DataType>> **LocatorRoleInfo**	<<DataType>> **SupportRoleInfo**	<<Enumeration>> **Role**
isSupported: boolean	isSupported: boolean maxForce: double	isSupported: boolean maxForce: double	Clamp Locator Support

Figure 4-17: Data Types Related to the Capability ProvidesRole

Each data type contains a Boolean element isSupported which is set to true, if the fixture module supports a particular role. The data type ClampRoleInfo does not provide any further details, because the relevant parameters of the clamping functionality are already represented in other capability classes of the fixture module, such as AdjustClampingForce and AdjustTipPosition. For the locator and support role, the attribute maxForce can be used to specify the maximum allowed reaction force in Newton, the module can experience without being damaged. Finally, the attribute currentRole is provided to retain the current

role of the associated fixture module. This attribute is only used in the software of the fixture coordinator, while the software of the fixture modules remains unaware of the current role.

4.6. Model Elements of the Package "Transport Components"

Transport components are defined as those parts of the fixture on which the fixture modules can be mounted and repositioned. The term transport component is neither a traditional term used in the fixturing domain nor is it limited to a specific geometric structure. Instead, transport components and fixture modules are abstractions that modularise a fixturing system into two functional groups: fixture modules which interact with the workpiece and transport components which allow the repositioning of the former during the reconfiguration procedure. Figure 4-18 presents a UML class diagram for the package and illustrates its dependencies to the other packages.
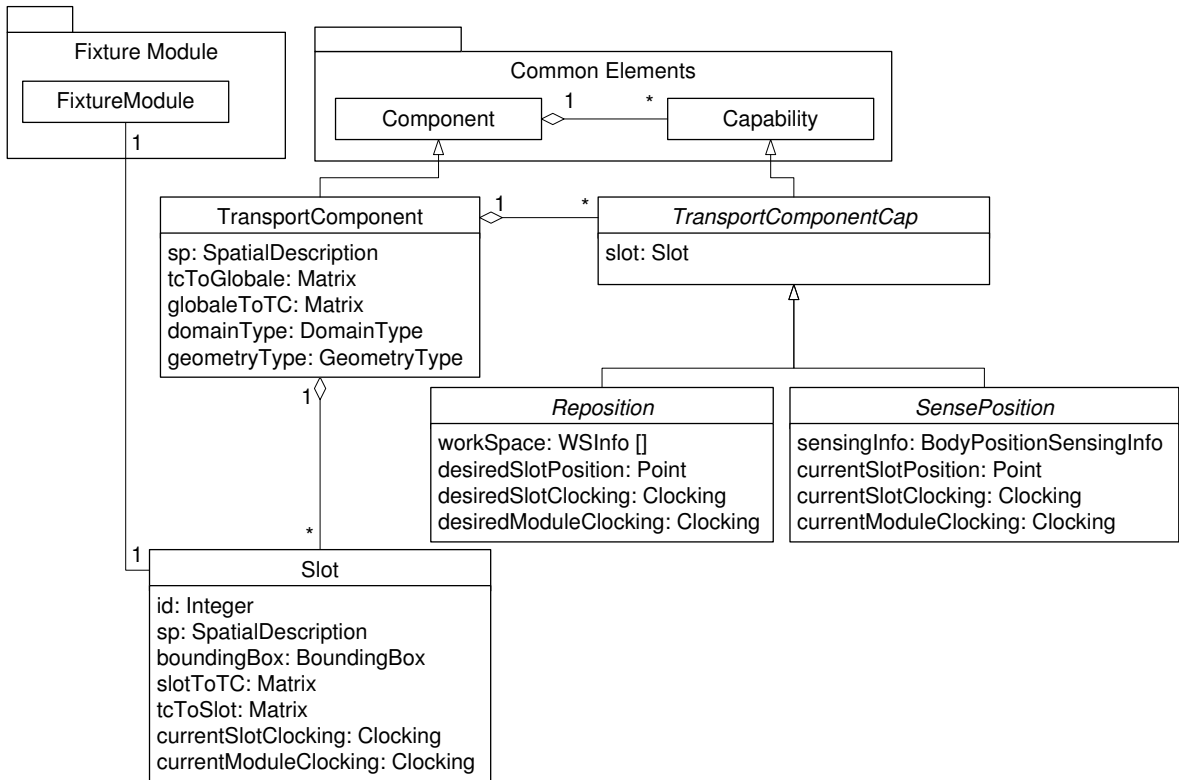
Figure 4-18: Overview of the Package "Transport Component"

4.6.1. Transport Components

The class TransportComponent inherits from the class Component and can therefore be attached with multiple capabilities, which inherit from the class TransportComponentCap. The position and orientation of the transport components is assumed to be constant during the operation of the fixture. This means, these components are not subject to the reconfiguration procedure. The framework can be configured with the position and orientation of a transport component and retains this information in the class attribute of the data type SpatialDescription. Based on this, the transformation matrices for the conversion from the global coordinate system to the local coordinate system of the transport component and vice versa can be generated as described in section 4.3.1. The matrices are stored in the class attributes tcToGlobale and globaleToTC.

Typical examples for transport components are the linear guides presented in section 3.5.1 which allow the continuous movement of the attached fixture modules. In contrast to this, a base plate with mounting holes can be regarded as a transport component which allows the positioning of the modules in two dimensions. However, in this scenario the modules cannot be repositioned continuously, but are limited to the positions of the mounting holes. Other types of transport components, such as magnetic base plates, can alleviate this restriction and provide a continuous 2D workspace for the modules. These examples indicate that there are great differences in terms of the shapes, geometries and the mechanical methods for the mounting and moving of fixture modules on the transport component. However, at the same time a number of common functional characteristics can be identified. Firstly, transport components can be grouped according to the degree of freedom they allow for the movement of the fixture modules. Secondly, there is a distinction between transport components that allow continuous movement and those where the modules can be positioned in a discrete number of locations. These two aspects are reflected by the attributes geometryType and domainType of the class TransportComponent whose data type definitions are provided unterhalb. The former specifies whether the transport component allows the positioning of the modules along a line (one dimension), on a plane (two dimensions) or in space (three dimensions). For this the enumeration data type GeometryType is used which is shown oben. The second attribute specifies whether the

transport component allows a continuous relocation of the fixture modules or if the possible positions are restricted to a discrete number of locations.

<<Enumeration>> **DomainType**	<<Enumeration>> **GeometryType**
Continous Discrete	OneDimensional TwoDimensional ThreeDimensional
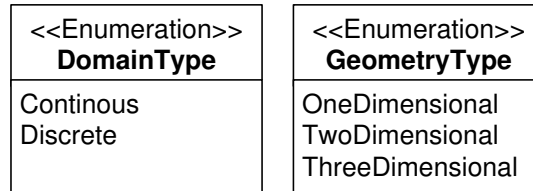
Figure 4-19: The Data Types DomainType and GeometryType

4.6.2. Slots

A slot is defined as a part of the transport component which can be connected with one fixture module at a time. Consequently, the number of slots determines the possible number of fixture modules on a transport component. Practical examples for slots are the movable carriers of the rail-based system, presented in section 3.5. By linking a fixture module with a slot object, the framework becomes aware of the position of the fixture module on the fixturing platform. This is because the position and orientation of the slots is defined relative to the coordinate system of the transport component whose posture in the global coordinate system is known. Thus, when the position of a particular fixture module is requested, the position of its related slot is used. Similarly, when a fixture module needs to be relocated, the position of the slot is changed.

Each slot on a transport component has a numerical identifier and defines an own local coordinate system whose position and orientation is described relative to the coordinate frame of the associated transport component. For this, the data type SpatialDescription is used which contains the rotational and translational parameters for the generation for the transformation matrices between both coordinate frames. These matrices are stored in the class attibutes slotToTC and tcToSlot. Figure 4-20 illustrates the spatial description of the local slot coordinate frame (blue), relative to the coordinate frame of the transport component (red). When a slot is moved during the reconfiguration procedure, its spatial description and the associated transformation matrices need to be updated in order to reflect the repositioning. Additionally, the framework allows to represent slots whose orientation on the transport component can be changed by rotating them around their coordinate axis.

This ability is termed "clocking" within the scope of the thesis. The allowed clocking range can be specified in the Reposition-capability class, described in section 4.6.3.

SpatialDescription
x: 200
y: 50
z: -50
rot_x: 0
rot_y: -90
rot_z: 0
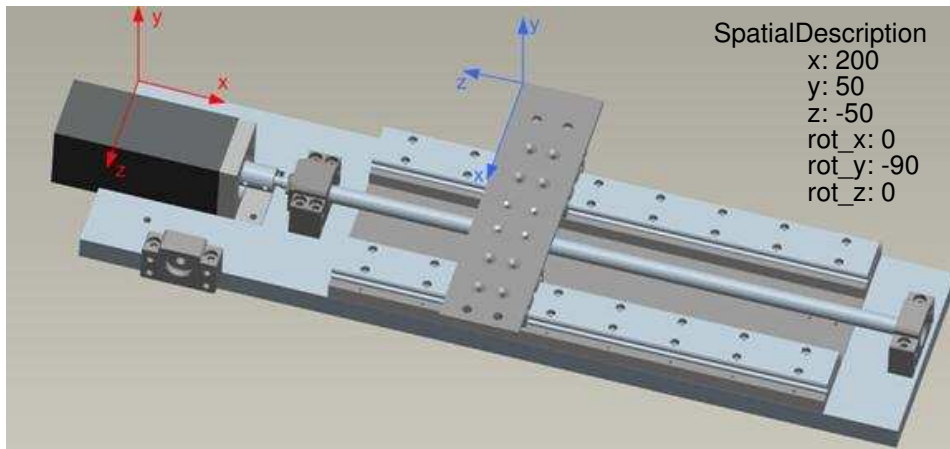
Figure 4-20: Instantiation Example of a Slot on a Transport Component

The current clocking value is retained in the Slot-object using the attribute currentSlotClocking, defined by the data type Clocking. The value for current clocking must be within the limits set by the clocking range which is defined in the Reposition-class. Further, the clocking values are interpreted as offsets from the original spatial description of the slot. Negative values indicate a clockwise rotation around an axis, while positive values indicate an anti-clockwise rotation. At the same time, the clocking values influence the orientation of the slot on the transport component and hence change the rotational parts of the spatial description attribute. This is illustrated in the example shown in Figure 4-21. The drawing shows a slot on a transport component which allows the clocking of -/+45° around its y-axis, beginning from its initial orientation as indicated by the dotted line. In the current setup, the slot is rotated around its y-axis by 15° in clockwise direction. This value is retained in the currentSlotClocking attribute and it is also reflected in the rotational part of the slot's spatial description. The separation of the current clocking values from the current spatial description allows to determine the original (default) orientation of the slot at all times, as well as the currently allowed clocking in clockwise and counter-clockwise direction. Hence, by subtracting the current clocking values from the allowed clocking values one can derive that the slot in the displayed setup can still be rotated by 30° in clockwise and 60° in counter-clockwise direction. By subtracting the current clocking values from the spatial description, the original orientation of the slot around the y-axis can be calculated as 0°.

ClockingRange_x:
  cw_max: 0°
  ccw_max: 0°
ClockingRange_y:
  cw_max: -45°
  ccw_max: +45°
ClockingRange_z:
  cw_max: 0°
  ccw_max: 0°

SpatialDescription
  x: 50mm
  y: 0mm
  z: 100mm
  rot_x: 0°
  rot_y: -15°
  rot_z: 0°
CurrentSlotClocking:
  rot_x: 0°
  rot_y: -15°
  rot_z: 0°

Information for y-axis:
Allowed Clocking clockwise: -45° - (-15°) = -30°
Allowed Clocking counter-clockwise: 45° - (-15°) = 60°
Original Orientation: -15° - (-15°) = 0°
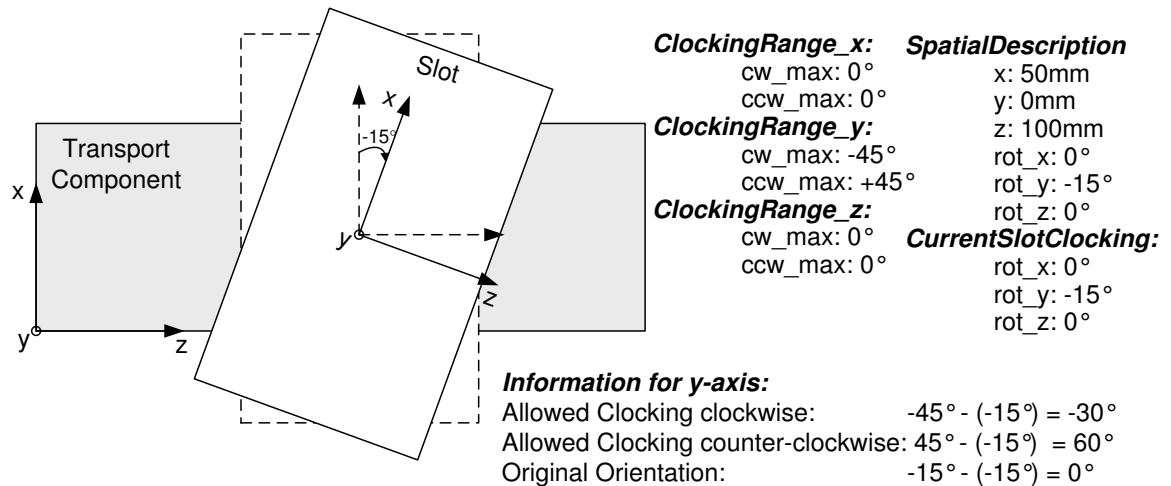
Figure 4-21: Example Instantiation of Slot with Clocking

Additionally, the framework supports fixture setups where the connection between a slot and a fixture module allows the clocking of the module on the slot. An example for this could be a base plate with mounting holes which allow the rotation of the fixture modules whilst remaining in the slot. For such cases, the class attribute currentModuleClocking is provided which follows the same principle as the clocking of the slot. The allowed clocking range of the module must be provided by the operator when the a fixture is connected with a slot. This information is retained in the Reposition-capability class that is associated with the transport component.

4.6.3. Capabilities of Transport Components

The capabilities of a transport component are modelled in the class TransportComponentCap and its subclasses which specify the limitations for the repositioning of the slots on the transport component and the position feedback functionality. Additionally, the capability classes are linked to the publish/subscribe architecture which allows the communication of the current and desired slot position and orientation. The class contains a reference to a particular slot on the transport component. Consequently, the capability objects are ultimately related to the slots which are connected with the fixture modules. Based on this link, the combined workspace of the associated fixture module can be determined. Since the research work concentrates on the reconfigurability of fixture modules, the model does not decompose the transport components into sub-devices with an own set of capabilities. Instead, the transport

component capabilities are limited to the repositioning of the slots and the feedback of the current slot positions.

4.6.3.1. The Capability Reposition Capability

The class Reposition stands for the capability of a transport component to change the position of a certain slot within a specified workspace. Consequently, if a slot is linked to a fixture module, the latter can be repositioned accordingly. Since a transport component can have multiple slots, it can be attached with potentially many Reposition-objects.

During the reconfiguration procedure this class is used to retrieve the desired position of the slot on the transport component, the desired slot clocking and the desired clocking of the fixture module on the slot from the fixture coordinator software. For this purpose, the attributes desiredPosition, desiredSlotClocking and desiredModuleClocking are provided. The workspace for the repositioning of a slot is described using an attribute of the data type WSInfo which contains the allowed linear movements and the clocking ranges for the slot and the fixture module (see section 4.5.2). Because the module clocking depends on the connected fixture module, the value for the module clocking range is set to a default of 0 degrees, as long as the slot is unlinked. The operator can update these values when a slot is linked with a fixture module. Furthermore, the domain type of the transport component influences the workspace description in this class. For this reason, the class Reposition contains a list of workspace elements. For transport components which support the continuous repositioning of their slots, one workspace entry is created. In contrast, for discrete transport components multiple workspace entries are defined, one for each possible position on the transport component. Figure 4-22.a shows a transport component with a continuous domain type which results in a workspace defining the minimum and maximum positions of the slot on the transport component. In this example, the slot does not allow any reorientation. As a consequence, the clocking range specifies a value of 0° for each axis in clockwise and counter-clockwise direction.
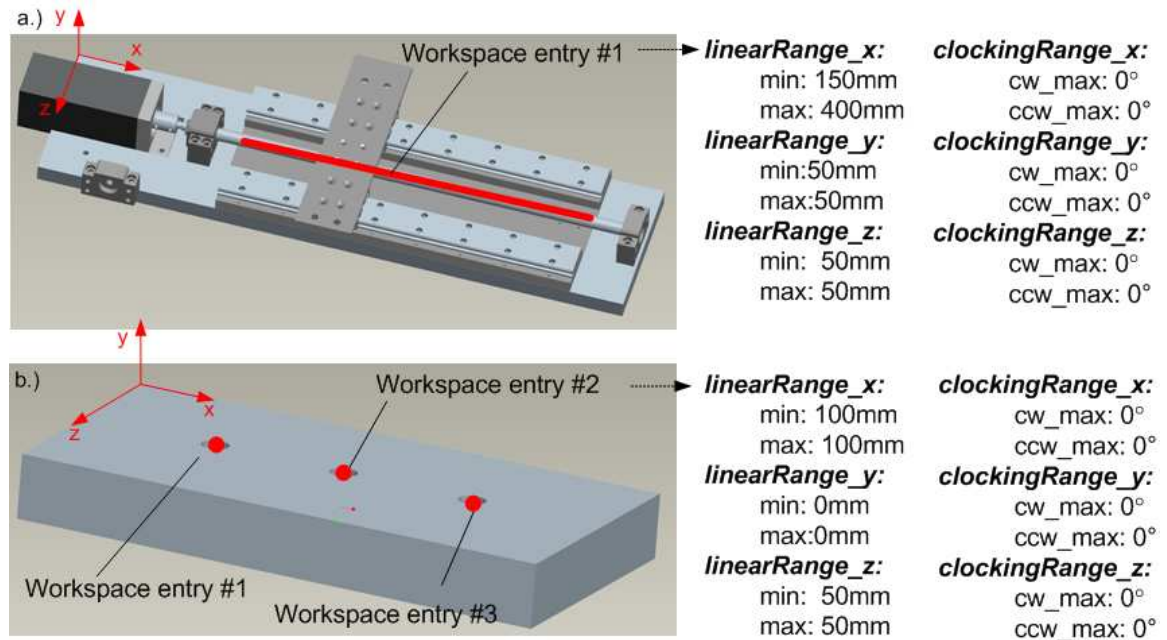
Figure 4-22: Workspace Definitions for Slots on Continuous Transport Components (a) and Discrete Transport Components (b)

On the other hand, Figure 4-22b shows a discrete transport component which does not allow any linear movements of the slots. However, fixture modules can be mounted in three different positions, resulting in three workspace entries for the slot object. The minimum and maximum values of each entry are equal, thereby defining a point rather than a range. In the drawing this is illustrated using the workspace entry two.

4.6.3.2. The Capability SensePosition

This class represents the ability of the transport component to feed back the position and orientation of a particular slot. Additionally, the values for the current module clocking can be fed back. For this purpose, the class provides the attributes currentSlotPosition, currentSlotClocking and currentModuleClocking. To allow the correct interpretation of these values, the attribute bodyPositionSensingInfo is used whose data type definition was already described in section 4.5.2.

4.7. Model Elements of the Package "Reconfiguration"

The model elements in this package are required during the reconfiguration procedure to represent the pre-defined fixture design parameters and the individual steps to convert the

current fixture setup into the desired configuration. Figure 4-23 provides a UML class diagram of the package. The following sections describe the depicted classes in more detail.
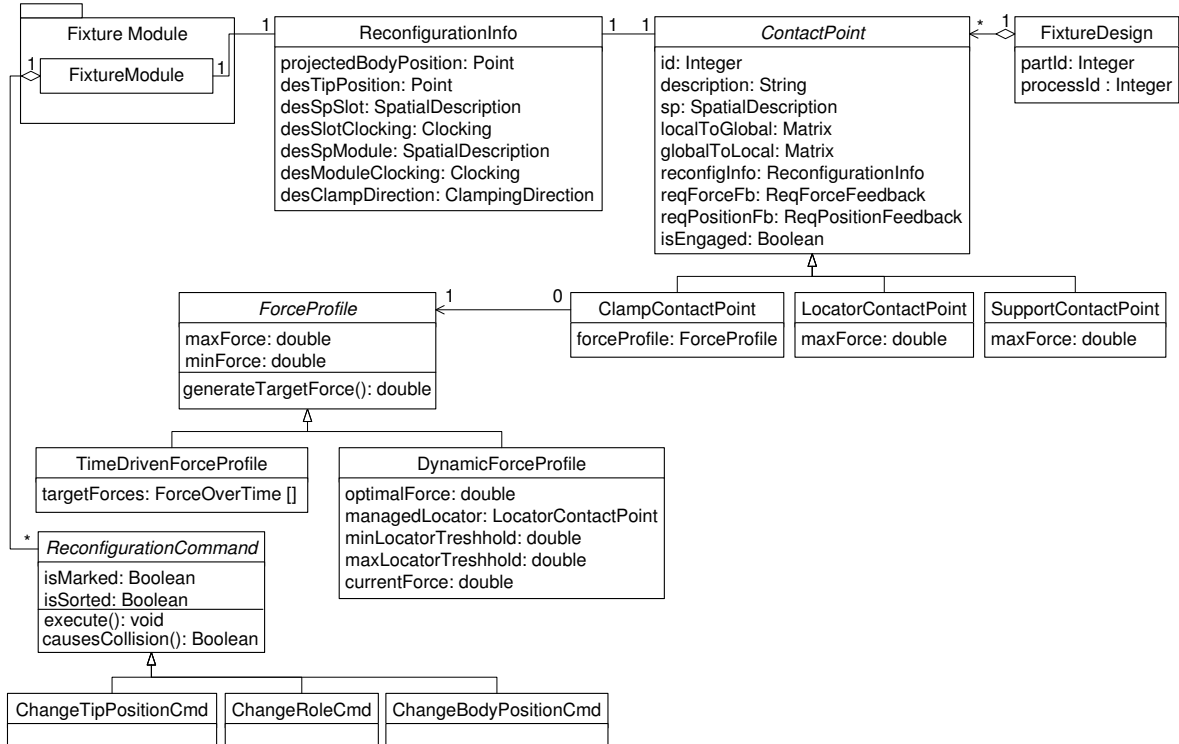
Figure 4-23: Class Diagram of the Package "Reconfiguration"

4.7.1. Fixture Design Information

A key assumption of the research study is the availability of pre-defined fixture design parameters for each workpiece and process. This information can be provided in form of a data base or through other means, such as configuration files. For the object-oriented representation of the design information, the data model defines the class FixtureDesign which contains the the numerical identifiers the associated workpiece and the manufacturing process. Additionally, it can be attached with a variable number of objects inheriting from the base class ContactPoint. The latter contains the design criteria for each point, the fixture is in contact with the workpiece. This information is limited to hardware-independent parameters such as the position of the contact point in global coordinates or the required clamping force. Hardware-specific details such as the use of a vendor-specific device model or a certain clamping technology like pneumatic or electro-mechanical mechanisms are not defined in the contact point information. This approach renders the

framework independent from particular hardware and allows the operator to upgrade an existing fixture with new devices as long as the design parameters are satisfied.

As it can be seen in the class diagram, each contact point has a numerical identifier and a textual description. Its local coordinate system is specified relative to the global coordinate frame by the attribute of the data type SpatialDescription. Based on this, the matrices for the coordinate transformation between the local and the globale coordinate systems can be calculated and stored in the attributes localToGlobal and globalToLocal. The local coordinate system determines the position where the fixture module shall contact the workpiece. The x-axis of the local coordinate system is directed towards the workpiece. Figure 4-24 illustrates the contact point definition for a simple workpiece. Contact points with a filled circle indicate clamps whereas unfilled circles indicate locator elements.
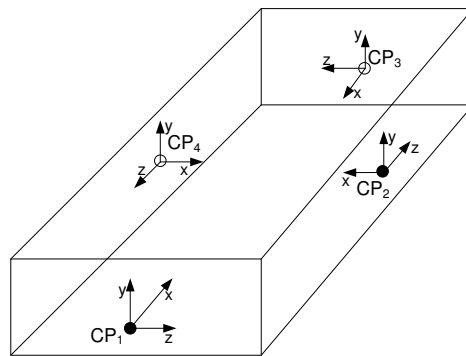
Figure 4-24: Illustration of Contact Points

To specify the feedback requirements of a contact point, the attributes reqForceFb and reqPositionFb are provided whose data types are defined unterhalb. Both structures contain a Boolean element defining whether or not a particular feedback functionality is required. If this is the case, the element sensingInfo contains further details which must be satisfied.

<<DataType>> **ReqForceFeedback**	<<DataType>> **ReqPositionFeedback**
isRequired: Boolean sensingInfo: SensingInfo	isRequired: Boolean sensingInfo: SensingInfo
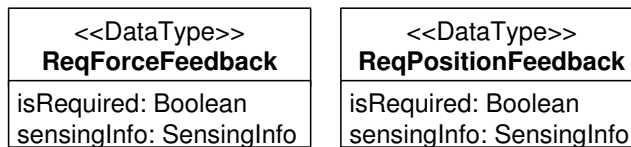
Figure 4-25: Data Types to Define the Requirements for the Force and Position Feedback

Finally, the Boolean attribute isEngaged can be used to declare a contact point as inactive in a particular design by setting its value to false. Hence, if a fixture design requires less contact points than others, it can declare a contact point as not engaged. During the reconfiguration procedure, this contact point will be assigned to one of the fixture modules.

Since the contact point defines the target position for the module, it is ensured that it is moved out of the way and remains inactive for the duration of the process.

The essential step during the reconfiguration methodology is the process of matching the contact points with the available fixture modules from the physical setup. This link is represented by a reference to an object of the class ReconfigurationInfo which contains references to both the FixtureModule and the ContactPoint object. The class stores all required information for the reconfiguration of the modules in its attributes which are acquired during the procedure, described in section 5.3.3. This includes the target values for the body position of the module, the tip position, the clamping direction, the spatial descriptions for the module and the slot, as well as their clocking values. To indicate whether a fixture module shall act as a clamp, locator or support element during the operation, the data model defines three subclasses, inheriting from ContactPoint. The classes LocatorContactPoint and SupportContactPoint have the same structure, since in the context of adaptive fixturing both roles define passive elements. These classes provide the means to specify the maximum amount of force, a matching fixture module must be able to withstand without being damaged. For contact points that require a clamp, the class ClampContactPoint is provided which can be configured with a reference to a force profile, defining the behaviour of the clamp during the operation.

4.7.2. Force Profiles

The force profiles are modelled with the child classes inheriting from the base class ForceProfile. The latter defines two attributes for the minimum and maximum force values in Newton, the associated clamp can exert during the clamping procedure. Furthermore, these classes implement the object-oriented "Strategy" design pattern [104]. The advantage of the Strategy-pattern is the ability to change algorithms at run-time without the need for recompiling the software. In the context of this research, it has been applied to allow the framework to be configured with different kinds of force profiles in a flexible way. According to the structure of the design pattern, the base class ForceProfile defines a common interface generateTargetForce() which is called to retrieve the force value in Newton for the associated fixture module during the operation of the fixture. However, the

base class does not specify how the target force is calculated. Instead, the interface is implemented differently in the child classes. The class TimeDrivenForceProfile can be used to define a profile that specifies the clamping force depending on the elapsed time of the manufacturing process. For this, the class has a list with entries of the data type ForceOverTime. This data type contains an element for the force magnitude in Newton and an element specifying a point in time in milliseconds. This allows the definition of step-like profiles over time as shown in Figure 4-26.
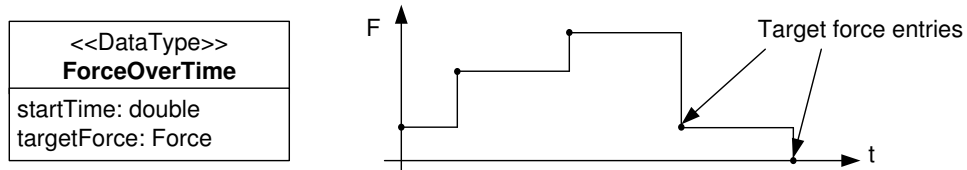
Figure 4-26: The Data Type ForceOverTime

For cases where the dynamic adaptation of the clamping force in response to the measured reaction forces acting on the locators is required, the class DynamicForceProfile can be used. The class allows to specify a locator whose reaction forces determine the magnitude of the target clamping force. Additionally, an optimal clamping force can be specified in Newton which the associated fixture module tries to approach during the operation. However, if the reaction force on the associated locator falls below a certain threshold as specified by the attribute minLocatorThreshhold, the clamping force is increased to ensure the workpiece remains in contact with the locator. Conversely, if the reaction force exceeds the threshold specified in the attribute maxLocatorThreshhold, the clamping force is decreased to prevent workpiece deformation. This class interprets the minimum and maximum force values from the base class as a band in which it is allowed to adapt the clamping force. Hence, the force adaptation described above is limited by these values as illustrated in Figure 4-27.
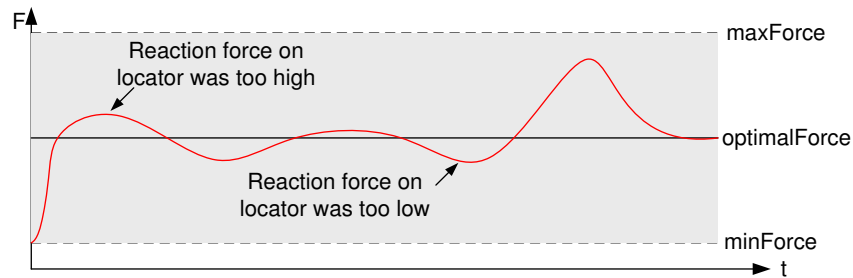
Figure 4-27: Illustration of a Dynamic Force Profile

The diagram shows a possible curve for the clamping force as the fixture module tries to approach the optimal force values, while reacting to the reaction forces of the managed locator. A similar approach has been presented by Wang et al. [50] as reported in the literature review (see section 2.2.5). However, the aim of this research is not to describe the generation of dynamic force profiles. Instead, this section shows how the structure of the data model supports a wide variety of different force profile approaches. As a result of the Strategy-pattern, further force profile strategies can be added to the framework without affecting the existing class structure.

4.7.3. Reconfiguration Commands

The last group of classes in this package are those for the reconfiguration commands which are used for the execution of the reconfiguration procedure. As described in chapter 5, each fixture module generates its own reconfiguration steps and stores them as objects of the subclasses of ReconfigurationCommand. Consequently, when all fixture modules have completed this procedure, a global list can be generated containing all reconfiguration steps necessary to adapt the current fixture setup into the desired configuration in order to accommodate the next workpiece.

The classes for the reconfiguration commands follow the object-oriented "Command" design pattern [104]. According to this, the base class ReconfigurationCommand defines a common interface that consists of the parameter-less method execute(). Based on this, a variable number of child classes can be defined which implement the execute()-method differently. The class ChangeBodyPositionCmd is used to change the body position of the fixture module on the transport component. Essentially, the class publishes the target position according to the communication infrastructure described in chapter six until the movement is complete. The class ChangeTipPositionCmd is used to change the module's tip position by extending or retracting its actuator. Finally, the class ChangeRoleCmd is used to change the role of the fixture module for the next clamping process in terms of the roles clamp, locator or support element.

There are several advantages of this design pattern which are all based on the common interface defined in the abstract base class. Firstly, this approach allows easy enhancement of the system with new reconfiguration tasks. If future enhancements of the reconfiguration procedure require new reconfiguration tasks, the class hierarchy can easily be extended by the further command classes without affecting the rest of the model. Secondly, decomposing the entire fixture reconfiguration task into atomic steps modelled as objects reduces the complexity of the procedure for fixture coordinator. For the coordinator, the execution of the entire reconfiguration sequence consists of simple calls of the execute()-methods of each command which is explained in section 5.3.6. Finally, having the reconfiguration steps modelled as software objects allows to evaluate their effects before they are executed and re-sorting them when collisions between modules are predicted. For this, the base class defines the method causesCollision() which returns true if the execution of the command would result in a collision between fixture modules. The collision avoidance algorithm is explained in section 5.3.5.

4.8.   Chapter Summary

A novel data model has been developed to provide the basis for the conceptualisation of a fixturing system in the framework. The central idea of the model is the representation of a fixture in terms of fixture modules, devices and transport components. The fixture modules are components that interact with the workpiece, while the transport components are elements that allow the repositioning of the modules on the fixturing system. The devices are the subcomponents of the fixture modules which determine their capabilities. In order to ensure a platform-independent definition of the data model, all elements have been defined using Unified Modelling Language.

The developed model addresses the needs of an emerging generation of advanced fixturing systems which integrate a variety of sensor and actuator components. While existing data models have concentrated on the design phase of modular fixtures, the presented approach focuses on the operation of reconfigurable, adaptive fixturing systems. In addition to class inheritance, a set of more advanced object-oriented techniques like design patterns and software delegation have been applied to the fixturing domain in order to achieve a highly

adaptable data model which is able to reflect the changing capabilities of a wide variety of different fixturing systems.

5. Fixture Reconfiguration Methodology

5.1.   Introduction

Two scenarios for fixture reconfiguration have been presented in the use case analysis in chapter 3. The use case "Change Fixture Setup" is concerned with the required steps when fixture modules or devices are added, removed or replaced. Conversely, the use case "Adaptation of Current Setup" refers to the scenario where the fixture automatically adapts the configuration of its existing fixture modules in order to accommodate the requirements of a particular workpiece. This includes adjusting their positions on the transport components and the change of the force profiles. These use cases are addressed by the two parts of the reconfiguration methodology which are illustrated in Figure 5-1. For both parts of the methodology, the diagram shows the required inputs and outputs. The former are shown as parallelograms whereas the latter are depicted as round boxes.
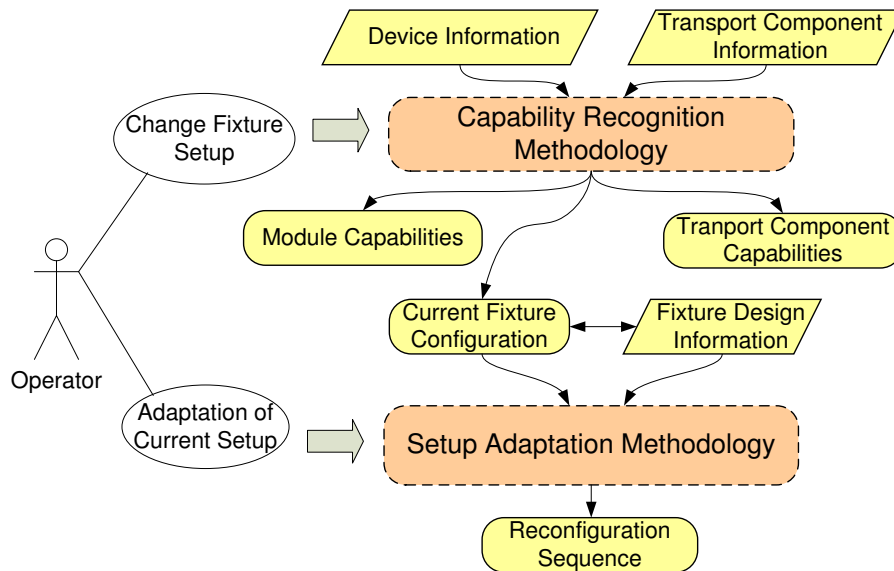
Figure 5-1: Reconfiguration Methodology Overview

The capability recognition methodology determines the capabilities of each fixture module and links them with the transport components in order to generate a global view of the functionalities of the fixturing system. This procedure requires input information from the operator about the devices in each module as well as the transport components according to the structure of the data model described in chapter four. Based on this, each fixture module determines its own capabilities and publishes them according to the communication

concept. As a result, the fixture coordinator discovers the fixture modules and becomes aware of their capabilities. The setup adaptation methodology requires the current fixture configuration and the predefined fixture design parameters as inputs which may come from a data base or provided through XML files. Based on this information, an object-oriented approach is followed to compare the current and the desired fixture configuration as described in section 5.3.2. As a result, the reconfiguration sequence for the adaptation of the fixture is generated and can be executed.

This chapter describes the algorithms for both scenarios and combines them into an integrated methodology for fixture reconfiguration. Similar to the object-oriented data model, the presented algorithms are not tailored to one particular fixture design. Instead, they aim to be applicable to a plethora of adaptive fixturing systems. Section 5.2 provides a detailed description of the decision-making processes for the capability recognition and the generation of the object model. The algorithms for the setup adaptation methodology are subject to section 5.3. Finally, a comprehensive chapter summary is part of section 5.4.

5.2. *Capability Recognition Methodology*

The capability recognition methodology follows a hierarchical approach which is decomposed in two levels. In the first level, each fixture module determines its own capabilities based on its devices, utilising the model elements described in chapter 4. Based on this, they publish their capabilities using the communication infrastructure, described in chapter 6. The second level takes place in the fixture coordinator which receives the capabilities of the fixture modules and the transport components and combines them to generate a complete view of the fixturing system. Below the general assumptions and requirements for both levels are summarised.

5.2.1. Assumptions and Requirements

5.2.1.1. Independent Software for the Fixture Modules, Transport Components and the Fixture Coordinator

The methodology assumes the existence of individual software processes for the fixture modules, the transport components and the fixture coordinator. As can be seen in Figure

5-1, the software processes for the transport component and the fixture modules generate separate object models and use the capability objects to communicate with other systems via the publish/subscribe communication infrastructure.
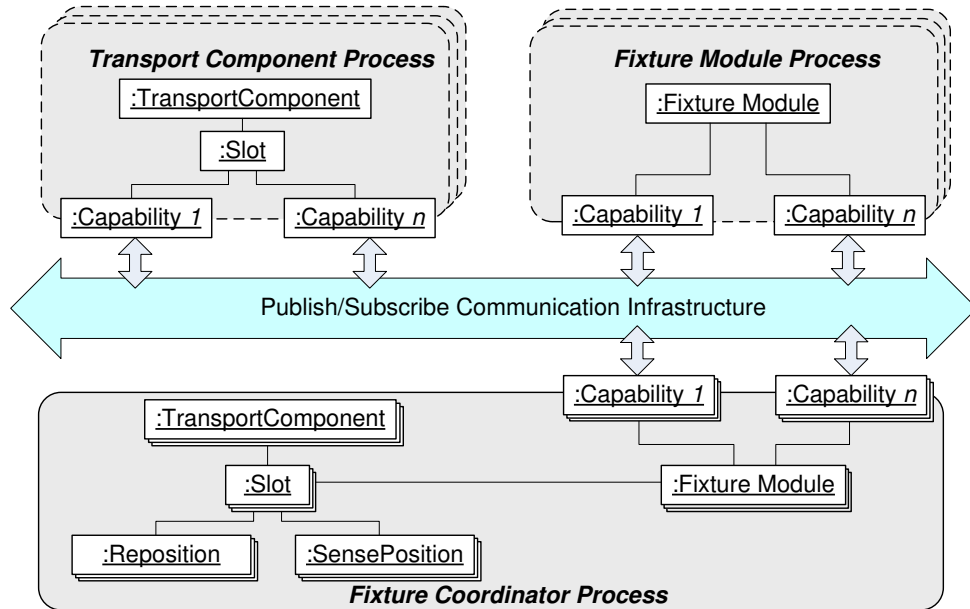
Figure 5-2: Interactions Between the Software Processes for the Fixture Modules, the Transport Components and the Fixture Coordinator

When the fixture coordinator is informed about the modules and the transport components, it generates an own set of objects to represent them, thereby creating a global view of the entire fixturing system. However, in the fixture coordinator only the capability objects for the fixture modules are utilised to exchange information. These objects are linked to the same data topics as the independent software processes for the modules and the transport components. In this way, the fixture coordinator can concentrate exclusively on the fixture modules, which reduces the complexity of the decision-making algorithms performed by it. A detailed description of the realisation of the communication infrastructure is the subject of chapter 6. The methods presented in this chapter focus on the decision-making procedures, taking place in the fixture coordinator and the fixture modules. However, in order to limit the scope of the thesis, details about the internal structure of the software processes for the transport components are omitted.

5.2.1.2. Required Inputs for the Capability Recognition on Module Level

In order to generate the local object model, the fixture module software must be provided with information about the capabilities of its incorporated devices and their logical links between each other. The device description must include the following information:

- A unique numerical identifier for the fixture module
- Technical information for each device according to the data model, in particular
    - The device type
    - A unique numerical identifier for the device
    - The measuring range and resolution for sensing devices
    - The stroke range, swing range and a reference to its connected sensors for clamping devices
    - the path to a software library to access the device
    - Additional device-specific parameters required by the library. Examples for such inputs are the board identifier and the channel number for the data acquisition card, used by a sensor device or the axis number for the motion control card of an actuator device.
    - The spatial description of the position and orientation of the coordinate system of the device, relative to the coordinate system of the fixture module.

The device information can be provided in several ways, including a data base, manual operator input or a configuration file. For this research study, an XML-scheme has been used which is shown in the example listing in Appendix A. The information for each device of the fixture module is provided within individual <device> blocks. This contains general details about each device, such as the identifier, the device type and the description text. Additionally, the details for the capabilities of each device are enclosed in separate sections. Fixture modules can consist of multiple sensor devices connected to either one clamp or one locator or support element. The references to the connected sensors are provided in the configuration file within the <feedbackdevices>-block which lists the identifiers to the sensors. This information is used to build the object hierarchy according to the "Composition" design pattern, described in section 4.4.1. The references to the software libraries, responsible for the hardware access of the devices are provided in the <library>-block. The implementation of these libraries is beyond the scope of the research as this

depends on the vendor-specific hardware. However, the common interface is explained in chapter 6. Furthermore, detailed parameters for the operation of the library can be specified within the <library-parameters>-block. This block is passed to the library during its initialisation which is assumed to be able to parse and interpret the contents.

5.2.1.3. Required Inputs for the Capability recognition on Fixture Level

When the software of the fixture coordinator is initialised, it needs to be provided with details about the transport components according to the specification, described in section 4.6. This information includes:

- The domain type and geometry type of the transport component
- A numerical identifier for the transport component
- The spatial description of the position and orientation of the coordinate system of the transport component, relative to the global coordinate system
- Information about each slot on the transport component, including
    - A numerical identifier
    - The spatial description of the position and orientation of the coordinate system of the slot, relative to the coordinate system of the transport component
    - The workspace of the slot on the transport component, specifying the minimum and maximum coordinates of the slot with regards to the local coordinate system of the transport component
    - Information about the position feedback of slot (see section 4.6.3)

This information can be obtained from a data base, XML-files or manual inputs from the operator. Additionally, each software process for the control of a transport component can publish the details about its capabilities. Based on the provided details, the fixture coordinator instantiates the software objects in order to represent each existent transport component and its slots.

5.2.2. Capability Recognition on Module Level

Figure 5-3 shows a flow chart with the steps performed within the local software routine of each fixture module to generate the local object model for its devices and capabilities. In the first two steps of the procedure, the numerical module identifier and the device descriptions are read. Based on this information, an empty object of the class FixtureModule is created in the third step. However, at this point the object lacks any information about its device configuration, because there are no objects for the devices and their capabilities attached to it. In order to configure it for the existing setup, the objects for the devices and their capabilities are created in the subsequent steps. This results in the generation of the objects representing the capabilities of the fixture module. The following sections describe the steps to gradually produce an object-oriented representation of the fixture module. A summary of the utilised UML notation is provided in the symbology section in the beginning of the thesis.
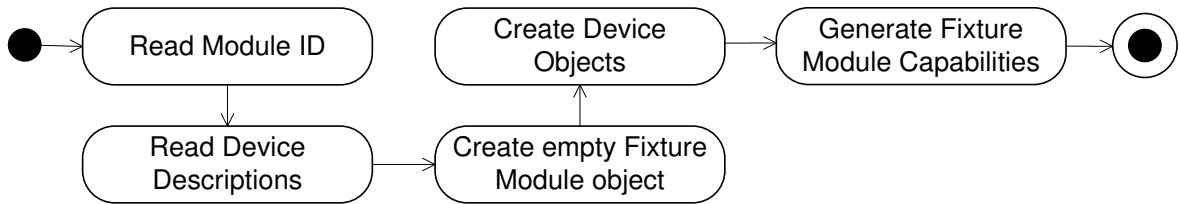
Figure 5-3: Flowchart for the Capability Generation on Module Level

5.2.2.1. Creation of the Device Objects

The fourth step is concerned with the creation of the device objects which have direct access to the hardware. Figure 5-4 shows an UML object diagram for the devices and their capabilities of a fixture module, consisting of a linear actuator equipped with a force sensor. As it can be seen in the diagram, for each device an object of the appropriate class is created and its attributes are configured with the information from the configuration file. These objects contain a reference to a software library which handles the hardware access to the device. Additionally, each device object is attached with adequate capability objects which are generated from the information provided by the device description. They are used to define the functionality of their associated device to higher level objects and to trigger this functionality by calling the installed library.
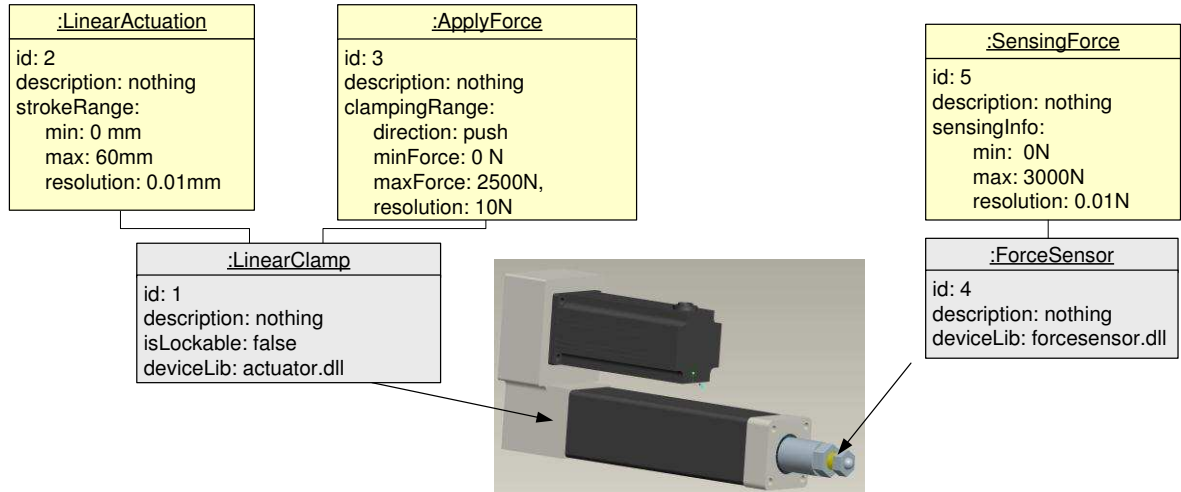
Figure 5-4: Example for the Generation of Leaf Device Objects

The generation of the device capabilities follows a set of rules which are summarised in Table 5-1. For clamping devices multiple capabilities can potentially be generated, if the device description provides sufficient information. In particular, the capability classes Locate and Support can be generated if a clamping device is lockable.

Device type		Allowed Capability classes
Force sensor	→	SenseForce
Displacement sensor	→	SenseDisplacement
Linear clamp	→	ApplyForce, LinearActuation, Locate, Support
Swing clamp	→	ApplyForce, SwingActuation, Locate, Support
Locator element	→	Locate
Support element	→	Support

Table 5-1: Allowed Capability Classes for the Device Types

To express the logical links that exist between the devices a tree structure is generated, based on the "Composite" design pattern, described in section 4.4.1. To connect two devices in the object model, a new object of the class CompositeDevice is created. The latter is attached with the capabilities of the sub devices, thereby generating a combined functional view. Additionally, the resulting capability objects of the composite device are each linked to the particular lower level capability objects they have been created for. This way, requests can be delegated down to the capabilities of the device objects, which access the hardware by calling the library interface. Figure 5-5 shows a UML object diagram to illustrate the concept for the previous example fixture module.
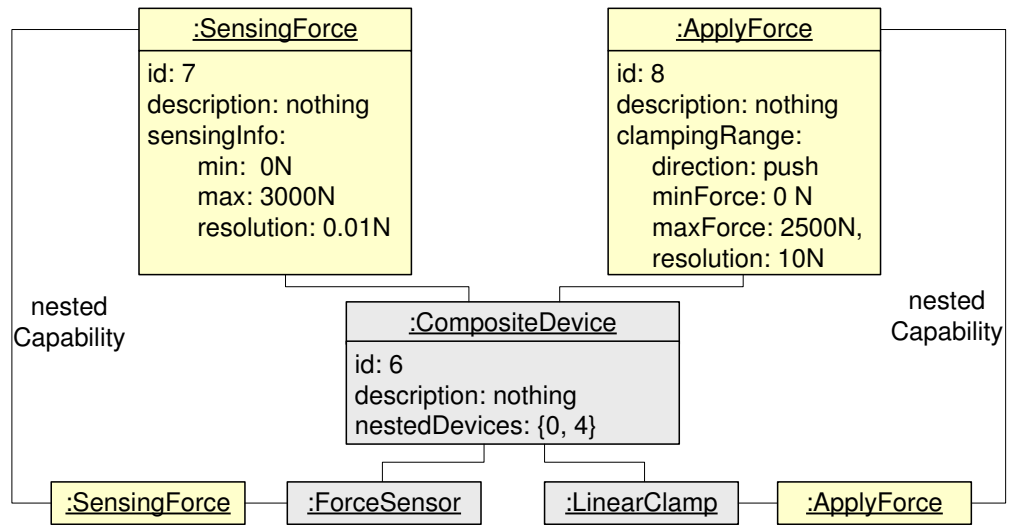
Figure 5-5: Example for the Generation of Composite Device Objects

For the sake of simplicity, the diagram only displays the ApplyForce capability for the clamp and the resulting composite object whilst omitting the capability object for the linear actuation. Further, the class attributes for the bottom objects are omitted since they have already been shown.

5.2.2.2.   Generation of the Fixture Module Capabilities

In the last step, the fixture module object is configured with the generated device structure by attaching it with the device objects of the upmost layer. During this step, the objects representing the fixture module capabilities are created, based on the device capabilities. As described in section 4.5.2, only the module's capability objects are connected with the publish/subscribe communication infrastructure. Consequently, they represent the interface of the module for other subsystems without disclosing details of the internal device structure. Table 5-2 summarises the set of rules for the generation of the fixture module capabilities. Initially, one ProvidesRole-capability is created and connected to the fixture module. By default, its attributes indicate that the module supports none of the defined roles. Subsequently, each device capability is mapped to a newly created object of an adequate class for the fixture module capabilities which were described in section 4.5.2. If the added device has an ApplyForce capability, the fixture module object is attached with an object of the type AdustClampingForce whose class attributes are filled with the information of the device capability. Additionally, the ProvidesRole capability of the module is updated accordingly.

Capabilities of added device object		Generated capability for the fixture module
ApplyForce	→	AdjustClampingForce, ProvidesRole.clampRoleInfo.isSupported := true
LinearActuationCapability or SwingActuation	→	AdjustTipPosition
SenseDisplacementCapability	→	SenseTipPosition
SenseForceCapability		
If device has ApplyForce capability	→	SenseClampingForce
else	→	SenseReactionForce
Locate	→	ProvidesRole.locatorRoleInfo.isSupported := true
Support	→	ProvidesRole.supportRoleInfo.isSupported := true

Table 5-2: Rules for the Generation of the Capabilities for Fixture Modules

A device with the ability to sense force can potentially result in multiple capabilities for the fixture module, depending on whether the force sensor is connected to a clamp or a passive element. In the first case, the device capability of the type ApplyForce is existent, resulting in the generation of the SenseClampingForce capability. Otherwise, the SenseReactionForce capability is created. Moreover, if a force sensor is connected with a lockable clamp, both fixture module capabilities are generated because the module can act as a clamp and a passive element. During the operation, one of them is deactivated, depending on the current role of the module. Additionally, each of the created fixture module capabilities is linked to the device capability it has been generated for. Figure 5-6 shows the final object model for the example module that has been used throughout this section.
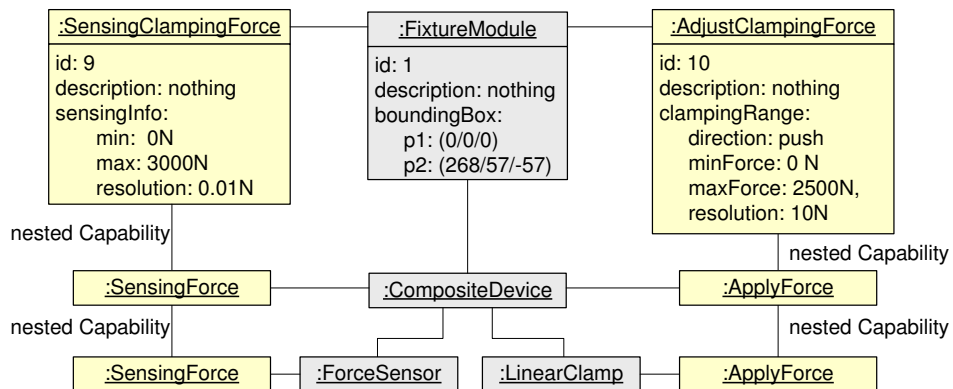
Figure 5-6: Example for the Instantiation of the Fixture Module Capabilities

At the bottom, the device structure and the associated capabilities are displayed in a simplified way, since they were explained in the previous section. The fixture module object is attached with the tree structure, which in this case consists of one composite

device and its two leaf devices. Based on the capability objects of the composite device, the fixture module is attached with objects of the classes AdjustClampingForce, SenseClampingForce and AdjustTipPosition. The latter is not shown in the picture to simplify the diagram. Further, the local object for the fixture module does not contain information about its position and orientation in the global coordinate system. This information is generated by the fixture coordinator in the next step when the modules are linked with the transport components. Ultimately, the module software publishes its capability information according to the communication concept. As a result, other subsystems such as the fixture coordinator discover each module and their capabilities. A detailed description on the publishing of the capabilities can be found in chapter 6.

5.2.3. Capability Recognition on Fixture Level

While the steps described in the previous section are performed for each fixture module, a second data model is instantiated in the fixture coordinator. This includes the objects for the representation of the transport components and the discovered fixture modules. Figure 5-7 illustrates the steps that are performed by the fixture coordinator.
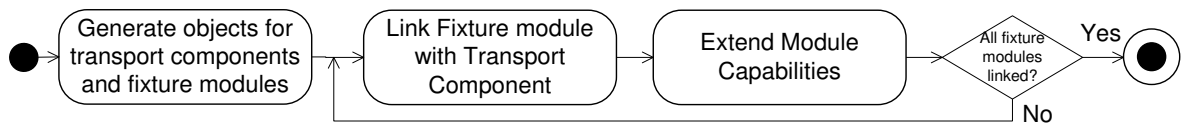
Figure 5-7: Flowchart of the Capability Recognition on Fixture Level

5.2.3.1. Generate Objects for Transport Components and Fixture Modules

In the first step the objects for the transport components are generated, based on the provided configuration details. For each transport component a set of objects is instantiated for its slots and capabilities. As mentioned before, these objects are exclusively used to represent the existing transport component layout in the internal data model of the fixture coordinator. Figure 5-8 illustrates the object generation for two different types of systems. Figure 5-8.a shows a continuous transport component consisting of a rail with one carrier that can be connected with a fixture module. Consequently, one object of the class TransportComponent is generated which is linked to one Slot-object. The workspace for the movement of the slot is captured in the capability class Reposition. This includes, the linear range for the slide-movement along the rail which is indicated by the two points

(x_{min}/y_{min}/z_{min}) and (x_{max}/y_{max}/z_{max}). Additionally, the workspace defines the allowed clocking of the slot around its axis. In the example unterhalb the slot is assumed to be rigidly mounted on the rail, therefore allowing no clocking.
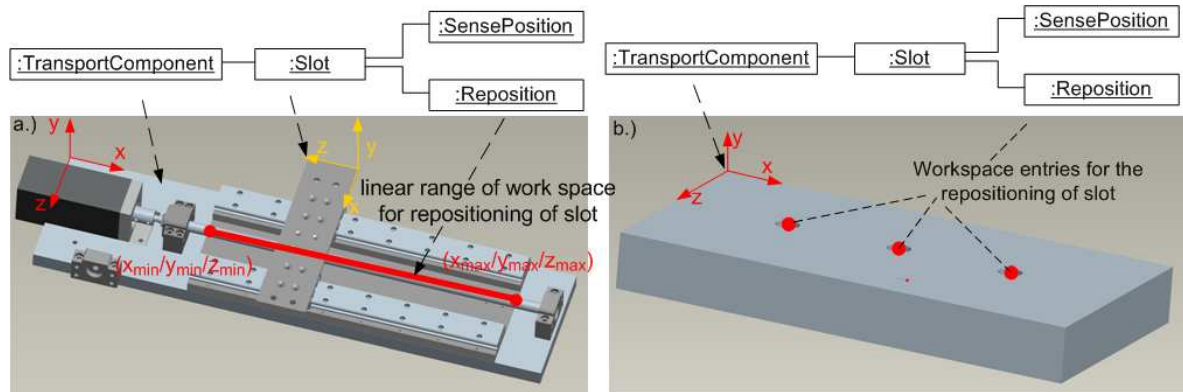
Figure 5-8: Object Generation for a.) Continuous and b.) Discrete Transport Components

For discrete transport components as shown in Figure 5-8.b a different approach is applied. Instead of creating three Slot objects for the three mounting holes, only one object is generated. This Slot object is linked to one Reposition-capability object containing three workspace entries. A fixture module can be connected with the slot in one of the points specified by the workspace entries. Consequently, the actual position of the slot for discrete transport components is unknown until they are linked with a fixture module. Therefore, the class Slot is an abstract concept that does not necessarily reflect a concrete hardware component in the system. Instead, it is a means to attach attributes to the connection between a fixture module and a transport component. When fixture modules are discovered by the system, further Slot objects are generated. The maximum number of slots is limited by the number of workspace entries. This approach is different from continuous transport components where all Slot objects are created immediately, depending on the number of carriers. Essentially, it makes it possible to model discrete transport components with a large number of mounting holes without the generation of too many capability objects which would otherwise overwhelm the publish/subscribe communication infrastructure.

In addition to the instantiation of the objects for the transport components, the fixture coordinator is informed by the communication infrastructure about newly discovered fixture modules which have published their capabilities. For each discovered fixture module, the fixture coordinator instantiates an own set of objects representing the module

and its capabilities. However, even though the fixture modules are physically mounted to the transport components, this link is not yet existent in the object model of the fixture coordinator. The reason for this is that the transport component objects are only aware of their slots, but so far they lack the information whether or not a particular slot is connected with a fixture module. Equally, the positional feedback information of the fixture modules, obtained through the SenseTipPosition capability, is meaningless at this time, since a reference to the global coordinate system is missing.

5.2.3.2. Link Lixture Modules with Transport Components

To overcome the aforementioned problem, the second step is concerned with linking the objects for the fixture modules and the slots. For this, additional operator input is required, specifying which fixture modules and slots are connected. For each link, the operator must provide the following details:

- The spatial description of the module with regards to the slot coordinate system. Based on this, the 4 by 4 matrices for the coordinate transformation between the slot and the module's local coordinate systems and vice versa can be generated.

- The clocking range for the module on the slot. This specifies whether or not the module can be reoriented on the slot during the operation of the fixture.

- For discrete transport components, the operator must additionally select the position of the slot from the workspace entries. The reason for this is that the position of a slot on discrete transport components is unknown until it is linked with a fixture module. Based on the operator input, the coordinate transformation matrices between the slot and the transport component's local coordinate systems and vice versa can be generated.

Based on the provided information, the reference to the slot object is set in the fixture module object and vice versa, thereby establishing the link in the model.

5.2.3.3. Extend Module Capabilities

The link between a slot and a fixture module results in two new capabilities for the fixture module which are generated in the third step. Firstly, based on the SensePosition capability of the transport component, the module becomes aware of its body position and orientation

in the global context. It is therefore attached with the capability class SenseBodyPosition. To obtain the current body position of the module, three coordinate transformations are necessary which are summarised in the equation below.

$$M = T_{tc\_to\_global} \cdot T_{slot\_to\_TC} \cdot T_{module\_to\_slot}$$ **(Equ. 5-1)**

According to the order of matrix multiplications, the local coordinate system of the fixture module is first transformed into the coordinate system of the slot, using the matrix $T_{module\_to\_slot}$. The result is transformed into the coordinate system of the transport component, using $T_{slot\_to\_TC}$. Finally, the matrix $T_{TC\_to\_global}$ transforms the result into the global coordinate system. By multiplying matrix M with the origin of the local coordinate system of the fixture module P(0/0/0/1) in homogenous coordinates, the latter is expressed in global coordinates. The result is stored in the attribute currentBodyPosition of the SenseBodyPosition capability object which has been attached to the fixture module. Figure 5-9 shows the complete object model for a rail with one fixture module and illustrates the coordinate transformations.
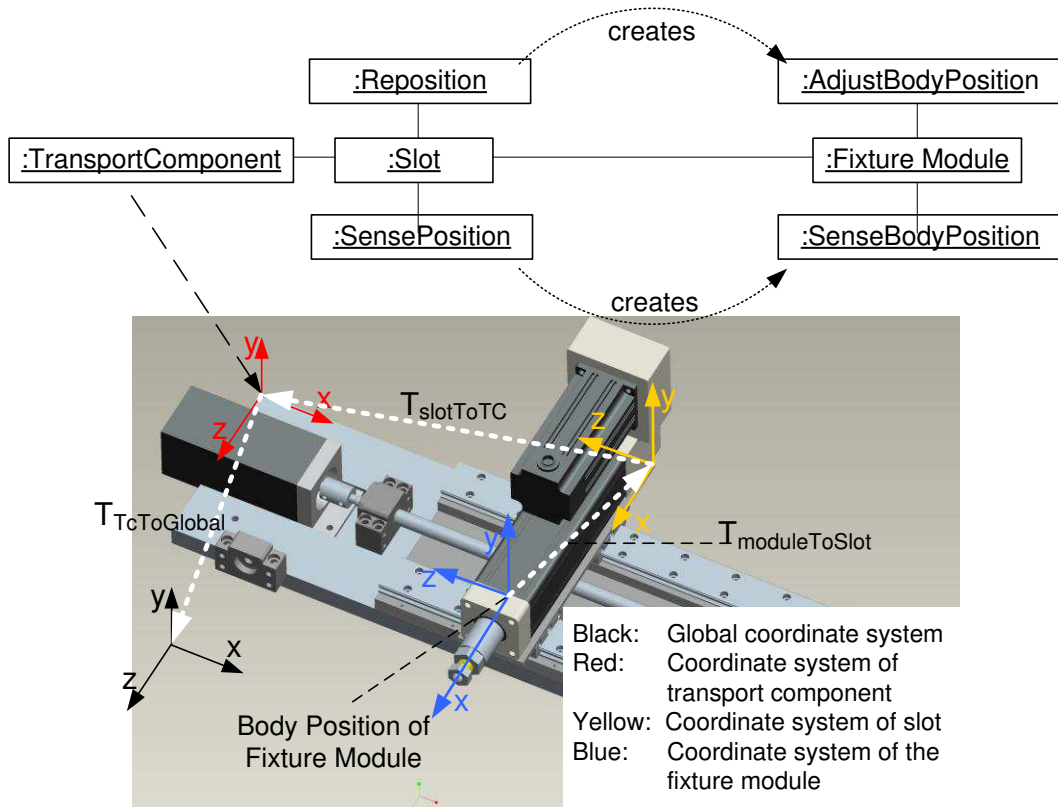
Figure 5-9: Example Instantiation after Linking one Fixture Module with a Slot

For other types of transport components the same principle is applied, leading to a similar object model. For this reason, a second example is not shown here. Secondly, the module gets the ability to change its body position within the limitations imposed by the Reposition-capability of the transport component. Hence, the fixture module object is attached with the capability class AdjustBodyPosition and its workspace is determined based on the Reposition-capability. The Reposition-capability defines the minimum and maximum position for the linear movement of the slot relative to the local coordinate system of the transport component. Using these values, two transformation matrices $MIN_{slot\_to\_TC}$ and $MAX_{slot\_to\_TC}$ are generated according to the principle described in section 4.3.1. The first matrix provides the coordinate transformation between the slot and the transport component when the former is in its minimum possible position. The second matrix provides this transformation when the slot is in its maximum possible position. Consequently, the overall transformation matrices for the minimum and maximum body position of the fixture module are:

$$M_{min} = T_{TC\_to\_global} \cdot MIN_{slot\_to\_TC} \cdot T_{module\_to\_slot} \qquad \textbf{(Equ. 5-2)}$$

$$M_{max} = T_{TC\_to\_global} \cdot MAX_{slot\_to\_TC} \cdot T_{module\_to\_slot} \qquad \textbf{(Equ. 5-3)}$$

The resulting matrices are multiplied with the origin of the local coordinate system of the fixture module P(0/0/0/1) in homogeneous coordinates. After converting the result into Cartesian coordinates, the minimum and maximum body positions of the fixture module in global coordinates are obtained which are stored in the workspace attribute of the AdjustBodyPosition capability. Additionally, this attribute stores the allowed clocking range of the slot and the module. The values for the former can directly be obtained from the Reposition object whereas the values for the latter are retrieved as an operator input when the link is established. For discrete transport components the previously described calculations must be repeated for all workspace entries.

During the operation of the system, the aforementioned capabilities of the fixture module and the transport component are closely connected through the communication infrastructure. These interrelations will be explained in section 6.3.1.

5.3. *Setup Adaptation Methodology*

This part of the reconfiguration methodology aims at the generation of the reconfiguration sequence which adapts an existing fixture layout for different parts of one product family or different manufacturing processes. Essentially, this consists of the repositioning of the fixture modules and the adjustment of behavioural aspects like the clamping force profiles.

5.3.1. Assumptions and Requirements

The following requirements must be fulfilled in order to generate the reconfiguration sequence:

- **Awareness of existing fixture setup**

  The object model of the fixture coordinator must be generated prior to the setup adaptation which contains the current positions and states of all fixture modules and transport components. This is achieved by the method described in the previous section which is carried out whenever the fixture is switched on or a change of the hardware occurs. Furthermore, the position and orientation of the transport components are assumed to be constant during the operation of the fixture. Consequently, the algorithms described in this section concentrate exclusively on the reconfiguration of the fixture modules.

- **Availability of pre-defined fixture design**

  The fixture coordinator must be provided with the pre-defined fixture design parameters for each workpiece. The fixture design information consists of a number of contact points with the workpiece which specify the positions, clamping forces and clamping directions.

- **Availability of information about the workpiece and manufacturing process**

  To retrieve the correct fixture design during the reconfiguration process, the fixture coordinator needs to have information about the workpiece and manufacturing process in question. The research study assumes the availability of this information in whatever form. Hence, the development of workpiece recognition algorithms is not within the scope of this work.

- **The workpiece is correctly positioned in the fixture**

As a result of the decision-making steps of the methodology, the fixture modules are positioned according to the specifications of the contact points. It is beyond the scope of the research to compensate for positional errors during the loading of the workpiece or for geometrical errors of the workpiece itself.

5.3.2. Overview of the Decision-making Process

The setup adaptation method fundamentally relies on matching the contact points from the design with the fixture module objects representing the current configuration of the physical setup. As a result, each module object can individually determine the steps required to transform its current state according to the design specifications. In this way, the generation of the reconfiguration sequence is delegated to the module objects in a decentralised way, thereby making the entire reconfiguration routine independent from the number of modules. The reconfiguration sequence itself is realised with the "Command" design pattern which was explained in section 4.7.3. Figure 5-10 provides an overview of the steps of the entire decision-making process for the fixture adaptation methodology.
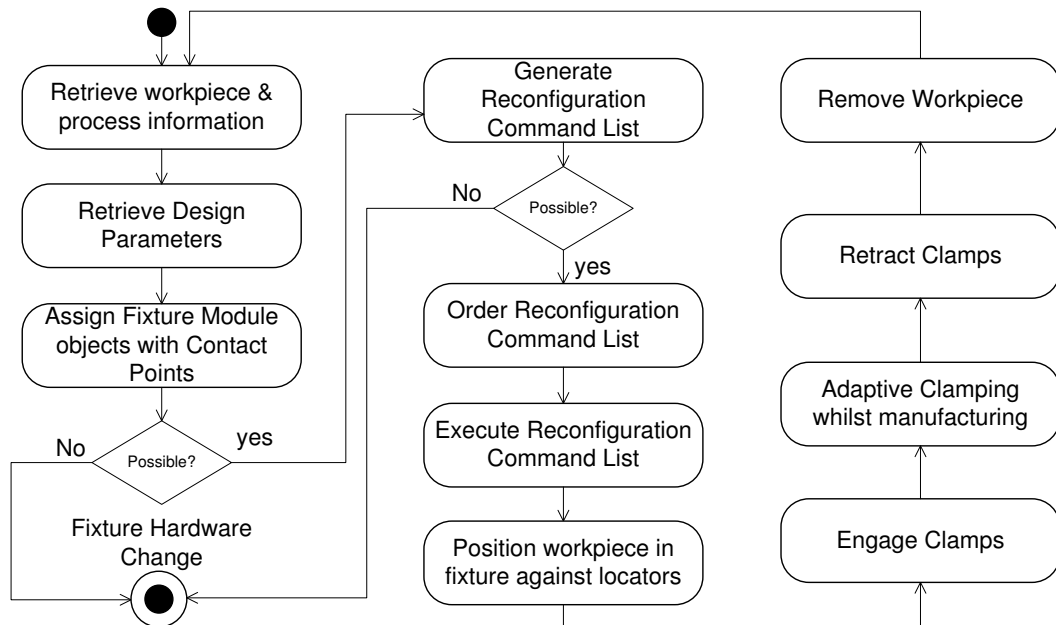
Figure 5-10: Decision-making Process Overview

The fixture adaptation procedure starts with the retrieval of the identifiers for the next workpiece and the manufacturing process. This information can be provided by the operator or through automated feature recognition systems. Based on this information, the corresponding fixture design is retrieved from a data base in the next step. The fixture

design contains all contact points between the fixture modules and the workpiece, each of them specifying a position and further details such as the clamping direction and force profiles, as described in section 4.7.1. The essential step of the methodology is to assign the fixture module objects representing the current configuration with the appropriate contact points. Once this relation is established, the required actions to transfer the current configuration into the target configuration can be derived by each individual module object using the command approach. If the assignment is not possible it can be concluded that the current fixture setup cannot be transferred into the desired status. In this case a manual change of the fixturing hardware is necessary which will ultimately trigger the capability recognition method, described in section 5.2. If the assignment is possible, the reconfiguration commands are generated and stored in a list. In order to avoid collisions between the modules, the command list is then sorted according to a set of rules as described in section 5.3.5. Finally, the fixture coordinator gradually reconfigures the fixture by executing the reconfiguration commands one after another. In particular, this moves the locators to their target positions. The clamping modules are repositioned on the transport components, yet remain retracted. After this, the workpiece is placed in the fixture and positioned against the locators. This can be done manually by the operator or with the use of a robot. Finally, the clamps modules are actuated until they reach the target tip position. This is followed by the adaptation of the clamping force during the manufacturing process as defined by the force profile, accessible from the contact point object. After the completion of the manufacturing process, the clamping modules are retracted, thereby releasing the workpiece which can subsequently be removed from the fixture. A new iteration starts with the retrieval of the information for the next workpiece. The following sections provide a more detailed description of the steps of the procedure.

5.3.3. Assignment of Fixture Modules with Contact Points

This step is essential for the reconfiguration methodology because it enables the fixture modules to become aware of their target position, orientation and force profiles. The module assignment faces the following challenges. Firstly, the contact point specifications are defined independently from the fixturing hardware. Consequently, there is no indication which fixture module can physically reach a particular contact point. Secondly, one contact

point can potentially be assigned with several fixture modules. For this reason, the procedure consists of two parts. In the first part, the possible fixture module candidates for each contact point are found, whereas the second part selects the most appropriate match for each contact point. Figure 5-11 shows the flow chart for the decision-making procedure to find the potential candidates. As can be seen, the algorithm iterates through the list of contact points and uses an additional inner loop to compare them with all fixture modules.
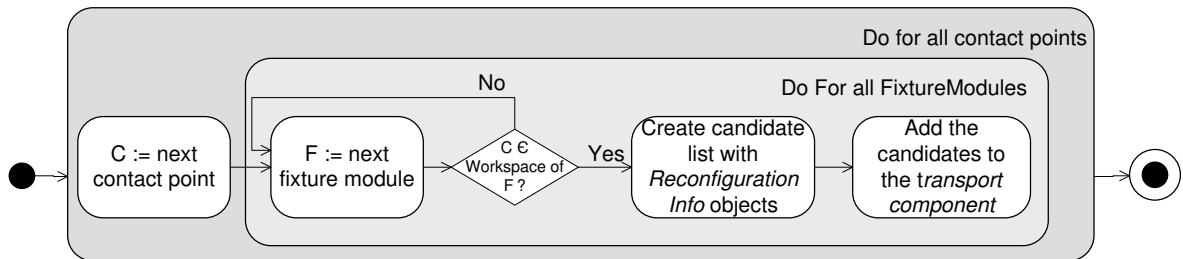
Figure 5-11: Flowchart of the Module Assignment Sequence – Part I: Finding Potential Candidates

For each module, the algorithm iterates through the entries of the workspace list which is provided by its AdjustBodyPosition capability. For each entry it is verified, if the tip of the fixture module can reach the contact point. If this test returns with a positive result, a new candidate is found which is subsequently attached to the transport component, as shown in the flow chart. The test comprises a number of steps which are demonstrated in Figure 5-12 and Figure 5-13. To facilitate the understanding of the principle, the drawings are limited to 2D. However, the described algorithms can be directly applied in 3D space and have been successfully implemented in the experimental test bed, described in chapter 7. Figure 5-12.a illustrates a fixture module in the form of a linear actuator in its current position and orientation.
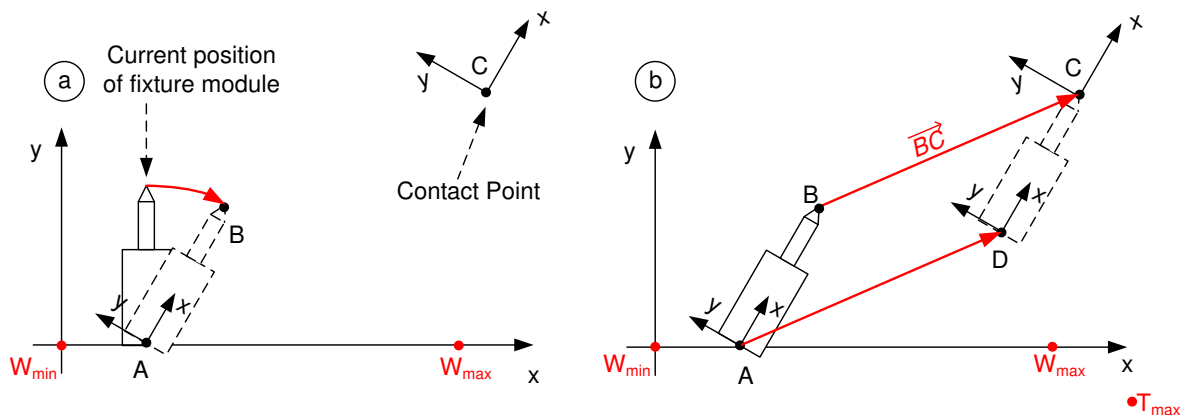
Figure 5-12: Illustrative Example for the Calculation of the Projected Body Position

In this scenario, the module is mounted on a one-dimensional, continuous transport component such as a rail-guide which allows the repositioning of the module along the line between the points W_{min} and W_{max}. The contact point C has an own local coordinate system which is arbitrarily oriented against the global coordinate system. The x-axis indicates the direction in which force shall be exerted. The first step consists of testing, if the module can be brought into the same orientation as the contact point. This renders an inverse kinematics problem, since the reorientation of the fixture module can potentially be achieved by the clocking around its axis and by the clocking of its associated slot. Since the research is not aimed at contributing towards inverse kinematics algorithms, a heuristic approach has been followed. According to this, all permitted slot and module clocking combinations are checked within the limitations, specified by the clocking ranges in the AdjustBodyPosition capability of the fixture module. This approach is feasible for the majority of cases, because due to tight rigidity requirements fixtures typically allow no or limited reorientation of the mounted modules. For each clocking combination, the transformation matrix from the local coordinate system of the fixture module to the global coordinate system (see equation 5-1) is calculated. Based on this, the elements of the rotational part of this matrix are compared with the equivalents in the transformation matrix of the contact point. If all elements have the same values, the module has the same orientation as the contact point. If no combination can be found for any of the workspace entries of the fixture module, the latter cannot be assigned to the contact point and the algorithm proceeds with the next module. If the module can be brought into the same orientation as the contact point, the target body position of the fixture module on the transport component is calculated. For this, the vector $\overrightarrow{BC}$ between the current and the desired tip position is calculated and the module is virtually displaced with this vector, as shown in Figure 5-12.b. As can be seen in the drawing, the resulting point D is not necessarily within the workspace of the fixture module. For this reason, the point D must be translated to point E whose coordinates are within the workspace for the body position, as shown in Figure 5-13.a.
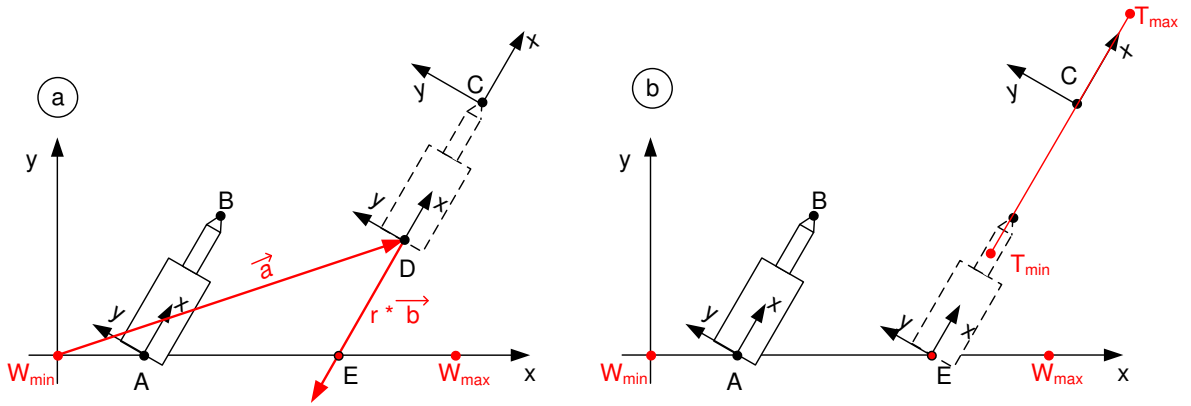
Figure 5-13: Steps to Retrieve the Projected Body Position

This can be done using the vector equation unterhalb for the line which is coincident with the x-axis of the coordinate system of the contact point.

$$\vec{x} = \vec{a} + r\vec{b} \qquad \text{(Equ. 5-4)}$$

The position vector $\vec{a}$ is readily available from the coordinates of point D and the direction vector $\vec{b}$ can be derived from two arbitrary points on the x-axis of the contact point. Consequently, the aim is to determine the coefficient r so that the resulting coordinates for point E lie within the workspace. This can be done by solving the following system of inequations:

$$\vec{w}_{min} \geq \vec{a} + r\vec{b} \qquad \text{(Equ. 5-5)}$$

$$\vec{w}_{max} \leq \vec{a} + r\vec{b} \qquad \text{(Equ. 5-6)}$$

, where $\overrightarrow{w_{min}}$ and $\overrightarrow{w_{max}}$ are the position vectors to the minimum and maximum limits of the workspace entry. If no solution for r can be found, the module cannot be assigned with the contact point. Otherwise, the smallest value from the solution interval is applied in equation 5-4 which results in the target body position E. In the final step, it is verified if the module can still reach the contact point from this position. For this, the contact point coordinates are transformed into the local coordinate system of the fixture module, taking into account its derived target body position. The resulting values for these coordinates can directly be compared with the minimum and maximum limitations for the tip position of the fixture module which are illustrated as T_{min} and T_{max} in Figure 5-13.b. If this test returns with a positive result, the fixture module is regarded as a possible candidate for the contact point. This is expressed with a new object of the class ReconfigurationInfo. This object contains all the necessary information for the repositioning of the fixture

module which were derived during the previous calculations, including the target body position, the target value for the tip position and the desired clocking values for the slot and the module, if applicable. All created ReconfigurationInfo objects are then added to a list attached to the transport component on which the module is mounted. In this way, each transport component collects all possible assignment options for its fixture modules as the algorithm progresses.

Since each transport component can have several modules, there might be more than one candidate per contact point. For this reason, the second part of the module assignment procedure selects the best match from the candidate list. As can be seen in Figure 5-14, this is achieved by iterating through the transport components and reordering their candidate lists in such a way that the most appropriate candidates are sorted in front of less adequate candidates.
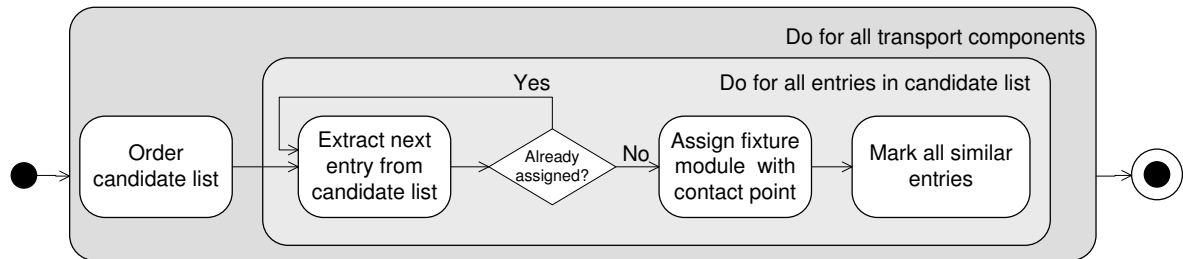
Figure 5-14: Flowchart of the Module Assignment Sequence – Part II: Selection of Candidates

The first ordering criteria is the satisfaction of the functional requirements of the contact points. In order to do this, each requirement of the contact point is compared with the related capability of the fixture module object. Based on this, a fitness value is calculated for each element in the candidate list, using the equation below.

$$g(F,C) = \sum_{i=0}^{n} r_i * 1/n$$

, with n: number of requirements of contact point C   **(Equ. 5-7)**
           i: requirement index
           r: {1 if requirement r_i is fulfilled,
                0 if requirement r_i is not fulfilled}

The factor r_i has either a value of 1 if the ith requirement of the contact point is fulfilled or a value of 0 if this requirement is not fulfilled. The multiplication of r_i with the scale factor $1/n$ ensures that the final result of the equation is always a value between 0 and 1. In this way, the method is independent from the number of requirements imposed by the contact point. Figure 5-15 shows an illustrative example for this calculation. The presented setup

consists of three fixture modules, mounted on a base plate. Module 1 is assumed to consist of a lockable clamp, equipped with sensors for positional and force feedback. The maximum clamping force this module can achieve is 1000N and, if locked, it can act as a locator, withstanding a reaction force of up to 5000N. Module 2 consists of an unlockable clamp that can exert up to 3500N of force. Additionally, the module has sensor devices for the positional feedback of the actuator tip and the clamping force. Finally, module 3 is a locator equipped with a force sensor which can withstand reaction forces of up to 5000 N.
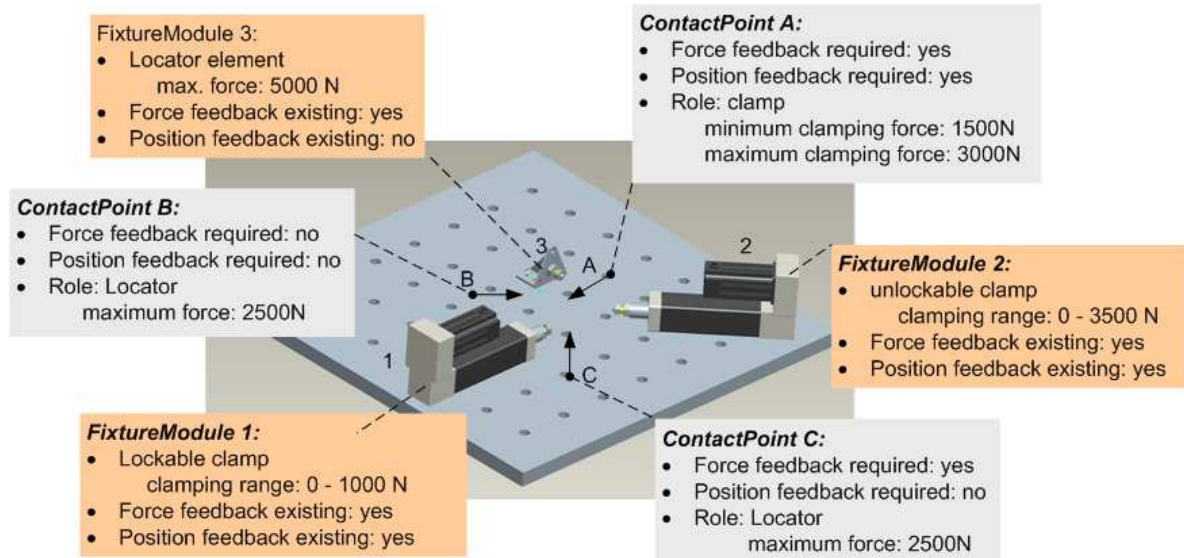
Figure 5-15: Illustrative Example for the Calculation of the Fitness Value

As can be seen, the current setup is confronted with a fixture design consisting of the contact points A, B and C whose requirements are also shown. It is further assumed that all contact points can be reached by all fixture modules which results in a candidate list containing all possible combinations of fixture modules and contact points. In order to calculate the fitness value for the candidate containing fixture module 1 and contact point A (candidate 1-A), all four requirements of the contact point are compared with the attributes of the capability objects, linked with the module. This is shown in Table 5-3.

Requirement	Related Capability	Fulfilled?	Value	Fitness value
Role: Clamp	ProvidesRole	Yes	1 * (1/4) = 0.25	0.25
clamping range: 1500–3000N	AdjustClampingForce	No	0 * (1/4) = 0.0	0.25
Force feedback required	SenseClampingForce	Yes	1 * (1/4) = 0.25	0.5
Position feedback required	SenseTipPosition	Yes	1 * (1/4) = 0.25	*0.75*

Table 5-3: Example Calculation of the Fitness value for Candidate 1A

For the other candidates, the fitness values are calculated in the same way, resulting in the list, shown in the upper row of Table 5-4. The list is then reordered so that the candidates with higher fitness values are sorted before those with lower values.

	1	2	3	4	5	6	7	8	9
Before	1-A (0.75)	1-B (1.0)	1-C (1.0)	2-A (1.0)	2-B (0.5)	2-C (0.5)	3-A (0.25)	3-B (1.0)	3-C (1.0)
After	1-B (1.0)	1-C (1.0)	2-A (1.0)	3-B (1.0)	3-C (1.0)	1-A (0.75)	2-B (0.5)	2-C (0.5)	3-A (0.25)

Table 5-4: Ordering of the Candidate List for the Illustrative Example

A special case exists for one-dimensional, continuous transport components such as rail guides. These types require an assignment method that takes into account the mounting order of the modules because this restricts the allowed repositioning of the modules. As a result, there is the risk that modules are assigned with contact points, they cannot reach because other modules prevent them from being moved to their target positions. This is illustrated in Figure 5-16 which shows a rail with three fixture modules, each of them able to reach the contact points according to their workspace definitions. An incorrect module assignment as shown in Figure 5-16b would obviously lead to an unsolvable situation for the reconfiguration procedure. In order to avoid this, the candidate list is sorted a second time according to the following two criteria.
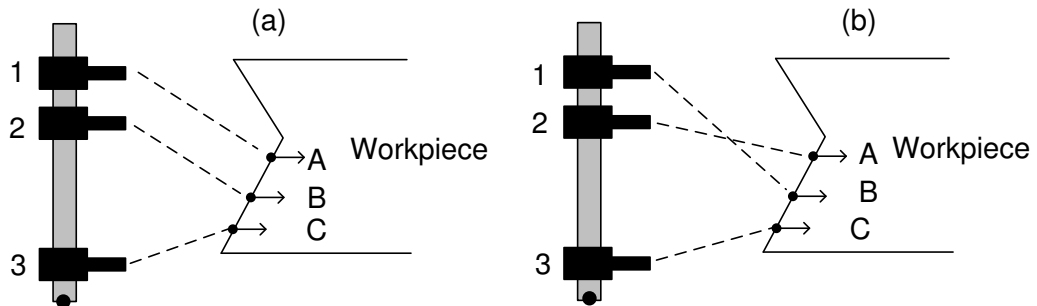
Figure 5-16: Importance of the Mounting Order for One-dimensional Transport Components

First the list is sorted in ascending order according to the distance between the origin of the local coordinate system of the transport component (displayed as a dot in the figure oben) to the current body positions of the fixture modules. This information can be directly retrieved from the fixture module objects. Secondly, the list is sorted in ascending order according to the distance between the origin of the local coordinate system of the transport

component and the target body positions. Consequently, for the example in Figure 5-16 a new sequence of the candidate entries evolves which is shown in Table 5-5.

	1	2	3	4	5	6	7	8	9
Unordered List	1-A	1-B	1-C	2-A	2-B	2-C	3-A	3-B	3-C
First Criteria	3-A	3-B	3-C	2-A	2-B	2-C	1-A	1-B	1-C
Second Criteria	3-C	2-C	1-C	3-B	2-B	1-B	3-A	2-A	1-A

Table 5-5: Illustration of the Ordering of the Candidate List for Rail-based Transport Components

After the list has been ordered, the best matches are selected. For this, the algorithm iterates through the sorted list and connects each unmarked entry with both, the FixtureModule object and the ContactPoint object, thereby establishing a link between the both. At the same time, all other entries of the candidate list which contain the same fixture module or contact point are marked to avoid that either of them are assigned twice. As a result, each module is assigned with exactly one contact point, thereby becoming aware of its desired configuration for the fixturing of the next workpiece. The algorithm is illustrated in Table 5-6 for the previous example.

	1	2	3	4	5	6	7	8	9
1st Iter.	3-C	2-C	1-C	3-B	2-B	1-B	3-A	2-A	1-A
2nd Iter.	3-C	2-C	1-C	3-B	2-B	1-B	3-A	2-A	1-A
3rd Iter.	3-C	2-C	1-C	3-B	2-B	1-B	3-A	2-A	1-A

Table 5-6: Final Assignment of Fixture Modules with Contact Points

As can be seen, the algorithm correctly selects candidates 2-B and 1-A as the best matches. If there is at least one contact point unassigned after the algorithm finishes, the current fixture layout cannot be adapted.

5.3.4. Generation of Reconfiguration Commands

After the completion of the assignment step, each fixture module can independently generate the reconfiguration sequence for the changes required by the desired configuration. For each reconfiguration step, the concerned fixture module creates an individual object of one of the subclasses of ReconfigurationCommand which encapsulates

the required target values. These objects follow the "Command" design pattern which has been described in section 4.7.3 and they are stored in a global list maintained by the fixture coordinator software. The decision-making strategy of the command generation step is illustrated in Figure 5-17 which is carried out for all modules. This leads to a complete list of the required reconfiguration commands in order to adapt the current fixture setup into the desired configuration. It should be noted that the creation of the command objects does not yet trigger the reconfiguration process. The command execution is described in section 5.3.6.
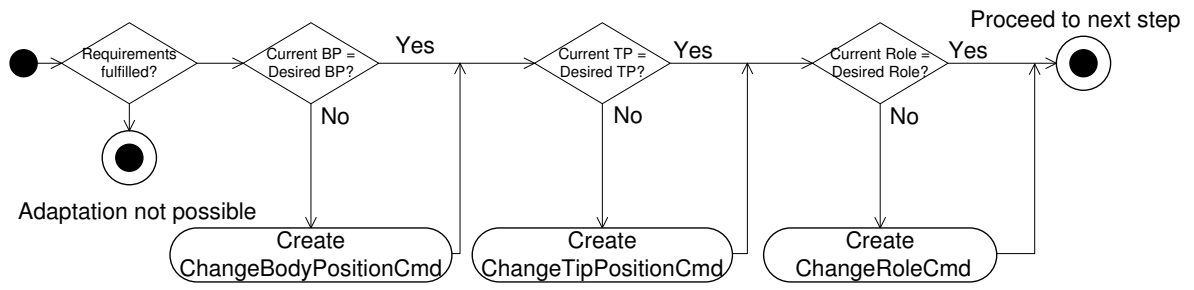
Figure 5-17: Decision-making for the Reconfiguration Command Generation

As can be seen in the flow chart, the first step consists of examining the fitness value of the ReconfigurationInfo object attached to the module, in order to verify if the fixture module meets all requirements of the contact point. This is necessary because the previous module assignment step may have resulted in matches which do not fully meet the functional requirements of the contact point. One reason for this is that the described sort algorithms only ensure that the matches with the highest fitness value are selected. However, it is not guaranteed that selected candidates fully match the requirements. Additionally, in case of one-dimensional transport components, the matches are selected according to the mounting order of the modules, thereby ignoring whether or not they meet the contact point requirements. To verify if all requirements are satisfied, it is checked if the previously calculated fitness value is equal to 1.0. If this is not the case, the functional requirements of the contact point are not satisfied and consequently the setup adaptation process is aborted. Instead, the module must either be exchanged or upgraded which ultimately triggers the capability recognition method, described in section 5.2. If all requirements are met, each module compares its current states with the desired values of the associated contact point. First, the current body position and orientation on the transport component are compared with the specifications, stored in ReconfigurationInfo object. If these are not equal, a new

command of the class ChangeBodyPositionCmd is created and appended to a global list, maintained by the fixture coordinator. Since the command object is configured with the ReconfigurationInfo object, it has access to all target values when it is executed later on. After this, the current tip position is compared with the desired tip position which is stored as an attribute of the ReconfigurationInfo object. If the values differ, a new command object of the class ChangeTipPositionCmd is added to the list. Finally, the current and the requested role of the fixture module are compared, resulting in a new object of the class ChangeRoleCmd, in case a difference is detected. Similar to the previous commands, the object is configured with the reference of the ReconfigurationInfo object.

5.3.5. Collision Avoidance

Before the reconfiguration of the fixture can be executed it is necessary to reorder the command list in order to prevent collisions between fixture modules. The reason for this is that the commands have been created in an arbitrary order, not taking into account any potential collisions between fixture modules. In particular, one-dimensional transport components such as rails need to have a mechanism to predict any collisions during the reconfiguration sequence. For other types of transport components with external mechanisms for the repositioning of the fixture modules, the collision problem is less problematic. For example, in case of a discrete transport component like a base plate with mounting holes, a robotic system can be used to reposition the modules. Typically, these systems have their own path planning and collision avoidance algorithms. For this reason, the algorithm described in this section is focused only on one-dimensional, continuous transport components.

Figure 5-18 illustrates the necessity to reorder the commands for one-dimensional transport components, using the previously described example rail with three fixture modules. On the left side of the drawing, the current module configuration is shown whereas the right side depicts the target configuration. Below the generated reconfiguration commands are listed. From the drawing it is clear, that the execution of this sequence would lead to a collision between the fixture module 1 and 2 when the first command is carried out. Consequently, the list needs to be reordered to make sure that module 2 is moved prior to module 1.

Additionally, it must be assured that the tip position of a module is changed after the module has reached its target body position and orientation on the transport component.

Position in List	Command type	Fixture Module	Description
⚠ 1	ChangeBodyPositonCmd	1	Move module to its target body position
2	ChangeTipPositionCmd	1	Extend actuator to target tip position
3	ChangeBodyPositionCmd	2	Move module to its target body position
4	ChangeTipPositionCmd	2	Extend actuator to target tip position
5	ChangeRoleCmd	2	Change modules role to Locator
6	ChangeBodyPositonCmd	3	Move module to its target body position
7	ChangeTipPositionCmd	3	Extend actuator to target tip position

Figure 5-18: Example for Possible Collision Between Fixture Modules

The algorithm to generate a collision-free reconfiguration sequence takes the unordered command list, called L_{IN}, as an input and creates a new empty list L_{OUT} where the command objects are placed in the right order. It then enters a loop which iterates through all entries of L_{IN}. For each command of L_{IN} it is verified if its execution would lead to a collision. If no collision is predicted, the command object is removed from L_{IN} and added to the output list L_{OUT}. Additionally, its effects for the associated fixture module are internally updated in the data model in order to be able to correctly test the remaining commands in L_{IN}. If, on the other hand, the command would cause a collision, it remains in the unsorted list. After all commands have been tested in the loop, it is verified whether or not the list L_{IN} is empty. If this is the case, the algorithms finishes and the collision-free command sequence can be retrieved from L_{OUT}. On the contrary, if there are any commands left in the list L_{IN}, the algorithm only continuous if at least one element was appended to L_{OUT} during the previously described loop. In this case, another iteration of the described steps is carried out with the remaining elements of L_{IN}. If, however, no elements were appended to the list L_{OUT}, a collision-free sequence cannot be found. The algorithm aborts and the automatic

setup adaptation of the current fixturing system is not possible. The complete algorithm is shown in the flow chart in Figure 5-19.
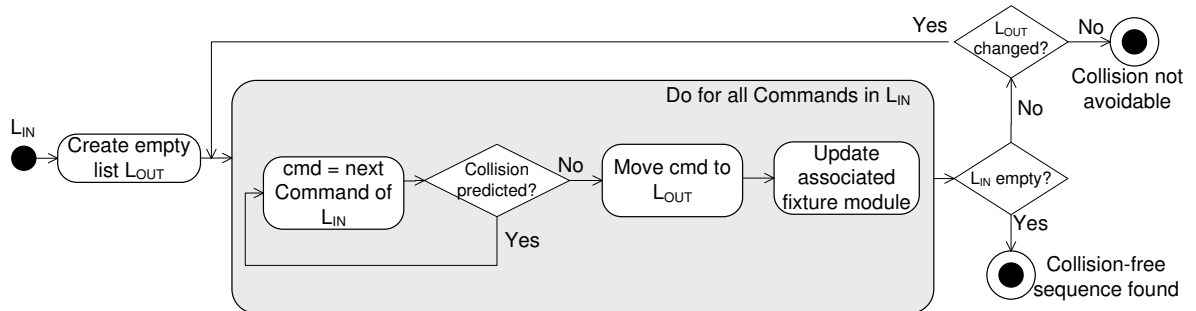
Figure 5-19: Decision-making Sequence for the Reordering of the Reconfiguration Commands

To predict collisions, each command subclass implements the method causesCollision(), described in section 4.7.3. Consequently, the collision verification task is delegated to each command object. The advantage of this object-oriented method delegation approach is that the entire algorithm becomes independent from the number and type of commands. Each command class can implement the collision verification differently without having an effect on the rest of the system. Equally, new command class can be introduced without affecting the overall framework. The subclass ChangeRoleCmd always returns false, since the mere change of the role does not cause any collisions with other modules. The class ChangeTipPositionCmd returns true if the list L_{IN} contains another object of the class ChangeBodyPositionCmd which is linked to the same fixture module. Consequently, in this case the command remains in the list as long as the command to change the body position is not moved to the list L_{OUT}. This strategy ensures that during the execution of the reconfiguration procedure, the modules are first repositioned on the transport component, before their tip position is changed. For the commands to change the body position, it is verified if another module is located between the current body position of the concerned fixture module and its target body position. For this, the direction vector $\vec{d}$ between the current and the target body position is calculated. Subsequently, the module's position is gradually translated along this vector, as shown in Figure 5-20. The drawing shows a simplified view of the bounding box surrounding a fixture module. The module is moved to the target position along the direction vector $\vec{d}$. The intermediate positions during the movement of the module are shown by the dashed boxes. For each intermediate position, including the target position, it is tested if the bounding box of this

module interferes with any of the other modules. If an interference is detected, the concerned command object remains in the list L_{IN}, as shown in the flow chart in Figure 5-19.
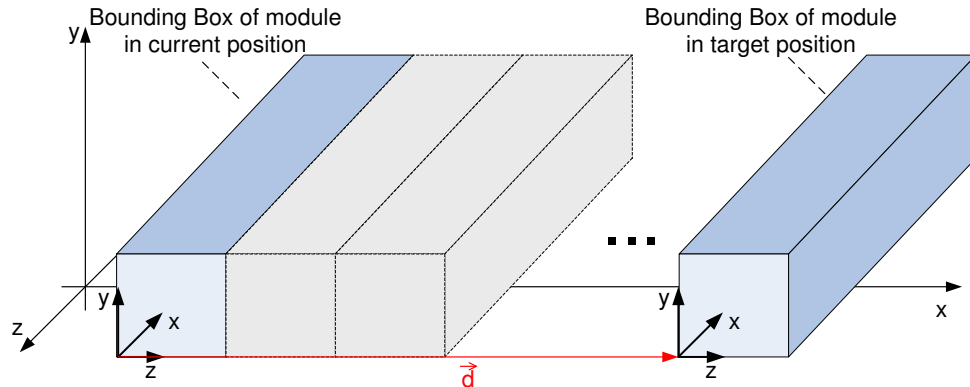
Figure 5-20: Illustration for the Collision Detection

The interference test can be done using any of the myriad of collision detection algorithms available in literature. In the scope of this thesis, the algorithm proposed by Gottschalk et al. [164] is used because it can efficiently detect collisions between two arbitrary oriented objects in 3D space. The algorithm requires the coordinates of the bounding boxes of the fixture modules and the matrices for the coordinate transformations from the local coordinate systems of both modules into the global coordinate system. Based on these inputs, it can be verified if two modules interfere with each other. The internal details of this algorithm are irrelevant for the overall decision-making of the reconfiguration procedure and are therefore omitted. A software library implementing the algorithm is available [165] and has been used for the prototype, described in chapter 7.

To illustrate the complete procedure, the command list for the example introduced in Figure 5-18 shall be ordered according to the algorithm described above. Iterating through the unordered command list, the first entry is the command to move the fixture module 1 to its new body position. As can be seen clearly from the drawing in Figure 5-18, this causes a collision with module 2. Consequently, the command object remains in the list. The second entry is concerned with the change of the tip position of module 1. As described before, this object also remains in L_{IN} because there is still an object of the type ChangeBodyPositionCmd in the list which is related to the same fixture module. The third entry does not cause a collision and is therefore added to the so-far empty list L_{OUT}. In the

same way, the remaining commands in L_{IN} do not cause any collision and are therefore moved to L_{OUT} one-by-one. Hence, after one iteration the contents of the lists L_{IN} and L_{OUT} are as shown below.

Unordered list L_{IN}

Index	Command type	Module
1	ChangeBodyPositonCmd	1
2	ChangeTipPositionCmd	1

Ordered list L_{OUT}

Index	Command type	Module
1	ChangeBodyPositonCmd	2
2	ChangeTipPositionCmd	2
3	ChangeRoleCmd	2
4	ChangeBodyPositonCmd	3
5	ChangeTipPositionCmd	3

Figure 5-21: Example - the Lists L_{IN} and L_{OUT} after the First Iteration

A second iteration through the list L_{IN} is carried out, because L_{IN} is not empty and the list L_{OUT} was changed during the previous loop. Thus, the command for the movement of module 1 is tested again for collisions. This time, however, the internal data model takes into account the target positions of the other modules as an effect of the previously sorted commands. Consequently, no collision is detected this time and the command is added to the end of list L_{OUT}. After the completion of the second iteration, the algorithm concludes, since L_{IN} is now empty. The final collision-free command sequence is shown below.

Unordered list L_{IN}

Index	Command type	Module
	{empty}	

Ordered list L_{OUT}

Index	Command type	Module
1	ChangeBodyPositonCmd	2
2	ChangeTipPositionCmd	2
3	ChangeRoleCmd	2
4	ChangeBodyPositonCmd	3
5	ChangeTipPositionCmd	3
6	ChangeBodyPositonCmd	1
7	ChangeTipPositionCmd	1

Figure 5-22: Example - the Lists L_{IN} and L_{OUT} after the Second Iteration

5.3.6. Command Execution

If all previous steps were successful, the command list can be executed to gradually transform the fixture configuration. This is done in two phases, as shown in Figure 5-23. These phases can be indicated by the fixture coordinator by the setting of state variables which the command objects can access. The first phase is carried out before the workpiece is placed in the fixture. All modules are repositioned on the transport components and the

roles of all modules are changed to their target specification. Additionally, the tip positions of all modules acting as locators are adjusted. However, the commands to adjust the tip position of modules acting as clamps are not executed in this phase. Consequently, these modules remain retracted in this phase. After the workpiece is placed in the fixture, the second phase commences which adjusts the tip positions of the clamping modules.
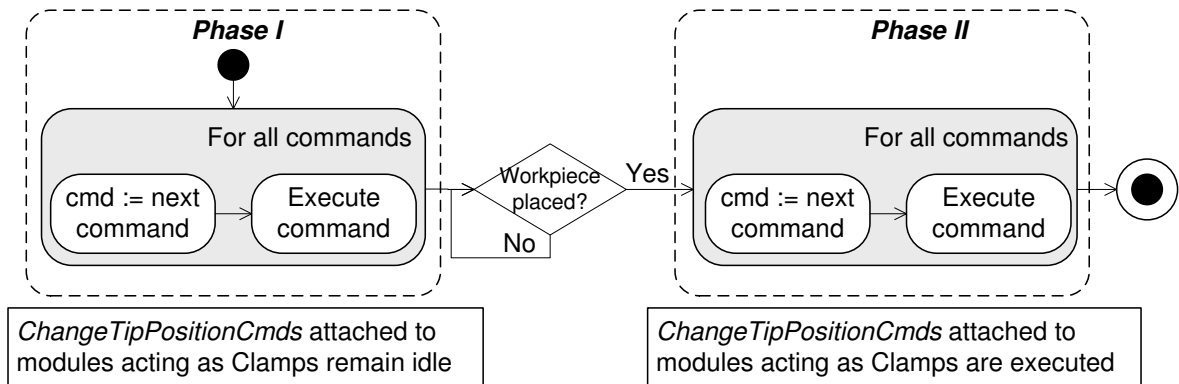
Figure 5-23: The Two Phases of the Command Execution Sequence

As can be seen in the flow chart, the command execution sequences in both phases look similar. This is because each subclass of ReconfigurationCommand can implement the execute()-method differently. The class ChangeRoleCmd updates the required role in the internal data model of the fixture coordinator by setting the attribute currentRole in the class ProvidesRole. In case, the module can act as a clamp or a locator, the sense force capabilities are activated adequately. For example, if the module acts a clamp in the next configuration, the capability class SenseClampingForce is activated and the capability SenseReactionForce is deactivated. This way it is avoided that conflicting force sensor information is received during the clamping procedure. The class ChangeTipPositionCmd implements the execute()-method such, that it returns immediately without doing anything during the first phase of the reconfiguration process. In the second phase however, the command object publishes the module identifier and the desired value for the tip position, using the communication infrastructure. These target values are received by the software of the concerned fixture module which subsequently performs the required movement and publishes the current tip position into a separate data topic. As a consequence the fixture coordinator is informed about the progress of the movement and updates its internal data model accordingly. The command object waits until either the target position has been reached or a deadline has elapsed in order to prevent the command from waiting eternally.

The execute()-method of the ChangeBodyPositionCmd class works according to the same principle. The target values are published and it waits until the desired values have been retrieved by the fixture coordinator and updated in its internal data model. The details for the communication infrastructure and the individual data topics are described in chapter 6.

The main advantage of the followed object-oriented design pattern and the delegation principle is that the software framework becomes independent from a particular fixture setup. This is because the command objects do not contain any implementation code to reconfigure a particular fixture module. Instead, each command publishes the desired values for its module, thereby delegating the responsibility for the execution to the fixture modules or other stakeholders which have been registered as subscribers. As a result, the fixture coordinator is unaware of the components responsible to carry out the actions of the command. For example, when a ChangeBodyPositionCmd command is executed, it triggers its associated fixture module object to publish the desired body position and orientation, thereby delegating the task to the equipment responsible for moving this fixture module. However, for the fixture coordinator it is irrelevant which component has subscribed to this information. In the experimental test bed, presented in chapter 7, the software programs controlling the movement of the rail carriers are the subscribers for this information. In other scenarios, the subscriber might be a robot, picking up each fixture module from its current position and placing it at the target position. Furthermore, due to the common interface of the command objects, the complexity of the entire reconfiguration process is reduced to simple calls of a variable number of execute()-methods. These methods are invoked by the fixture coordinator without knowing any implementation details or even the type of a particular command. As a result, the reconfiguration process becomes independent from these aspects. This allows programmers to introduce new command classes in the future or to change the implementation code of existing commands without affecting the overall logic. Additionally, the same algorithm works for simple and complex reconfiguration tasks in the same way as this is reflected only by the number of command objects in the list.

5.4.   Chapter Summary

A new decision-making methodology for fixture reconfiguration has been described which consists of two parts, namely the capability recognition method and the setup adaptation method. The first part describes how the elements of the object-oriented data model are instantiated by both, the fixture module software and the fixture coordinator, in order to reflect the capabilities of the current fixture setup. As a consequence, the software framework is rendered applicable to a large variety of different fixturing systems. The second part defines the steps to reconfigure an existing fixture layout for the next workpiece. The core idea is based on matching the fixture module objects with the contact point objects from the fixture design. This assignment makes it possible to delegate the generation of the reconfiguration sequence to each individual fixture module.

The methodology is a significant improvement over existing approaches because it addresses the adaptation of the fixturing software during the reconfiguration procedure. Unlike existing concepts which appear to be limited to a specific fixture layout, the presented methodology is applicable for a range of different systems. This is achieved through the dynamic generation of the object model elements in order to reflect the capabilities of a given system. In addition, the concept allows for the combination of capabilities when fixture modules are added and uses software delegation to fulfil requests during the operation of the fixture. Moreover, the methodology has contributed to the field of object-oriented design patterns by applying the Command pattern to a new area, namely the fixture reconfiguration problem.

6. Communication Infrastructure for Adaptive Fixtures

6.1. Introduction

The fixtures addressed by this research, consist of an arbitrary number of modules which can be added, removed or replaced to alter the capabilities. Consequently, these fixtures can be characterised as complex distributed systems with dynamically-changing network topologies. For this reason, the reconfiguration methodology and data model must be integrated with a communication infrastructure that is able to dynamically establish communication channels among the modules, the fixture coordinator and other subsystems that need to interact with the fixture.

Available middleware technologies were assessed against the communication requirements of adaptive fixtures in chapter 3. As a result of this evaluation the Data Distribution Service (DDS) was selected as the foundation for the communication framework. Consequently, the mechanisms provided by DDS must be adapted to the needs of the fixturing domain. In particular, suitable data types and data topics must be defined for the data exchange between the fixture modules and the fixture coordinator. Thus, the infrastructure described in this chapter constitutes the adoption of an emerging middleware standard to a new application domain. Additionally, the method interfaces of the data model objects are described which allow the access of the fixturing hardware.

Section 6.2 describes the class structure defined by DDS to realise the publish/subscribe communication and explains how the Quality-of-Service concept is implemented by the middleware standard. Based on this, the communication infrastructure for the adaptive fixtures is described in section 6.3. This includes, the definition of the data topics, the data types and the Quality-of-Service settings. The extension of the data model elements with publisher/subscriber classes and a method interface is the subject of section 6.4. Finally, section 6.5. illustrates the described concept with an example and outlines the interactions between the fixture coordinator and the modules during the clamping procedure.

6.2. *Publish/Subscribe with the Data Distribution Service*

The fundamental principle of the publish/subscribe concept was explained in detail in the literature review (see section 2.5). As discussed there, the approach is particularly suitable for many-to-many communication between an arbitrary number of participants in a dynamically changing network environment. The Data Distribution Service builds on the described communication principle and provides easy-to-use communication services which allow applications to exchange information in a platform-independent way. The following sections aim to give a more detailed overview on how communication is achieved using DDS, in particular the class model and the Quality-of-Service concept.

6.2.1. The Data Centric Publish/Subscribe Model

Data exchange with DDS is realised according to the Data Centric Publish Subscribe (DCPS) model. This model describes the interfaces and relations of all entities that participate in the communication which is shown in Figure 6-1. Although fundamental knowledge of these classes and their relationships is important in order to understand DDS, they do not have to be programmed manually by the application developer. Instead, any DDS implementation provides automated tools to generate these classes, based on the target platform and the data type definitions of the application. The data model for the fixture modules and the fixture coordinator must be enhanced by these classes and the methods they provide must be used in order to achieve communication.
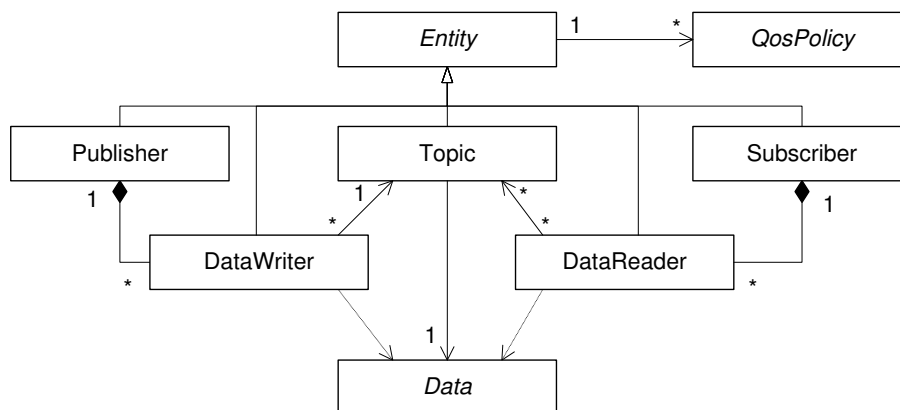
Figure 6-1: Class Diagram of the DCPS model (adopted from [166])

The core of the model is the class Entity. It is configurable with Quality-of-Service policies and can be attached with listener objects to be notified about events. Due to the inheritance

- 133 -

relationship these characteristics are passed on to all other classes of the model, each of them defining a specialised set of QoSPolicy objects to fine-tune the data transfer. The class Topic represents a data flow that is defined by an unique identifier and a data type. More specifically, it connects the publishing and the subscribing ends of the communication. The former consists of the class Publisher that is internally used by the middleware to send out data. It can be associated with multiple objects of the class DataWriter which provides a data type specific access for the application to trigger the publisher. This means, for every data type, a dedicated DataWriter-class is generated which provides the method interface to send samples of this type. Essentially, this consists of the method write() which expects one sample of a given type as a parameter. The subscribing side of the communication is similarly structured. Internally, data is received by objects of the class Subscriber. These can be accessed by the application through data type specific objects of the class DataReader. The latter are automatically generated for each data type and provide the method interface to receive data of a given type. In its most basic form, this consists of the method take() which returns the retrieved samples of a given data type to the application.

6.2.2. The Quality-of-Service Concept

As described in the literature review (see section 2.5), the Quality-of-Service concept is a widely-accepted method to configure the communication behaviour. The QoS model defined by DDS is a rich set of classes which are derived from QosPolicy and therefore can be attached to all objects that are involved in the communication. Each of these policies associates a name with a value and controls a specific aspect of the behaviour of the service. The DDS specification defines separate semantics for the publishing and the subscribing side of each QoS parameter. To ensure correct communication, the QoS policies at the publisher side must be compatible with those at the subscribing end. Figure 6-2 illustrates this for the data exchange between a publisher and a subscriber that are configured with individual sets of Quality-of-Service parameters. The middleware automatically verifies if the QoS settings for corresponding publishers and subscribers match according to the subscriber-requested, publisher-offered pattern. According to this pattern, communication is only established if the offered communication properties of the publisher meet the requested behaviour of the subscriber.
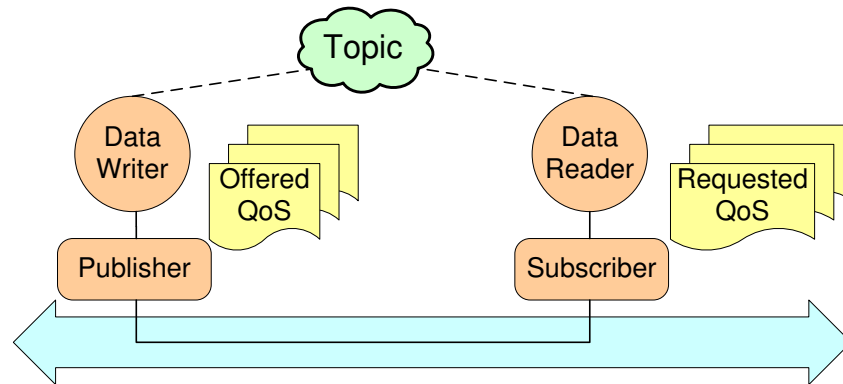
Figure 6-2: DDS Communication Model with Quality-of-Service

Furthermore, the utilisation of QoS settings addresses the needs of real-time applications because it provides precise control over resource usage and the timeliness of the data exchange. At the same time the concept preserves the flexibility inherent to the publish/subscribe model. Additionally, the QoS concept can be used to alleviate the communication challenges resulting from late-joining applications which is explained in detail in section 6.3.3. This aspect is particularly relevant for adaptive fixturing systems, since it provides the means to integrate new fixture modules or other subsystems at any point in time. The complete QoS specification of DDS can be found in [117].

6.3.   Publish/Subscribe Concept for Adaptive Fixturing Systems

6.3.1. Design of the Topic Structure

A number of data topics have been defined which provide the infrastructure for the exchange of information between the various components in the fixturing system. The concept consists of nine topics, each of them associated with one of the data types described in section 6.3.2. The data topic "Module Capability Description" is used by the fixture modules to publish their capability descriptions as one data sample during their initialisation routine. Consequently, the fixture coordinator must subscribe to this topic in order to be informed about the capabilities of the fixture modules. The data topic "Slot Link Info" is used by the fixture coordinator to publish which module has been linked with a particular slot on a transport component. The transport components are subscribers to this topic, thereby becoming aware of the fixture modules they are connected with. The

remaining topics are utilised for the exchange of the current sensor data and desired actuator values. Figure 6-3 illustrates the topic structure for the complete system. In the centre of the picture, the data topics are displayed with their unique identifier. Additionally, the data type that is exchanged through this topic is provided in brackets. An ingoing arrow from an application to a topic indicates that this application is a publisher for this topic, whereas an outgoing arrow classifies the application as a subscriber.
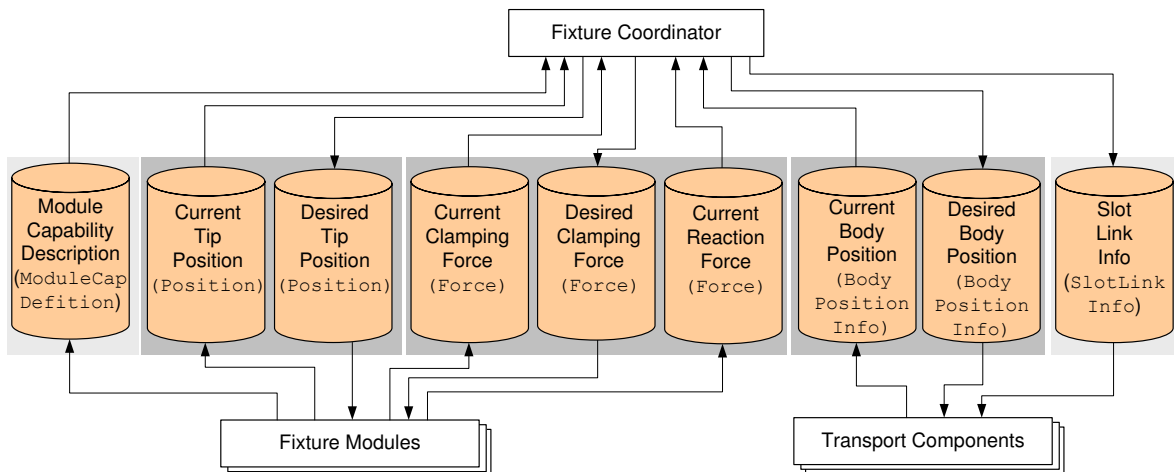
Figure 6-3: Topic Structure of the Publish/Subscribe Communication Architecture

For each module capability two separate data topics are defined. The first topic is used by each fixture module software to populate its current sensor readings while the fixture coordinator is registered as a subscriber. Conversely, the target values for the actuators are published by the fixture coordinator into the second data topic while the fixture modules are subscribers. This way, the fixture coordinator is a publisher for the topics *"Desired Tip Position"*, *"Desired Clamping Force"* and *"Desired Body Position"*. The fixture modules on the other side are publishers for the topics *"Current Tip Position"*, *"Current Clamping Force"* and *"Current Reaction Force"*. As can be seen in the diagram, the fixture modules are not connected with the topics *"Current Body Position"* and *"Desired Body Position"*. This is because the local software of the fixture modules is not aware of their own position and orientation in the global context. Instead, this information is exclusively generated in the fixture coordinator when a module is linked with a slot on a transport component. Only the transport components are able to change the position and orientation of the fixture modules by the repositioning of the associated slots. Consequently, the software of the transport components and not the fixture modules must subscribe to the desired body

position topic. Similarly, the transport components publish information about the current position of their slots, which is the basis to derive the current body position of the associated fixture modules. For the fixture coordinator, these interactions are not visible because in its data model each fixture module object is attached with the capabilities to adjust and feedback its body position. These capability objects are connected to the previously mentioned data topics, thereby establishing the communication with the transport component software. Figure 6-4 presents a detailed view of the described interactions.
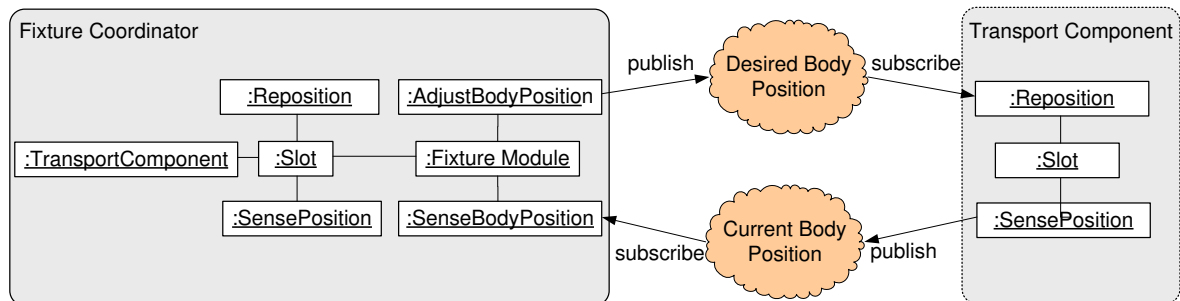
Figure 6-4: Interactions Between Transport Components and Fixture Modules

The right side of the drawing shows the software of the transport component which is responsible for the repositioning of the module. It continuously publishes the slot position and the orientation of the associated fixture module into the topic *"Current Body Position"*. Additionally, it repositions its slot when it receives new target values through the data topic *"Desired Body Position"*. On the left side the data model of the fixture coordinator is shown. As can be seen, it contains the objects for the representation of the transport components which do not participate in the communication procedure. However, the object representing the fixture module possesses two capabilities for the current and target body position which are generated when the module is linked with the transport component as described in section 5.2.3. The capability SenseBodyPosition is continuously updated with the position of the associated slot. Based on this information, it updates the transformation matrix $T_{slot\_to\_TC}$ in the fixture coordinator and calculates the new body position of the fixture module using the equation 5-1, described in section 5.2.3. Similarly, the capability AdjustBodyPosition of the fixture module is used by the fixture coordinator to reposition a module. For this, the target position of the slot, the desired slot clocking and the module clocking are published into the data topic "Desired Body Position", thereby triggering the

associated transport component which is responsible for the correct movement of the slot. The advantage of this approach is that the fixture coordinator can retrieve and change the body position of a module in the same way as any other capability, even though in reality the software of the transport component carries out the task.

The described topic structure can easily be extended by further data topics in the future when more capabilities for fixture modules are defined. For example, new topics can be defined to communicate the current temperature or error states. Similarly, the publish/subscribe paradigm facilitates the integration of the fixture with other subsystems the manufacturing environment. For example, a Human Machine Interface (HMI) can participate in the data exchange by registering publishers or subscribers for the appropriate data topics and receive data without affecting the rest of the system.

6.3.2. Specification of Data Types

The second step for the definition of the communication infrastructure consists of the specification of the data types which are used to transfer information over the topics. Due to the variety of fixture modules with different capabilities and data formats, the concept is challenged by the trade-off between an efficient data transfer and the interpretation of data. On one hand, other systems must be informed about the capabilities of a fixture module, including its limitations and how to interpret the data coming from it. On the other hand, it would not be efficient to publish this meta-information with every data sample during the operation of the fixture. To overcome this problem, the communication infrastructure clearly separates between data types which provide the meta-information needed by other systems to interpret the capabilities of the fixture module and data types for the actual data exchange. This approach allows each module to publish its capability description only once during its initialisation routine. After this, simple data structures can be used for the exchange of information during the operation of the fixture, thereby reducing network load and processing time during the clamping procedure. The following data types have been defined using the platform-independent Interface Definition Language (IDL). Based on these specifications, the source code for the realisation of the publish/subscribe

communication can be generated automatically for numerous programming languages and operating systems.

6.3.2.1. Data Types for the Description of the Fixture Module Capabilities

For the distribution of the module capabilities the structural data type ModuleCapDefinition has been defined in Listing 1. This structure contains the numerical identifier of the fixture module and information about the occupied space of the module. Additionally, further attributes are defined which specify the characteristics and limitations of each capability. However, the attributes do not reveal any information about the fixture module's structure or the capabilities of its subdevices. This information remains encapsulated in the fixture module itself, thereby providing a functional view to the fixture coordinator. In the following listing, the attributes are defined within the brackets. Each attribute is defined by a data type, followed by a name. According to widely accepted conventions, the data types start with capital letters while attribute names begin with small letters.

```
struct ModuleCapDefition{
        long id;
        OccupiedSpace occupiedSpace;
        SenseTipPositionCapability senseTipPositionCapability;
        AdjustTipPositionCapability adjustTipPositionCapability;
        SenseReactionForceCapability senseReactionForceCapability;
        AdjustClampingForceCapability adjustClampingForceCapability;
        SenseClampingForceCapability senseClampingForceCapability;
        ProvidesRoleCapability providesRoleCapability;
};
```

Listing 1: The Data Type ModuleCapDefinition

Each capability attribute is defined as a structural data type containing the relevant properties of a given capability to allow other systems to interpret and use this functionality. The following section provides the details of these properties.

SenseReactionForceCapability and SenseClampingForceCapability

The data type SenseReactionForceCapability is used to communicate the characteristics and limitations of the related capability class SenseReactionForce to other systems. If the fixture module is able feedback a reaction force, the attribute isSupported is set to true and the attribute sensingInfo is filled with the values of the capability class. The data type SensingInfo has been described in section 4.4.3 and defines the value range for the force

feedback, including its resolution and measuring unit. In case, further properties are required, the data type SenseReactionForceCapability can be extended by further attributes. Setting the field isSupported to false, indicates to other systems that the module cannot feedback a reaction force. Consequently, the other attributes are ignored in this case. The IDL definition of this data type are provided by Listing 2.

```
struct SenseReactionForceCapability{
        SensingInfo sensingInfo;
        boolean isSupported;
};

struct SenseClampingForceCapability{
        SensingInfo sensingInfo;
        boolean isSupported;
};
```

Listing 2: Definitions of the Data Types SenseReactionForceCapability and SenseClampingForceCapability

As can be seen in the listing above, the data type describing the capability for the feed back the clamping force has been defined in a similar way which is used to indicate whether or not the fixture module is attached with an object of the class SenseClampingForce.

SenseTipPositionCapability

This data type is used if the module is able to feed back the position of its actuator tip as a result of the capability SenseTipPosition. Since the tip position is published as a point containing the x, y and z values with respect to the local coordinate system of the module, this data type contains three elements specifying the feedback information for the x, y and z components. Listing 3 provides the IDL definition of this data type.

```
struct SenseTipPositionCapability{
        SensingInfo sensingInfo_x;
        SensingInfo sensingInfo_y;
        SensingInfo sensingInfo_z;
        boolean isSupported;
};
```

Listing 3: The Definition of the Data Type SenseTipPositionCapability

As described before, the attribute isSupported indicates whether or not the capability is supported by the fixture module. If this is set to true, the remaining attributes provide more detailed information about the value range, resolution and measuring unit for the x, y and z components of the tip position.

AdjustClampingForceCapability

If the fixture module contains an actuator that can exert a clamping force, the attribute adjustClampingForceCapability is filled with the relevant properties to allow other systems like the fixture coordinator to use this functionality. These values stem from the attributes of the class AdjustClampingForce which has been generated by the fixture module during its initialisation procedure. The definition of the data type is provided in Listing 4.

```
struct AdjustClampingForceCapability{
        ClampingRange clampingRangePush;
        ClampingRange clampingRangePull;
        ClampingDirection clampingDirection;
        boolean isSupported;
};
```

Listing 4: The Definition for the Data Type AdjustClampingForceCapability

The attribute isSupported indicates whether or not the related capability is existent. If set to true, the attribute clampingDirection indicates the supported directions in which the module can exert a clamping force. For this, the data type ClampingDirection is used which has been described in section 4.4.3 The attribute can have the values push, pull, both or unknown. Based on this, the two remaining attributes specify the details for each supported direction, using the data type ClampingRange. As explained in section 4.4.3, this information includes the minimum and maximum amount of force, the measuring unit and the resolution.

AdjustTipPositionCapability

To describe the capability of moving the actuator tip, the attribute adjustTipPositionCapability must be specified by the fixture module. Similar to the previous examples, this attribute is defined as a structured data type containing the relevant properties. This includes an element of the data type ClampWorkSpace whose structure has been defined in section 4.5.2. According to this, the workspace is defined by the stroke range of the actuator in x, y and z direction, relative to the local coordinate system of the fixture module. Additionally, the swing range around one of the coordinate axis can be described, provided that the fixture module consists of a clamp that can perform such a movement. Listing 5 provides the definition for the data type.

```
struct AdjustTipPositionCapability{
        ClampWorkSpace workspace;
        boolean isSupported;
};
```

Listing 5: The Definition of the Data Type AdjustTipPositionCapability

ProvidesRoleCapability

Finally, the attribute of the data type ProvidesRoleCapability is used to describe which functional roles the fixture module supports. Similar to the previous sections, this attribute is filled with the information of the associated capability class. Consequently, the data type consists of three elements to describe whether or not a certain role is supported. For this the already defined data types ClampRoleInfo, LocatorRoleInfo and SupportRoleInfo are used which have been described in section 4.5.2. The IDL definition of the data type is shown in Listing 6.

```
struct ProvidesRoleCapability{
        ClampRoleInfo clampRoleInfo;
        LocatorRoleInfo locatorRoleInfo;
        SupportRoleInfo supportRoleInfo;
};
```

Listing 6: The Definition of the Data Type ProvidesRoleCapability

6.3.2.2.   Data type for the link between fixture modules and slots

When a link is established between the objects of a fixture module and a slot in the fixture coordinator, the software process of the associated transport component needs to be informed. For this, the fixture coordinator publishes one element of the data type SlotLinkInfo into the equally named data topic. As a result, the software processes of the transport components are informed about which fixture modules they are connected with, since they are registered as subscribers for this data topic. The IDL definition of this data type is provided by Listing 7.

```
struct SlotLinkInfo{
        long module_id;
        long tc_id;
        long slot_id;
        boolean isLink;
        SpatialDescription sdModule;
};
```

Listing 7: The Definition of the Data Type SlotLinkInfo

Each sample contains the numerical identifiers of the fixture module, the transport component and the slot. In this way, the subscribers of the transport components can filter out the data samples relevant to them. The Boolean attribute isLink is set to true to indicate that a link between the fixture module and the slot has been established. Conversely, if it is set to false, it signals that the connection between the module and the slot no longer exists. Finally, the spatial description of the fixture module relative to the local coordinate system of the slot is specified by the attribute sdModule. Based on this, the software process of the transport component can generate the matrix for the coordinate transformation between the module's and the slot's local coordinate systems.

6.3.2.3. Data Types for the Exchange of Data during Adaptive Clamping

As a result of the exchange of the module capability description during the initialisation routine, the fixture coordinator knows how to interpret the data coming from a particular fixture module. Additionally, it is aware how a particular module interprets the target values of its actuator. Consequently, the real-time exchange of sensor data and target values during the fixturing procedure can be achieved using simple data structures. Listing 8 shows the definition of the structured data type Force which is used for both, the transmission of the sensor readings from the module to the fixture coordinator and the communication of the target clamping forces. Thus, during the operation of the fixture, the modules continuously publish samples of this data type into the topics "Current Clamping Force" and "Current Reaction Force", depending on the capability objects they have been attached with. To adjust the clamping force, the fixture coordinator publishes elements of this data type into the topic "Desired Clamping Force" which are received by the fixture modules subscribing to this topic.

```
struct Force {
        long module_id;
        ClampingDirection clampingDirection;
        double value;
};
```

Listing 8: The Definition of the Data Type Force

The data type consists of a numeric attribute for the module identifier, the clamping direction and the force value itself. However, no further details like the measuring unit are required, since the meta-information to interpret the force value have been exchanged as

part of the module capability description. The module identifier is required to distinguish between the force samples of the different fixture modules in the system. Similarly, the module identifier must be specified by the fixture coordinator when it publishes the target clamping force for a particular fixture module. Only the module with the matching identifier changes its clamping force by activating its actuator device accordingly. The attribute clampingDirection is used to indicate the current or desired direction in which the force is exerted. The possible values for the attribute can either be "push" or "pull".

In a similar way, the current and desired tip positions can be exchanged using the data type Position whose IDL definition is provided by Listing 9.

```
struct Position{
        long module_id;
        double x;
        double y;
        double z;
};
```

Listing 9: The Definition of the Data Type Position

To feed back the current tip position, a fixture module publishes one sample of this data type into the specified data topic as described in section 6.3.1. Subscribers of this topic can identify the source of this information by examining the attribute module_id and update their internal data model accordingly. The same principle is used by the fixture coordinator to issue the target positions for the fixture modules. It publishes data samples containing the module identifiers and the desired values for the position into the data topic *"Desired T*ip *Position"*. As a result, the fixture modules subscribing to this data topic are informed about the request and reposition their actuator if the module identifier of the received sample matches with their own id.

Finally, the data type BodyPositionInfo is used to exchange the values for the position and orientation of the fixture modules on the transport components. To trigger the repositioning of a module, the fixture coordinator issues one sample of this data type into the topic "Desired Body Position". This contains the numeric identifiers of the module, the transport component and the slot. Additionally, the element position provides the target values for the position of the slot in the local coordinates of the transport component. If required, the

target clocking values for the module and its slot can be defined. As described in section 4.6.2, these values specify the desired rotations around the coordinate axis of the module and the slot, respectively. Negative values indicate a clockwise rotation while positive angles signal a counter-clockwise rotation.

```
struct BodyPositionInfo{
        long module_id;
        long tc_id;
        long slot_id;
        Point position;
        Clocking slotClocking;
        Clocking moduleClocking;
};
```

Listing 10: The Definition of the Data Type BodyPositionInfo

The software processes of the transport components receive the published data samples, since they are registered as subscribers for the mentioned topic. Based on the attribute tc_id, each subscriber can verify if a sample is addressed to it. If this is the case, it triggers the repositioning of the specified slot according to the received target values. The feed back of the current position and orientation of the fixture modules is carried out reversely. For each fixture module that is connected with a slot, the transport components publish a data sample into the topic "Current Body Position". This time, the attributes are filled with the current position of the slot and the clocking values. When the fixture coordinator receives a sample from the data topic, it verifies the source of the information, based on the identifiers and updates the corresponding objects in its data model accordingly.

6.3.3. Quality-of-Service Parameter Specification

The third step for the definition of the DDS-based communication infrastructure consists of the specification of the Quality-of-Service settings for the various topics. In this context, different communication requirements exist which are explained in the following sections.

6.3.3.1. Quality-of-Service Settings for the Dissemination of Module Capability Descriptions

In order to be discovered by other systems, each module publishes its capability description as one data sample during its initialisation routine. However, the fact that this information is published only once, raises the challenge of the so-called "late-joining applications".

Essentially, if the fixture coordinator is launched later than the fixture modules, it does not receive the module capability descriptions, issued before its arrival. Consequently, it cannot interpret the values coming from the modules.

To alleviate this challenge there are two possibilities. The first solution would be to impose a strict start sequence which regulates when the various components of the system have to be launched. However, this solution would jeopardise the aim of creating a loosely coupled communication infrastructure where modules can be added and removed at any time. Therefore, as a second solution a mechanism is preferred that automatically re-distributes the module capability information to late-joining applications. In traditional, particularly client/server-based systems, the problem of redistributing historical data is often solved by periodical broadcasts or by explicitly requesting the required information in a synchronous message sequence. Both approaches would cause significant communication overhead and add complexity to the application logic of the modules. For this reason, the proposed solution is based on the idea that each publisher of the module capability description stores its last-written data sample locally. As a result, it can automatically re-distribute this data whenever a new subscriber for the associated data topic is detected. DDS provides an effective way to establish this method with the QoS concept. In this way, the responsibility for the discovery of new modules and the redistribution of their capability descriptions can be delegated to the middleware and the data is only exchanged when it is really necessary. For the realisation of this strategy, the data writers and data readers for the module capability descriptions need to be attached with the QoS settings as shown in Figure 6-5. The picture also shows how other systems such as Human Machine Interfaces (HMI) can be integrated with the communication infrastructure.


```
HISTORY.depth = 1
RELIABILITY.kind = Reliable
DURABILITY.kind = TRANSIENT_LOCAL
```
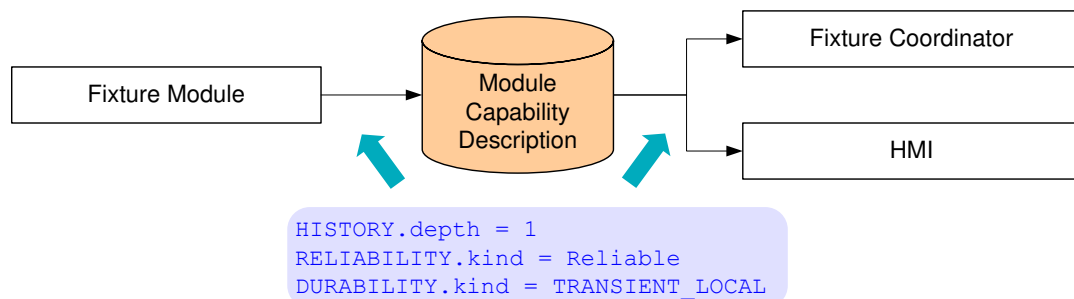
Figure 6-5: QoS Settings for the Distribution of the Module Capability Descriptions

For the publishing side, the QoS parameter HISTORY specifies if and how many published data samples are stored for late-joining subscribers. With its attribute depth set to 1 and the DURABILITY.kind parameter defined as TRANSIENT_LOCAL, it is assured that the last published sample is stored locally in the publisher. Finally, this strategy is only applicable for reliable data transfer which is specified by the value of the RELIABILITY parameter. This way, DDS automatically redistributes the capability information whenever a new subscriber for the data topic "Module Capability Description" is discovered.

A similar approach can be applied to notify the fixture coordinator when the connection to certain fixture modules gets lost. For this, the publishers and the subscriber for the module description need to be configured with the QoS parameter LIVELINESS which determines if and how the middleware detects communication status changes of entities in the network. In more the detail, the attribute LIVELINESS.kind must be set to AUTOMATIC which ensures that the middleware informs the fixture coordinator automatically when "lost" modules are detected. Additionally, the attribute LIVELINESS.lease_duration must be configured with a time span which specifies how often the status is checked.

6.3.3.2. Quality-of-Service Settings for the Exchange for Clamping Data

The exchange of sensor data and target values for the actuators during the fixture operation is subject to real-time requirements. Thus, the communication infrastructure must provide a mechanism to control the timeliness of the data transfer, as well as the resource usage and memory consumption. This can be achieved by adjusting the QoS parameter sets for the publisher and subscriber objects.

In this context, there is a trade-off between the reliability of the data transfer and its timeliness. In order to guarantee a reliable data transfer, any middleware needs to check if data packets are transmitted correctly and resend lost samples if necessary. However, the redelivery of packets takes time and hence destroys the timing determinism of the data transfer [167]. This behaviour would not be acceptable for the exchange of sensor data during the operation of the fixture. Instead, in this scenario it is more important to retrieve the most recent sensor updates, rather than trying to redeliver old samples that have been

lost. This can be achieved by setting the QoS parameter RELIABILITY to the value of "BEST_EFFORT". Further, the publishers and subscribers can be configured with the QoS parameter DEADLINE in order to specify the allowed time frames for the data transfer. In detail, this parameter defines the time period within at least one data sample must be exchanged. If there is no data update during the specified time, the middleware informs the application about the violated timing constraint. An exemplary QoS configuration is illustrated in Figure 6-6 for the exchange of the clamping force. Similar settings are required for all other topics, based on the requirements of a particular application.
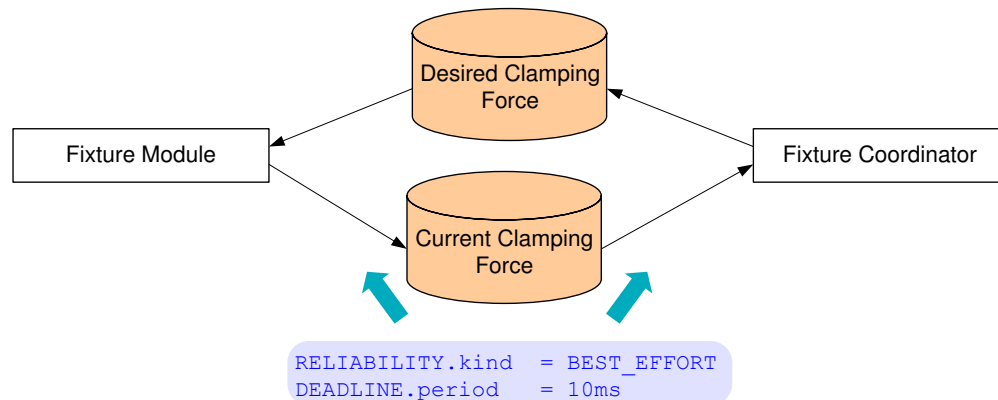
Figure 6-6: Example for the QoS Settings During the Clamping Sequence

Other QoS policies that influence the real-time behaviour are LATENCY_BUDGET and TIME_BASED_FILTER. The first QoS parameter specifies the maximum allowed time span between the publication and subscription of a data sample. Consequently, this policy allows to define priorities for the data transfer. For example, the concept can be used to specify that the communication of the current reaction force values is more urgent than the dissemination of displacement sensor readings. Secondly, the QoS policy TIME_BASED_FILTER can be used to limit the number of data samples a subscribing application receives, thereby controlling both network bandwidth, as well as the memory and processing resources for this application. This can be used to overcome the impedance mismatch, described in section 3.4.1, which affects subscribing applications that cannot process data at the same rates as it is generated by the publishers. For example, if the current clamping force values shall be displayed by a HMI application with a graphical user interface, it is critical to ensure that the HMI is not flooded with too much data. To prevent this, the subscriber can be configured with the TIME_BASED_FILTER parameter to limit

the number of samples it receives, regardless of how fast force sensor values are issued by the modules. This is illustrated in the drawing unterhalb.
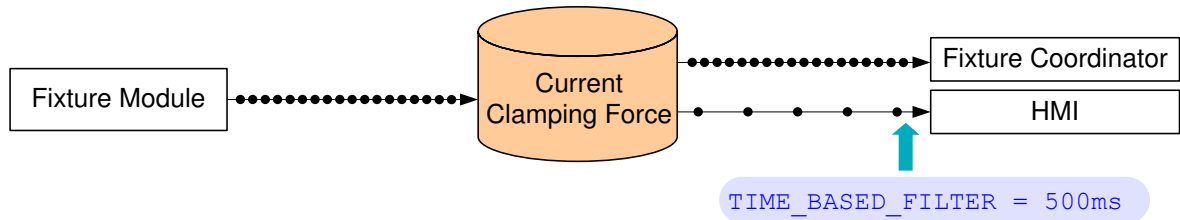
Figure 6-7: QoS Settings for the Limitation of Received Data Samples.

6.4.   *Extension of the Data Model*

6.4.1. Publisher and Subscriber Objects

To accomplish the communication through the data topics, the model elements instantiated by the fixture coordinator and the software processes for the fixture modules and transport components, need to be extended with adequate publisher/subscriber objects. The most appropriate location for these model extensions are the capability objects of the fixture modules. In this way, only those publishers and subscribers are generated which are really required for the information exchange, based on the capabilities the fixture modules offer.

As can be seen in the class diagram in Figure 6-8, three publisher and three subscriber classes have been defined to send and receive force and positional information. The former three classes inherit from the class IPublisher which encapsulates the DDS-internal objects to realise the publishing of data. This includes the objects for the data topic and the DDS-internal publisher. Additionally, this class provides a common interface for its child classes which consists of the method initialise(). This method must be called in order to create and register the publisher/subscriber objects with the Data Distribution Service. Each child class provides a method for the publishing of a specific data type. Internally, the publication is realised with a data writer object that is generated, based on the data type definitions. Thus, the class ForcePublisher contains an object of the class ForceDataWriter. To issue a force value, the method publish() must be invoked which expects the value to be published as an argument. The subscriber side is similarly structured. The parent class ISubscriber provides an interface common to all of its child classes and defines the DDS internal objects for the

topic and the subscriber. Each child class contains a customised data reader and the method subscribe() to receive data and make it available to the application.
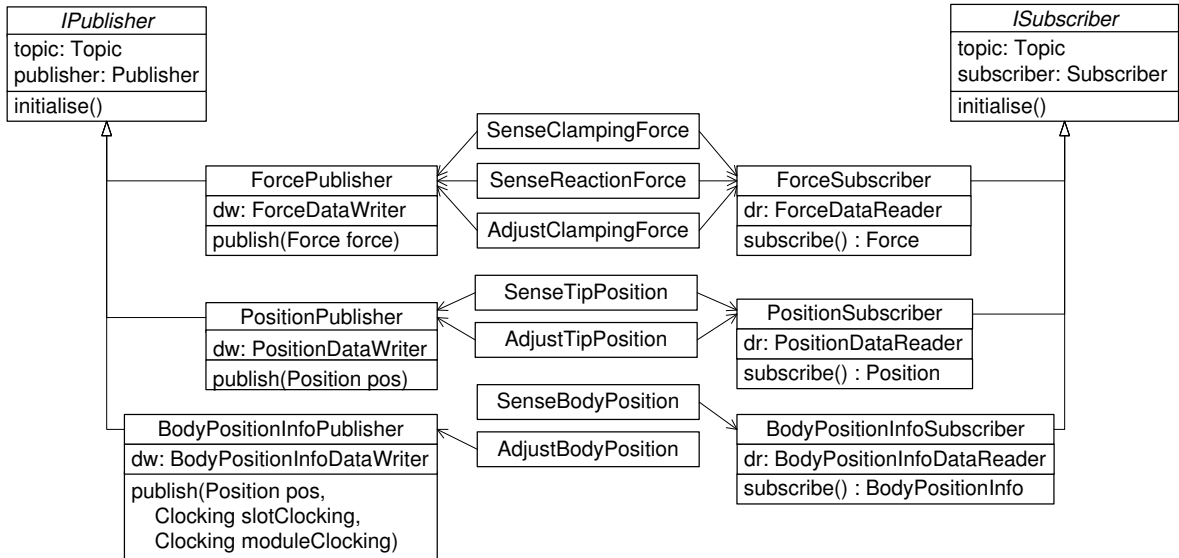
Figure 6-8: Model Extension of the Capabilities with Publisher and Subscriber Objects

The essential step during the initialisation of these objects is to register them with the correct data topic. This is governed by the capability-class they are associated with. For example, a ForceSubscriber or ForcePublisher which is created by the capability SenseClampingForce must be registered with the data topic *"Current Clamping Force"*, while the publisher/subscriber objects created by the capability AdjustClampingForce are linked to the topic *"Desired Clamping Force"*. However, it is important to remember that the model elements for the fixture modules and their associated capabilities are instantiated not only in the fixture coordinator software but also in the software processes of the modules. This means, the publisher/subscriber objects must be registered with different data topics depending on whether they are instantiated in the fixture coordinator or the local fixture module software. Figure 6-9 illustrates this with an example of a fixture module that has the capability to adjust and feedback its clamping force. In the fixture coordinator two capability objects are instantiated which are shown in the upper part of the picture. The capability SenseClampingForce registers an object of the class ForceSusbcriber with the data topic *"Current Clamping Force"* in order to receive sensor updates from the fixture module. To send the target clamping force values to the module, the capability AdjustClampingForce registers a ForcePublisher object with the topic *"Desired Clamping Force"*. In the local software routine of the fixture module, however, the relations between

the data topics and the publish/subscriber objects are reverted which can be seen in the lower part of the picture. The capability SenseClampingForce registers a ForcePublisher object with the topic *"Current Clamping Force"* to issue the current sensor data to remote systems. Finally, the capability AdjustClampingForce registers an object of the class ForceSubscriber to receive the desired clamping forces from the fixture coordinator.
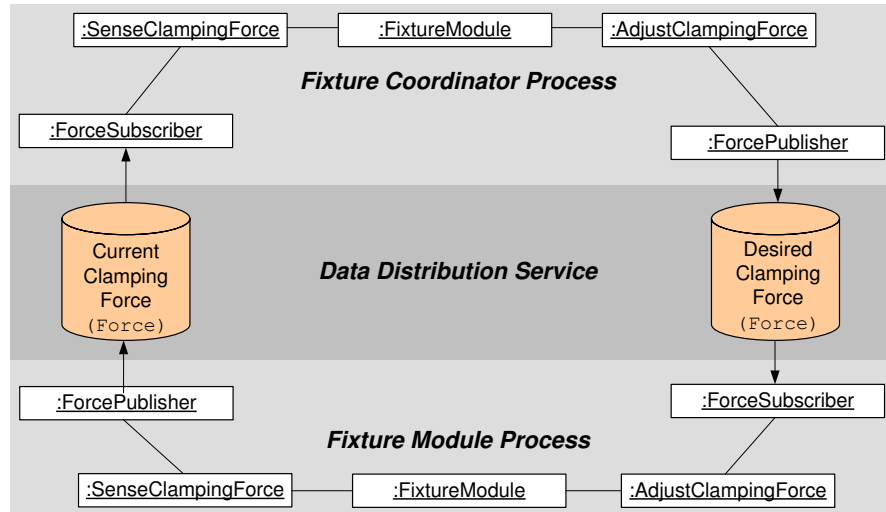
Figure 6-9: Example for the Instantiation of the Publisher/Subscriber Objects

Similar relations exist for all other publisher/subscriber objects which are summarised in the following table. It shows which publisher/subscriber objects are created by a particular capability in the fixture coordinator and the fixture modules. As described before, the capabilities SenseBodyPosition and AdjustBodyPosition are not instantiated in the fixture module program. Instead, the transport components are responsible for the communication of the related information via the associated data topics.

Topic	Capability	Fixture Coordinator	Fixture Module
Current Clamping Force	SenseClampingForce	ForceSubscriber	ForcePublisher
Desired Clamping Force	AdjustClampingForce	ForcePublisher	ForceSubscriber
Current Reaction Force	SenseReactionForce	ForceSubscriber	ForcePublisher
Current Tip Position	SenseTipPosition	PositionSubscriber	PositionPublisher
Desired Tip Position	AdjustTipPosition	PositionPublisher	PositionSubscriber
Current Body Position	SenseBodyPosition	BodyPositionInfoSubscriber	-
Desired Body Position	AdjustBodyPosition	BodyPositionInfoPublisher	-

Table 6-1: Relations Between Topics, Capabilities and Publisher/Subscribers in the Fixture Modules and the Fixture Coordinator

In addition to these objects, the software processes need to instantiate the publisher/subscriber objects for the information exchange through the data topics *"Module Capability Description"* and *"SlotLinkInfo"*. These objects are not linked to a particular capability class, because they are used to transfer the configuration details for the generation of the data model. As can be seen in Figure 6-10, each software process of a fixture module creates one object of the class ModuleCapabilityPublisher. This object is used to publish one sample of the data type ModuleCapDefinition into the specified topic. The fixture coordinator software and other subsystems that need to discover the fixture modules, create one object of the class ModuleCapabilitySubscriber.
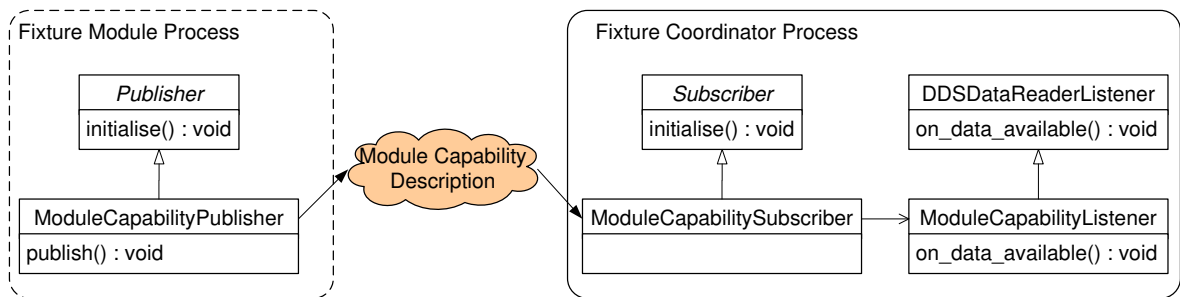
Figure 6-10: Publisher/Subscriber Classes for the Communication of the Module Capability Descriptions

As can be seen in the diagram, this class does not have a method to retrieve data from the associated topic. Instead, a so-called Listener-object is registered with it, that inherits from the DDS-provided class DDSDataReaderListener. The latter defines the method on_data_available() which is automatically called by the middleware whenever a new data sample is available in the data topic. This way, the fixture coordinator is asynchronously informed about the discovery of the fixture modules whenever they publish their capability description. The described approach follows the object-oriented "Observer" design pattern which has been described by Gamma et al. [104]. In a similar way, the information about the connection between the fixture modules and the slots is communicated. The fixture coordinator creates one object of the class SlotLinkInfoPublisher which is connected with the specified topic. Whenever the operator connects a slot with a fixture module, one sample of the data type SlotLinkInfo is published. This information can be retrieved by the software processes of the transport components by instantiating an object of the class SlotLinkInfoSubscriber which is associated with a listener, as can be seen in Figure 6-11.
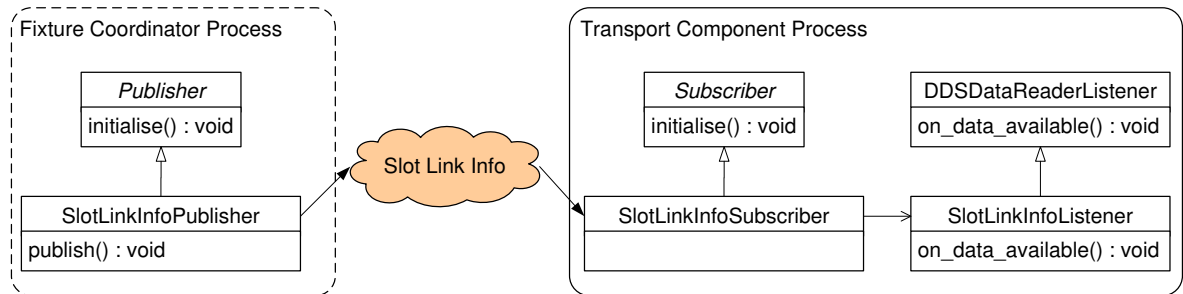
Figure 6-11: Publisher/Subscriber Classes for the Communication of the Slot Link Information

The described class structure of both previous examples raises the legitimate question why the classes ModuleCapabilitySubscriber and SlotLinkInfoSubscriber have been defined, since obviously data is received by the listener classes. The reason is that the listeners cannot exist on their own. Instead, they must be associated to the DataReader-objects which are contained in the subscriber classes.

6.4.2. Method interface of the Capability and Device Classes

To trigger the previously described publisher and subscriber objects it is necessary to extend the capability and device classes of the data model with a method interface. The interface of the class FixtureModuleCap consists of the method perform() which must be called in order to carry out a capability. As can be seen in the class diagram below, the method is parameterless and does not reveal a purpose. Instead, it defines a common interface which is implemented differently by its child classes. Consequently, all capabilities are triggered the same way which makes the framework independent of the type and number of capabilities, a particular fixture setup supports. It also allows programmers to define new capability classes in the model without affecting the overall concept. During the fixturing process the fixture coordinator iterates through the capability list of each module and calls the perform()-methods one after another. The class AdjustClampingForce publishes the target force which can be retrieved from the ContactPoint-object, associated with the fixture module. Sensing capabilities first retrieve the current values from the modules by calling their associated subscriber object. If new data is received, the internal data model in the fixture coordinator is updated accordingly. To provide access to the received values for other parts of the system, so-called getter-methods are defined by each capability class. Equally, setter-methods are defined to configure the capability classes with the target values to be published. Figure 6-12 shows

the class diagram for the mentioned classes with a focus on the method interface. For each method its name is specified, followed by the parameter list in brackets. If the method has a return value, the data type of this value is separated by a colon.

```
                          ┌─────────────────────┐
                          │   FixtureModuleCap   │
                          ├─────────────────────┤
                          │   perform()          │
                          └─────────────────────┘
```

AdjustClampingForce	SenseReactionForce	SenseTipPosition	SenseClampingForce	ProvidesRole
getDesiredForce(): Force setDesiredForce(Force f)	getCurrentForce(): Force setCurrentForce(Force f)	getTipPosition(): Point setTipPosition(Point p)	getCurrentForce(): Force setCurrentForce(Force f)	getRole(): Role setRole(Role r)

AdjustTipPosition	SenseBodyPosition	AdjustBodyPosition
getDesiredTipPosition(): Point setDesiredTipPosition(Point p)	getBodyPosition(): Point getSlotClocking(): Clocking getModuleClocking(): Clocking setBodyPosition(Point p) setSlotClocking(Clocking cl) getModuleClocking(Clocking cl)	getBodyPosition(): Point getSlotClocking(): Clocking getModuleClocking(): Clocking setBodyPosition(Point p) setSlotClocking(Clocking cl) getModuleClocking(Clocking cl)

Figure 6-12: Method Interfaces for the Fixture Module Capability Classes

In a similar way, the local software process of each module iterates through its capability list and continuously calls the perform()-methods. This time, actuating capabilities (e.g. AdjustClampingForce) first try to retrieve a new target value from the associated subscriber and then delegate the request down to their nested capability in order to perform the actuation. For sensing capabilities, the procedure is carried out reversely. For example, when the perform()-method of the class SenseClampingForce is called, the capability object first delegates the request to its nested capability until the interface of the device class is triggered to retrieve the current sensor value. The result is returned to the capability object of the fixture module which passes it to its associated publisher object to communicate the current value to the fixture coordinator. A detailed illustration of the described interactions is provided in section 6.5. To access the nested capabilities, specific methods are invoked which are defined in the device capability classes, as shown in Figure 6-13.

```
                          ┌─────────────────────┐
                          │   DeviceCapability   │
                          ├─────────────────────┤
                          │                      │
                          └─────────────────────┘
```

LinearActuationCapability	ApplyForce	SenseDisplacementCapability	SenseForceCapability
actuate(Point targetPosition)	applyForce(Force targetForce)	senseDisplacement(): double	senseForce(): Force

SwingActuationCapability	Locate	Support
actuate(Point targetPosition)		
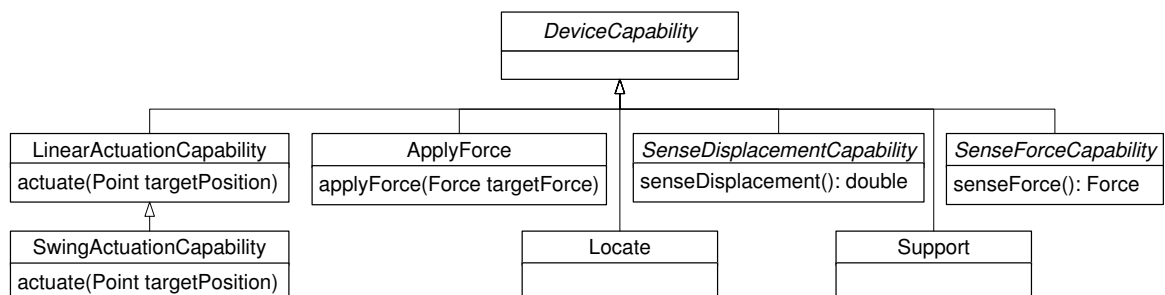
Figure 6-13: Method Interfaces for the Device Capability Classes

If the device capability class also contains a nested capability, the request is delegated further by calling the method interface of the nested capability. This way, the request is delegated down the object hierarchy until the capability object is reached which is attached to the sensor or actuator device object. Here, the method interface of the associated Device-object is called which encapsulates the access to the hardware. Additionally, the actuate()-method defined by the classes LinearActuationCapability and SwingActuationCapability converts the target position into coordinates of the actuator device, using the matrix moduleToDevice which is provided by the device object. An overview of the methods provided by the device classes is shown in the class diagram in Figure 6-14.
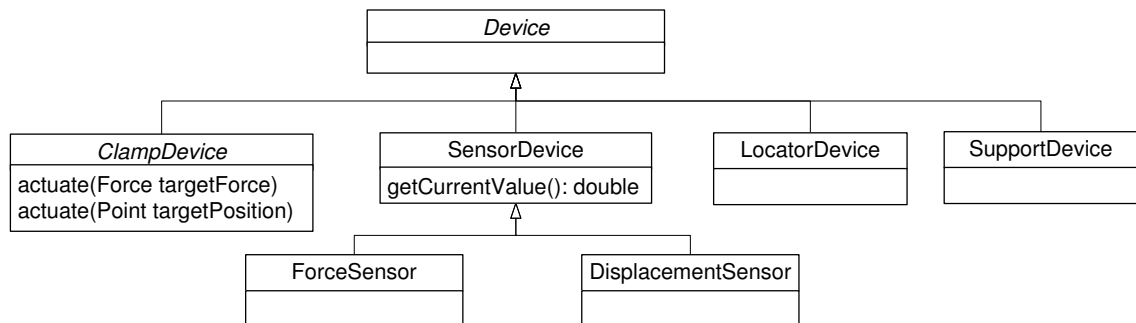
Figure 6-14: Method Interfaces for the Device Classes

To simplify the diagram, the composite pattern (see section 4.4.1) is not shown as it has no impact on the interface definitions. Further, the method interface of locator and support devices is empty because they are typically passive elements without any intelligence.

6.4.3. Library Interface Definition for the Hardware Access

The methods provided by the device classes must not contain the implementation code for the hardware access because this would prevent these classes from being re-used for a variety of devices from different vendors. Instead, the classes are configured with software libraries, tailored for a particular device and vendor. Consequently, all requests are ultimately delegated to the methods offered by devices libraries. This way the hardware access is extracted from the rest of the software framework which makes the framework reusable for several different setups.

For each device type a library interface has been defined according to the following class structure. The parent class IDeviceLib defines the method interfaces for the initialisation of

the device library and its closure. Additionally, the class ISensorLib defines the interface of the method getCurrentValue() which is called to retrieve the current sensor value. Finally, the class IActuatorLib defines the method interfaces for the force and position controlled actuation which expect the target actuation and force values as parameters.

IDeviceLib
initialise(): bool
closeDevice(): bool

ISensorLib
getCurrentValue(): double

IActuatorLib
actuate(Force targetForce)
actuate(double targetActuation)
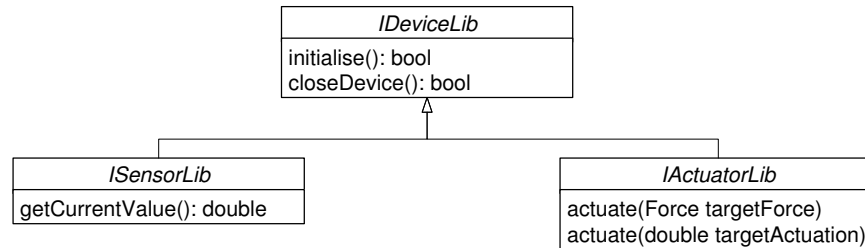
Figure 6-15: Library Interface Definitions

The implementation of these library functions is beyond the scope of this research, since this depends on the specificities of the hardware device in question. Instead, the described class structure must be extended by further child classes which implement the method interfaces, based on the hardware requirements of a particular device. This can be done with any appropriate programming language, such as LabView, C or C++. The initialise()-method must correctly register the I/O channels for the hardware communication and prepare the device for its operation. For actuator devices, this includes the execution of the procedure to find the home position. Similarly, the closeDevice()-method must contain the code to correctly release any used software resources. A typical implementation of the getCurrentValue()-method would access the data acquisition card of the sensor to read a digital voltage value. In the second step, this voltage value is translated into a force or position value, depending on the kind of sensor. For electromechanical actuators, a typical implementation of the positional actuate()-method converts the target actuation value into motor counts and then sends appropriate commands to the motion controller of the device. Concrete examples for the implementation of these methods are described in chapter 7, based on the hardware used for the demonstrator test bed.

6.5. Illustration of the Communication Sequence

To illustrate the previously described interactions during the clamping procedure, this section presents an exemplary setup consisting of one fixture module communicating with the fixture coordinator. The module consists of a force sensor and a linear actuator.

Consequently, during its initialisation routine the software process of the module creates the device and capability objects as described in section 5.2.2. On the top of this hierarchy, the module object is attached with the fixture module capability objects. These objects register the publishers and subscribers with the data topics as described in section 6.3.1. Additionally, each capability contains a reference to its nested capability. The latter can have another nested capability, unless it is connected with the object for the hardware device. Finally, each device object is configured with an object for the library, which inherits from the class structure, described in section 6.4.3. For example, the class ExampleSensorLib shown in Figure 6-16 is a child class of ISensorLib and implements the interface for the specific sensor hardware. Below the local object model of the fixture module is shown. For the sake of simplicity, the diagram is limited to the objects related to the feedback and adjustment of the clamping force.
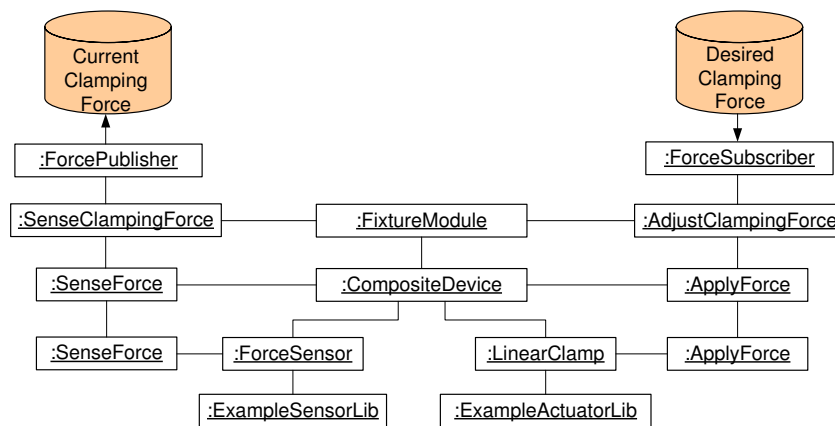
Figure 6-16: Example Object Model of a Fixture Module

During the clamping of a workpiece, the fixture module object continuously calls the perform() methods of all attached capability objects. When the perform()-method of the SenseClampingForce capability is invoked, the request is delegated to its nested capability by calling the method senseForce(). Since this object has another nested capability, the request is delegated further by another call of the method senseForce(). The receiving object is linked with the device object for the force sensor and consequently delegates the request to it by calling the method getCurrentValue(). The device object can access the hardware through the provided library and returns the current clamping force value. After passing the measured value up the object hierarchy, it is published to make it available to the fixture coordinator or other subsystems connected to the communication infrastructure.

The UML sequence diagram unterhalb shows the described sequence. The objects are represented by rectangles on the top of the diagram which are connected with vertical dashed lines. The latter are called life lines and symbolise the time flowing from top to bottom. The execution of methods is represented by oblong rectangles on the life line, thereby showing the sequence of actions. Further details on UML sequence diagrams can be found in Weilkins and Oesterreich [144].
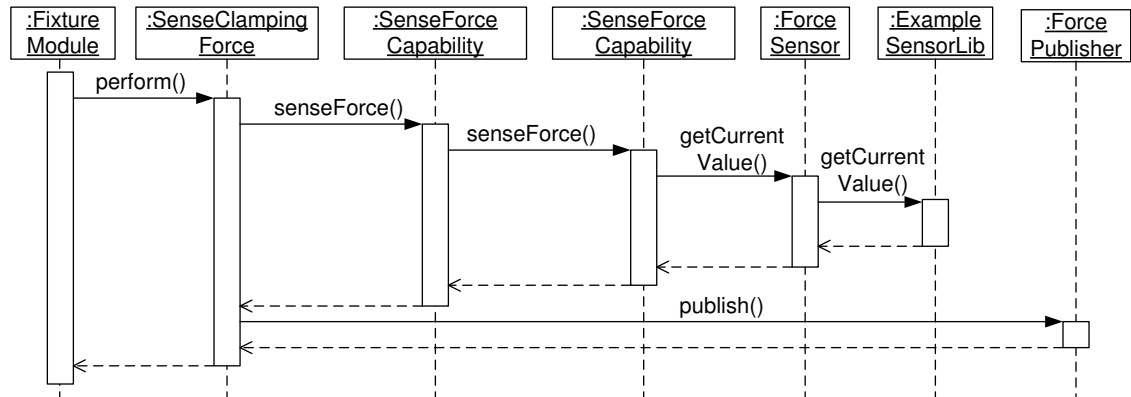
Figure 6-17: UML Sequence Diagram for the Force Feedback in the Module Program

When the perform()-method of the AdjustClampingForce capability is called by the fixture module, it first tries to retrieve a new target force value from its associated subscriber. If a new value is received, it calls the applyForce()-method of its nested capability, passing over the target force value. Since the receiving object has another nested capability, the request is delegated further until the capability object is reached, that has access to the object representing the clamp device. Consequently, the actuate(force)-method of the device object is called which adapts the clamping force by delegating the request to the library, it has been configured with. Figure 6-18 shows the UML sequence diagram for the interactions, carried out during the execution of one perform()-method.
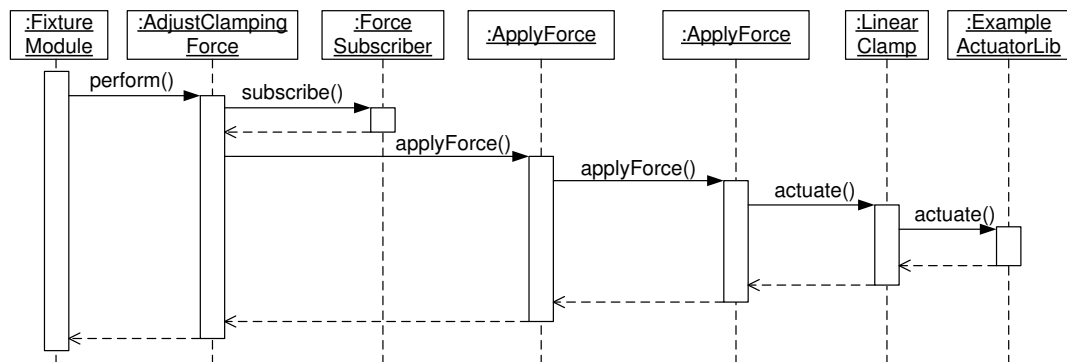
Figure 6-18: UML Sequence Diagram for the Force Adjustment in the Module Program

If no target force is received by the subscriber in the first place, the described sequence is carried with the last received target force which is stored as an attribute in the class AdjustClampingForce. This way, the fixture module does not stop adapting the clamping force in the time interval between two received target force values. Consequently, the adaptation is independent from the frequency the fixture coordinator issues target values and solely depends on the cycle time of the module process. The cycle time is defined as the time which the module needs to execute the perform()-methods of all attached capabilities.

The fixture coordinator software operates independently from the sequence executed in the fixture modules. During its initialisation routine it receives the capability description from the fixture module which results in the generation of the objects shown in Figure 6-19. To simplify the subsequent considerations, the diagram only shows the capabilities for the feedback and adjustment of the clamping force.
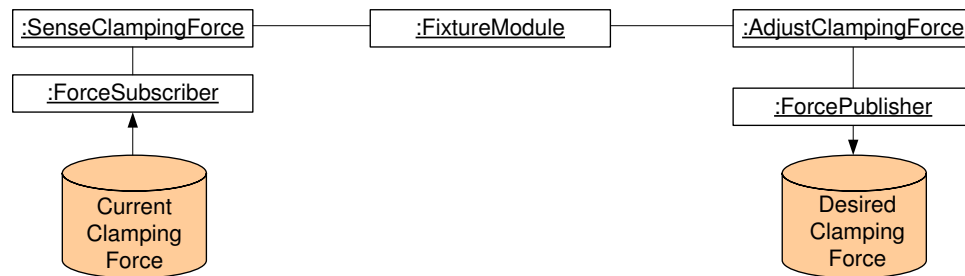
Figure 6-19: Example Object Model in the Fixture Coordinator

During the clamping procedure, the fixture coordinator also continuously iterates through the capability lists of all module objects, and calls the common interface of the perform()-methods. The implementation of this method in the class SenseClampingForce triggers the subscriber object to retrieve the latest sensor update. If new data has been received, the data model is updated accordingly. When the perform()-method of the class AdjustClampingForce is called, the desired clamping direction and the target force value are retrieved. The former can be obtained from the ReconfigurationInfo-object that is linked with the fixture module. The latter is generated by the ForceProfile-object which is attached to the associated contact point of the fixture module. As described in section 4.7.2, the class ForceProfile defines the common interface generateTargetForce() which must be implemented by its child classes. Hence, depending on the implementation of the child

class, the target force value can be generated according to different strategies. For example, a time-driven force profile returns a pre-defined value from a look-up table, based on the elapsed time of the manufacturing process. Other child classes of ForceProfile could return a target force, based on the current tool position or the currently experienced reaction forces on other fixture modules. Ultimately, the target force value is issued by the ForcePublisher object, associated with the capability. To release the clamps, a target value of 0 is published. Figure 6-20 provides the UML sequence diagram for the execution of both, the force feedback and the issuing of new target force values.
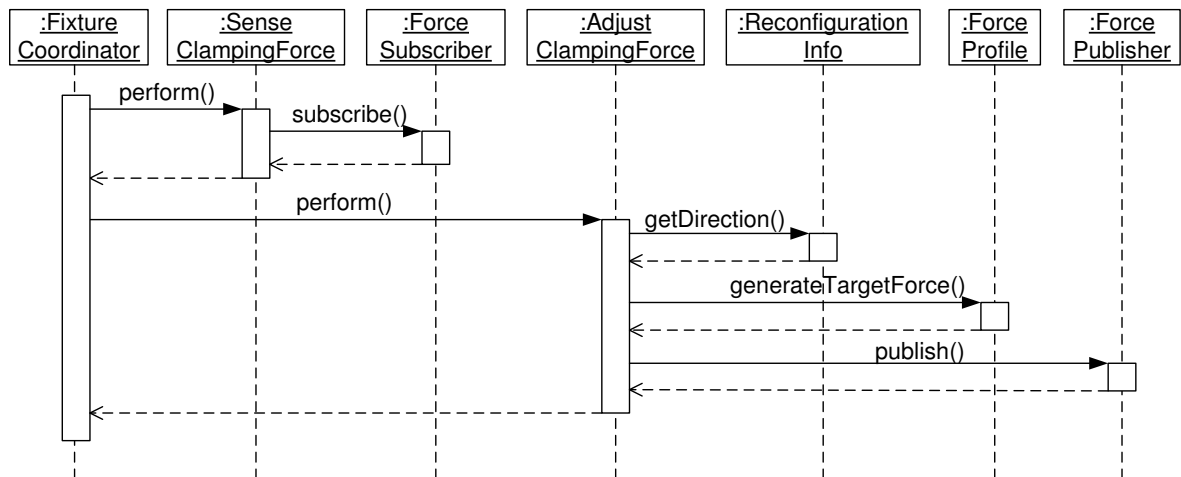
Figure 6-20: UML Sequence Diagram for the Capability Execution in the Fixture Coordinator

The described sequences are carried out continuously by the module software and the fixture coordinator, as they call the perform()-methods of the capability objects. As a result, both communication peers constantly exchange the sensor values and the target values in a loosely-coupled way. The advantage of the described delegation approach lies in its ability to reuse the class structure for different hardware setups. The common interface of the perform()-method allows to trigger all functionalities in the same way and hides the implementation details of lower layers. Consequently, it is possible to enhance an existing module with further capabilities or make changes to the device structure without the need of reprogramming the module software, since these alterations do not affect the common perform()-method interface. Additionally, the framework can be extended by new capability classes without disturbing the described interaction sequence.

6.6. Chapter Summary

A novel communication infrastructure for the data exchange between the fixturing components has been described. The infrastructure is based on the publish/subscribe paradigm and adopts the Data Distribution Service (DDS) which is an emerging communication standard. The required data topics and data types were defined, using the platform-independent Interface Definition Language (IDL). Additionally, the data model elements were extended by a method interface which supports the flexible operation of the fixture modules, based on the delegation approach.

Unlike existing fixturing approaches which are typically restricted to a predefined set of components with hard-wired communication links, the infrastructure makes it possible to dynamically discover the fixture modules with their associated capabilities and to establish the communication channels between them. As a result of the common method interface and the delegation of requests down to the device libraries, the class structure of the data model can be re-used for different scenarios without the need to re-programme the fixture software.

7. Illustration and Verification

7.1.   Introduction

This chapter aims at illustrating the research outcomes by applying the proposed software framework to an experimental test bed. The testbed has been built based on the conceptual design presented in chapter 3 and renders an adaptive fixture with the ability to reposition the clamps on a rail frame. For the operation of this prototype, two software applications for the fixture coordinator and the fixture modules have been developed which implement the object-oriented data model, communicate via the DDS-based communication infrastructure and realise the fixture reconfiguration according to the methodology, described in chapter 5. The programs can be configured with information about the hardware devices, transport components and fixture design parameters and are therefore not limited to the prototype fixture.

To demonstrate the general applicability of the research results, two representative test cases have been selected to show that the software framework can be used for different structural layouts of the test bed hardware. Additionally, it was confronted with different workpieces in order to test a variety of reconfiguration scenarios. The results of these tests indicate the validity of the proposed framework and suggest that the research outcomes can be utilised for other systems in the industrial context.

Section 7.2 describes the physical structure of the test bed, including the characteristics of the sensor and actuator components. The implementation of the software framework is subject to section 7.3. This focuses on describing how the data model, the methodology and the communication infrastructure have been integrated into a working software system that is able to operate the test bed. Finally, the sections 7.4 and 7.5 describe two experiments which demonstrate the basic capabilities of the framework.

7.2. *Description of the Test Bed Hardware*

The starting point for the development of the test bed hardware was the definition of a set of general requirements and constraints. These can be summarised as follows:

- **Reconfiguration capability**

  It shall be possible to reposition the fixture modules automatically in order to accommodate different part sizes and geometries

- **Modular design**

  It shall be possible to re-arrange the components of the test bed in different setups. Additionally, the fixture modules and transport components must be independently controllable.

- **Adaptive clamping capability**

  It shall be possible to apply a predefined clamping force and adapt it according to predefined force profiles.

- **Prismatic workpieces**

  For the initial verification it is sufficient to use simple prismatic workpieces of varying sizes and geometries. In this way, the complexity of the test bed hardware and associated costs can be reduced.

Based on these general requirements an early design concept was developed which is shown in Figure 7-1. The drawing shows a top-down view of a frame with four linear guides along which the fixture modules can be moved.
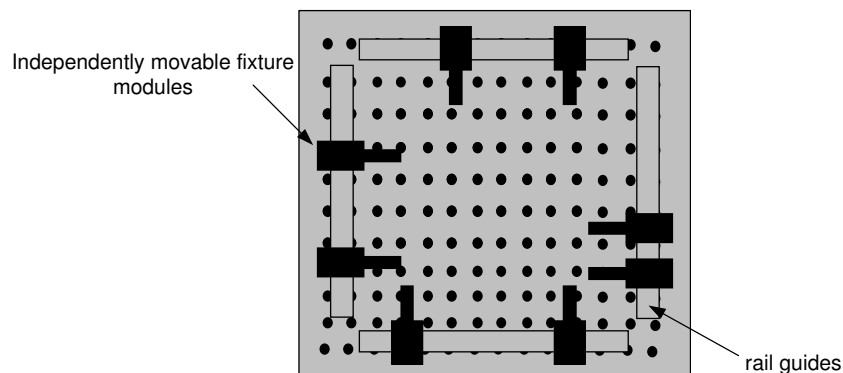
Figure 7-1: Preliminary Concept Drawings for the Prototype

The second step consisted of the definition of further design criteria which were influenced by the requirements of a related research activity, carried out by a another researcher:

- Maximal working envelope of the fixture: 500 x 500 mm

- Working temperature range: -20~70°C

- Maximal applied clamping force: 2500N

- Clamping direction: horizontal (side-clamping)

Finally, the mechanical design for the test bed was developed and the equipment was selected. This task has been performed by another researcher and is therefore not claimed as a contribution of this research..

7.2.1. Equipment Description for Transport Components

In order to reduce equipment cost and development time, it was decided to limit the system to two transport components, instead of using four. This solution allows to demonstrate the reconfiguration capabilities on two sides of the workpiece, while the remaining fixture modules require a manual repositioning. However, due to the modularity of both the hardware design and the software, the system can easily be upgraded to a fully-automated solution which repositions all fixture modules automatically.

The design of the first transport component is shown in Figure 7-2. As can be seen, the solution provides the means to reposition one fixture module automatically through a servo motor and a ball screw mechanism. The structure of the second transport component is similar. However, a second fixture module can be mounted on it which can be repositioned manually on the rail. In this way, it is possible to demonstrate the instantiation of the data model with different types of transport components and to test the collision avoidance algorithm described in section 5.3.4. The decision to achieve the repositioning of the second module manually was made to limit the complexity of the design, thereby reducing cost and development time. A detailed description of the chosen hardware components is provided by the following sections.
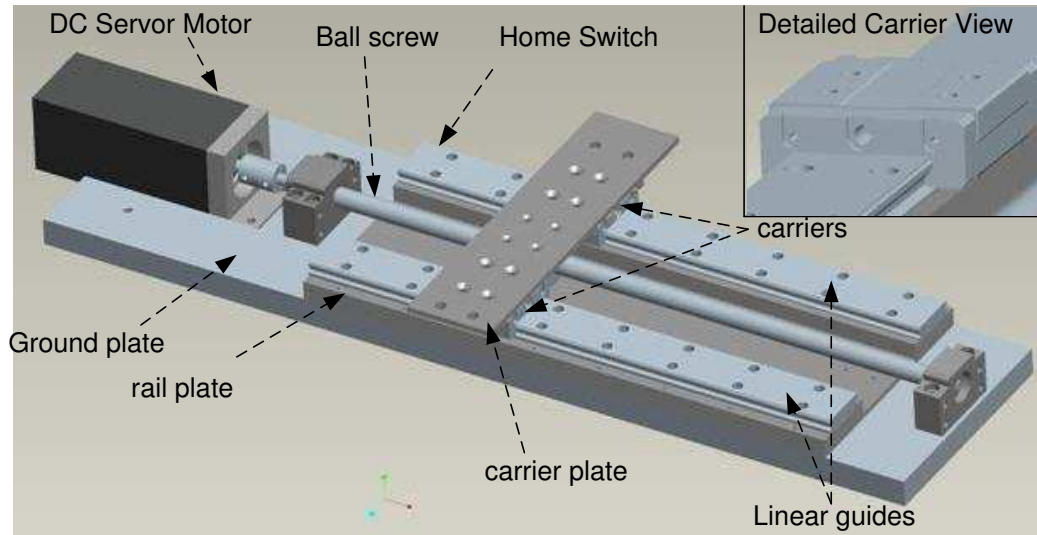
Figure 7-2: Design for a Transport Component with one Carrier

Linear guides and carriers

To guarantee the repositioning capability, two parallel low-friction linear guides are used, on which carriers can slide along. More specifically, the model SHW21-CR-2-ZZ-C1-+400L-P-II from the company THK Co. Ltd. has been selected. The length of one rail guide is 400mm and the span between the two guides is 120mm. The guides are mounted on a customised metal plate, hereafter called rail plate, with a dimension of 400mm x 170mm x 14mm. The carriers on each linear guide are connected by the carrier plate which provides a platform to mount a fixture module. This has a dimension of 250mm x 64mm x 5mm. Additionally, on one side of the linear guides on-off switch has been mounted which indicates the home position of the first carrier during the initialisation routine.

Motor and ball screw

The first set of carriers can actively be moved along the rail by a AC servo motor with a ball screw mechanism which is mounted on the ground plate with a dimension of 660mm x 170mm x 20mm. For the motor, the AC servo motor model AKM23C-ANBNC-01 from the company Danaher Motions, Inc. has been used which includes an internal encoder for the positional feed back. The resolution of the feedback signal is 2000 counts per revolution. The ball screw was supplied by THK Co. Ltd. and the chosen model is BNT1404 which has a lead pitch of 4mm. Therefore, one full revolution of the servo motor equals to a positional displacement of 4mm. From this ratio, the positional resolution of the system can be deducted from the following equation.

$$\frac{4 \text{ mm}}{x \text{ mm}} = \frac{2000 \text{ motor counts}}{y \text{ motor counts}}$$

Replacing parameter y with a value of 1 and solving the equation for parameter x results in 2µm as the smallest possible displacement. The second transport component provides an additional set of carriers which are not connected to the ball screw. Instead, they can only be moved manually. The design drawing of the second transport component is shown in Figure 7-3.
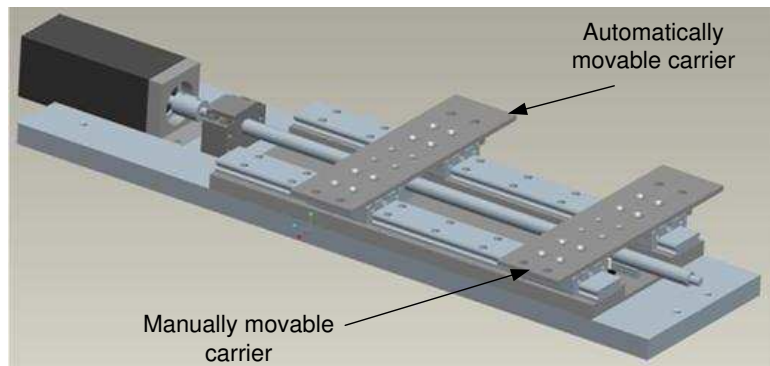
Figure 7-3: Design for the Transport Component with two Carriers

7.2.2. Equipment Description of one Fixture Module

To demonstrate the adaptive clamping capability, two identical fixture modules were built which consist of a linear actuator, driven by a servo motor. The actuator tip is equipped with a Kistler force sensor, described further below. A photograph of one linear actuator is shown in Figure 7-4.
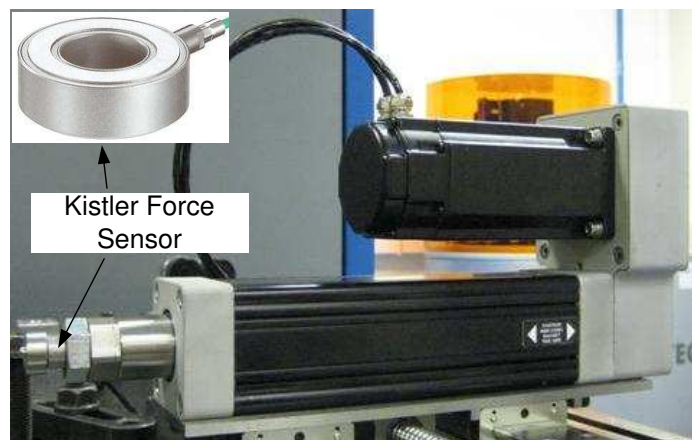
Figure 7-4: Linear Actuator with Mounted Force Sensor

The chosen actuator model is EC2-BK235-100-16B-60-MS6M-FT1M from Danaher Motion, Inc., whose technical specifications are listed in Table 7-1.

Category	Value
Motor type	Brushless AC Servo Motor
Screw type	Ball screw
Screw lead pitch	16mm
Gear factor	1:10
Maximum load capacity	3600 N
Self-locking	yes
Maximum No Load speed	1280 mm/s
Stroke	60mm
Positional feedback resolution	8µm (2000 counts/ revolution)

Table 7-1: Specification Summary for the Linear Actuator

As the force sensor, the Kistler PZT 9101A single component load washer has been selected because these components were readily available in the laboratory. Additionally, this type of force sensor has the advantage of a compact design, which allows to mount it near the actuator tip as shown in Figure 7-4. A selection of the specification details of the sensor is provided by Table 7-2.

Category	Value
Measuring range	0 – 20KN
Rigidity	1.8 KN/µm
Sensitivity	-4.3 pC/N
Dimensions (H x D x d)	6.5 x 14.5 x 8 (mm)

Table 7-2: Specification of the Kistler Force Sensor

7.2.3. Equipment Description for the Control Hardware

To operate the selected sensor and actuator components, a number of control hardware components are required, such as servo drives, a motion control card and a charge amplifier. These have been selected according to the recommendations of the hardware suppliers. Figure 7-5 presents a high-level block diagram of the used components and their connections between each other. All software applications run on an ordinary office PC with the operating system Windows 2000 installed. The PC is equipped with a PCI Motion Control Card (model number: NI PCI 7344) from the company National Instrument which can be accessed by a software interface to perform the motion control of up to 4 axes simultaneously.
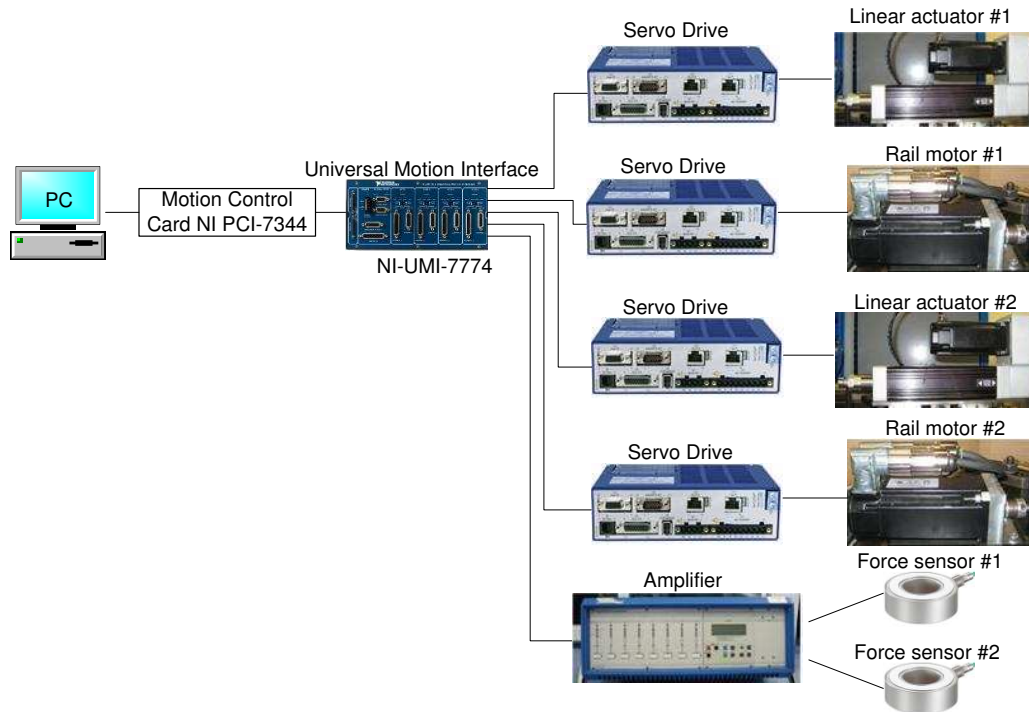
Figure 7-5: Block Diagram for the Control Hardware Components

To connect the motion control card with the servo drives from Danaher Motions, Inc., a so-called Universal Motion Interface (UMI) board is required which transfers the voltage signals from the motion controller to the servo drives. This has also been purchased from National Instrument in order to guarantee compatibility with the motion control card. The model number is NI UMI-7774. Each servo motor is controlled by its own drive unit which transforms the voltage signal from the UMI into current, required by the motor to generate the torque. The servo drive model ServoStar S20360-VTS from Danaher Motions, Inc. has been selected for the linear actuators, while the servo motors for the rails are driven by the ServoStar S20260-VTS unit from the same supplier. The two drive models only differ in the output current they are able to generate. Technical details on both devices can be obtained from [168]. Finally, the two force sensors are connected with the multichannel charge amplifier Kistler 5017B1800 which generates a voltage signal corresponding to the force that is experienced by the sensor. The amplifier has been calibrated to generate a voltage signal between 0 and +10V for a force range of 0 to 2500 N. Thus, every 1V represents a force increase of 300N. This signal is fed into the Analog-to-Digital converter (ADC) of the UMI-board which has a resolution of 12bit. As a result, the force sensing resolution can be deducted from the ratio, defined in Equ.7-2, as ~0.61N.

$$\frac{2500 \text{ N}}{x \text{ N}} = \frac{2^{12} - 1}{y} \qquad \textbf{(Equ. 7-2)}$$

7.3.   Description of the Prototype Software

To demonstrate the research results, two software applications have been developed. The first application is the software program for one fixture module. This program is configured with the software libraries for the device access and an XML-file containing the module description. During the test procedure multiple instances of these programs are launched, depending on the number of fixture modules that exist. The second application consists of the fixture coordinator software which has been enhanced with a graphical user interface (GUI). Additionally, this software program contains the logic for the transport components as separate threads. In computer science, a thread is a concurrently running task within a process [169]. Consequently, the transport component threads are carried out in parallel to the threads of the fixture coordinator and the graphical user interface, thereby preserving their independence. Figure 7-6 presents a block diagram of the interacting software components.
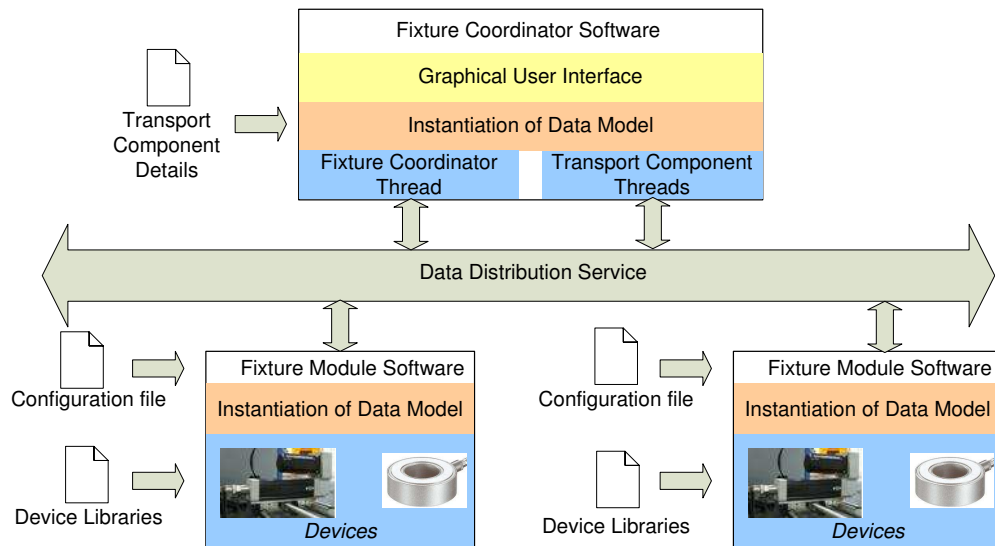
Figure 7-6: Overview on the Software Processes for the Prototype

The software programs for the fixture coordinator and the fixture modules have been implemented in the programming language C++, using the Microsoft Development Environment 2003, version 7.1.3088. The graphical user interface of the fixture coordinator has been implemented with the Microsoft Foundation Classes (MFC) which is a widely-

used framework for the programming of Windows applications [170]. Additionally, a number of open-source software libraries have been utilised which are listed in the table unterhalb. An "X" indicates that a certain library is used for the corresponding application.

Library Name	Purpose	Fixture Module Software	Fixture Coordinator Software
tinyXML [171]	Parsing and interpretation of XML-files	X	
Matrix TCL Lite 2.0 [172]	Matrix calculations	X	X
RAPID 2.01 [165]	Collision Detection during the reconfiguration sequence		X

Table 7-3: Utilised Third-party Software Libraries

Finally, communication between the software processes and threads is achieved via the commercially available DDS-platform RTI-DDS 4.1e from Real-Time Innovations, Inc.

7.3.1. Generation of the Publisher/Subscriber Classes

To generate the classes for the DDS-communication, the IDL-definition of the data types described in chapter 6 have been written in the file exampleAppl.idl. The content of this file can be found in Appendix B. Based on the IDL-definition, the DDS internal C++ classes are generated automatically by the tool rtiddsgen which is part of the DDS-platform. To run the tool, the following command line must be entered.

```
rtiddsgen –language c++ exampleAppl.idl
```

Listing 11: Command Line for the Generation of the DDS Classes

The resulting files are exampleAppl.cpp, exampleAppl.h, exampleApplSupport.cpp, exampleApplSupport.h, exampleApplPlugin.cpp and exampleApplPlugin.h. These files need to be compiled and linked to the source code of both, the application for the fixture modules and the fixture coordinator.

7.3.2. Configuration File Settings

The general content of the module description file has been discussed in section 5.2.1. This section describes the contents of this file for the fixture modules used in the test bed. Apart from the numeric identifier and the module dimension, the XML-file contains three <device>-sections for the linear actuator, the displacement sensor and the force sensor. For the linear actuator, the following definitions have been made with respect to the spatial relation of its local coordinate system. As can be seen in Figure 7-7, both the coordinate

systems of the fixture module and the actuator device have been placed in the centre of the actuator tip when the latter is not extended. This way, the value of the x-axis directly correlates with the current displacement of the actuator. These definitions are reflected by the values in the <spatialdesc>-block for the actuator device which can be seen on the right side in Figure 7-7.


```
<kind>LINEAR_CLAMP</kind>
<id>1</id>
<description>Nothing</description>
<spatialdesc>
        <x>0</x>
        <y>0</y>
        <z>0</z>
        <rotx>0</rotx>
        <roty>0</roty>
        <rotz>0</rotz>
</spatialdesc>
<isLockable>true</isLockable>
```
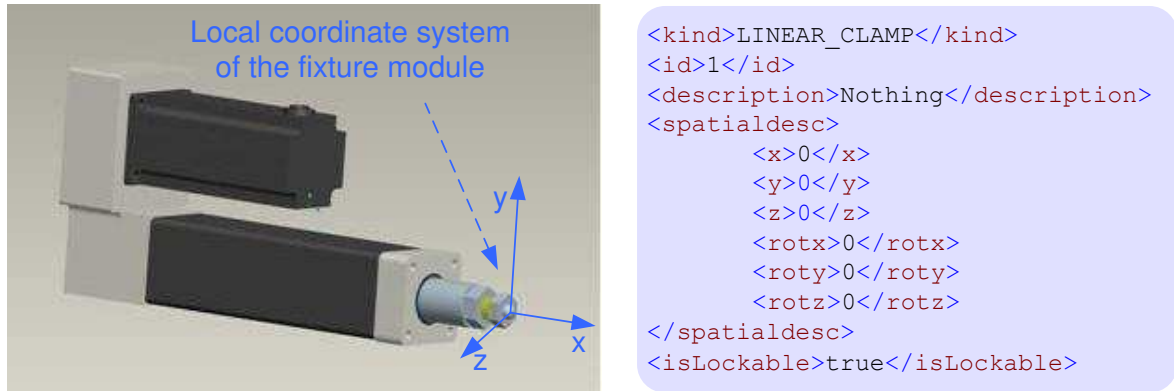
Figure 7-7: Definitions of the Local Coordinate Systems for the Fixture Modules

Further, the chosen actuator model is self-locking as a result of its internal structure. This is reflected by setting the <isLockable>-tag to true. The capability descriptions for the actuator device can be summarised as follows:

- Stroke range 0 to 60mm with a resolution of 8µm

- Clamping Range 0 to 2500 N in push direction

- Maximum allowable reaction force: 5000 N

The values for the stroke range and the maximum allowable reaction force can be obtained from the design specification of the actuator. Finally, the <library>-block and the <library-parameters>-block provide the name of the device library and the details that it requires in order to function correctly. This includes the identifier of the motion controller card which has a value of 1. Secondly, the identifier of the motion axis must be provided, which is 1 for the first module and 2 for the second module. Thirdly, the ADC-channel identifier to acquire the current force are provided which has a value of 1 for the first module and a value of 2 for the second module. Finally, the encoder resolution and the pitch is required to correctly convert between displacement values in mm and motor counts. Appendix A provides the complete listings of both configuration files. The XML-block for the displacement sensor is an example of how the proposed data model decomposes the physical setup into separate functional components. Even though, the displacement

feedback is provided internally by the linear actuator, a separate <device>-block is defined in the XML-file which results in the generation of a separate object in the data model. The reason for this is that the class used for the representation of the linear actuator does not provide a method interface to retrieve sensor values. Finally, the information for the force sensor is provided in the last <device>-block which is based on the calibration parameters of the charge amplifier and the resolution of the ADC of the UMI-board. Thus, the force feedback capability of the device is defined by a measuring range between 0 and 2500 N. The resolution which has been calculated earlier, is rounded up to +/-1N. The device library of the sensor requires the identifier of the motion control board and the channel number of the ADC on the UMI which retrieves the sensor signal. For the first module, this is the channel number 1, whereas the force sensor of the second module is attached to the channel 2.

7.3.3. Device Library Implementation

Three software libraries have been created which accomplish the access to the hardware devices. These have been implemented as Dynamic Link Libraries (dll) in the programming language C++, using the Microsoft Development Environment 2003, version 7.1.3088. This allows other applications, such as the fixture module software, to be dynamically configured with them at run-time. The created library files are listed below:

- **DisplacementSensor_EncoderS200Lib.dll**

  Implements the class DisplacementSensor_EncoderS200Lib as a child class of ISensorLib to retrieve the current displacement from the encoder of the linear actuator. The method getCurrentValue() accesses the motion controller to read the current position in motor counts from the encoder. The retrieved value is converted into a displacement, using equ. 7-1.

- **KistlerForceSensor_UMI_ADC.dll**

  Implements the class KistlerForceSensor_UMI_ADC as a child class of ISensorLib to retrieve the current force from the Kistler force sensor. For this the method getCurrentValue() first accesses the ADC of the UMI-board. The obtained value is then converted into a a force in Newton, using equ. 7-2.

- **NI_UMI7774_S200VTS.dll**

  Implements the class NI_UMI7774_S200VTS as a child class of IActuatorLib for the linear actuator with the UMI7774-board. The adopted algorithm for the force control is based on changing the actuators position in response to the force feedback. This is realised by two control loops, which are illustrated in the diagram below. The inner control loop is performed by the motion controller hardware and ensures that the target position is achieved. Additionally, there is an outer control loop implemented in the library software which continuously reads the current force feedback from the sensor and issues new target positions. The same approach has been used for the adaptive fixture developed at the National University of Singapore [49] which has been described as the state-of-the-art in adaptive fixturing in chapter 2.
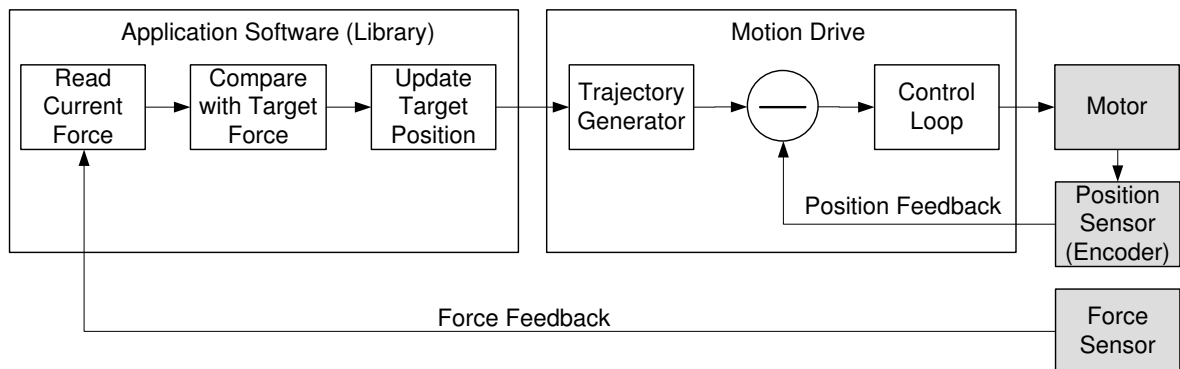
Figure 7-8: Block Diagram for the Force Control Algorithm

The source code of all libraries is provided in Appendix C. In order to allow the fixture module application to create the software objects of the library classes, each dll-file provides the method createLibraryInstance() whose signature is provided below:

```
void * createLibraryInstance(TiXmlNode * node)
```

Listing 12: Interface for the Method createLibraryInstance()

The method expects the XML-node with the relevant <library-parameters>-block for the library from the module configuration file. Since each library is tailored to a particular device, it knows how to interpret the details of the XML-node and can therefore extract the required information to create the library object. The advantage of this approach is that it allows the definition of different sets of configuration parameters for the device libraries,

depending on the device type, vendor-specific details, peripheral equipment and numerous other influencing aspects. For example, the actuator library used for the test bed requires the numeric identifiers of the motion controller card and the motion axis, as well as the values for the encoder resolution and the pitch of the ball screw. Clearly, another actuator type from a different vendor would require a different set of configuration parameters. By delegating the object generation to the library itself, the software framework becomes independent from the specificities of the hardware devices. The code for the createLibraryInstance()-method of each library is also provided in Appendix C. The fixture module software calls this method for each device library it has been configured with, as shown in Listing 13. The first line loads the dll-file with the name provided by the variable library. If this was successful, the next step consists of getting the memory address of the createLibraryInstance()-method which is accomplished by lines 5 and 6. Finally, if the address could be found, the method is called with the XML-node of the relevant <library-parameters>-block as a parameter. The return value of the method is converted into an object of the class IDeviceLib (see section 6.4.3) which allows the module to invoke the initialise()-method of the library object during its initialisation routine.

```
1       //load library
2       HINSTANCE lib = LoadLibrary(library);
3       //load function
4       if (lib){
5           createLibraryFunction = (CreateLibraryInstanceFunction) GetProcAddress(lib,
6                                                          "createLibraryInstance");
7               if (createLibraryFunction){
8                   //get pointer to newly created object
9                   deviceLib = static_cast< IDeviceLib* > ( createLibraryFunction(node ) );
10              }
11      }
```

Listing 13: Code Example to Load a Device Library

The initialise()-method of actuator library carries out a reference move in order to find the home position. Conversely, the initialise()-methods of the libraries for the force and displacement sensors are implemented empty, since the test bed setup does not require any allocation of software resources to operate these devices.

7.3.4. Implementation Overview of the Fixture Module Software

The software of the fixture module constitutes a skeleton program which can be configured with a module description file and device libraries. Consequently, no additional development effort is necessary for this application when new fixture modules are

introduced or hardware devices are changed. The application is started in a command line interpreter with the following syntax. The parameter config-file specifies the path and name to the XML-file containing the module description.

```
FixtureModuleAppl.exe [config-file]
```

Listing 14: Syntax to Start the Fixture Module Application from a Command Line Interpreter

The program first parses the xml-file, using the library tinyXML [171] and generates the object model for the devices and the capabilities, based on the acquired information. This is accomplished according to the rules described in section 5.2.2. For the test bed, the program generates one FixtureModule object which is attached with the objects of the capability classes AdjustTipPosition, SenseTipPosition, AdjustClampingForce, SenseClampingForce, SenseReactionForce and ProvidesRole. Further, each of the created capability objects initialises its publisher/subscriber objects. This procedure is outlined below for the position publisher of the SenseTipPosition-capability.

```
1  //create DDS-publisher
2  this->publisher = participant->create_publisher(DDS_PUBLISHER_QOS_DEFAULT,
3                                            NULL, DDS_STATUS_MASK_NONE);
4  //register data topic
5  this->topic = participant->create_topic(this->topicName, this->type_name,
6                            DDS_TOPIC_QOS_DEFAULT, NULL, DDS_STATUS_MASK_NONE);
7  //register data writer
8  DDS_DataWriterQos dwqos;
9  publisher->get_default_datawriter_qos(dwqos);
10 this->dw = publisher->create_datawriter(this->topic, dwqos, NULL, DDS_STATUS_MASK_NONE);
```

Listing 15: Source Code Extract from the Method initialise() of the Class PositionPublisher

The first step consists of creating the DDS-publisher object by invoking a method of the so-called domain participant which is shown in the first three lines in the listing above. The domain participant is an object provided by DDS which acts as an entry point to the service because it is used to create other objects, namely the publishers, subscribers and the data topics. The second step consists of creating the data topic. For this, the method create_topic() of the domain participant is invoked with the topic name (*"Current Tip Position"*) and the data type name ("Position*"*) as parameters. The complete reference for the other method parameters can be obtained from [173]. Finally, as can be seen in the last three lines, the just created DDS-publisher object is used to create the DataWriter-object which is registered with the topic. After the completion of the initialisation sequence for all capabilities, the module is connected to the communication infrastructure and it can publish

one entry of the data type ModuleCapDefinition in order to indicate its existence to other applications. Finally, the program enters a loop which continuously calls the perform()-methods of the fixture module capabilities, thereby exchanging data with other systems and accomplishing tasks by delegating requests through the object hierarchy. Listing 16 shows the source code for this loop which is executed until the fixture module is switched off.
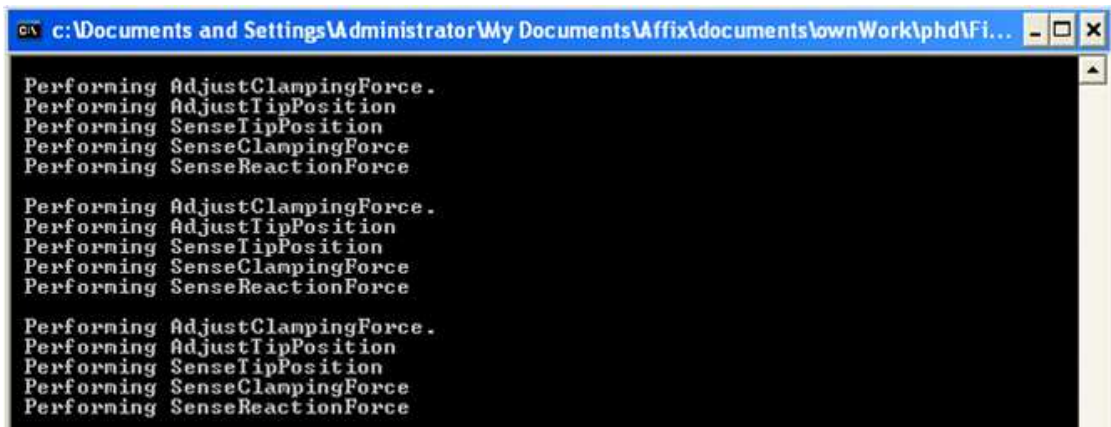
```
1 while(1){
2       //call the perform method on all fixture module caps...
3       int j = 0;
4       for(j=0; j<this->capabilityList.size(); j++){
5               ((FixtureModuleCap *)capabilityList[j])->perform();
6       }
7 }
```

Listing 16: Source Code for the Continuous Execution of the Module Capabilities

As can be seen in line 5, the program only invokes the common interface of the perform()-method which is defined in the parent class FixtureModuleCap. As a result, the application is independent from the implementation details further down the object hierarchy and can therefore operate with arbitrary fixture module configurations. Moreover, it allows to introduce new capability classes without the need to change the rest of the program. The application output of the fixture module program during the execution of the loop is provided in Figure 7-9.

Figure 7-9: Screen Shot of the Fixture Module Program During its Execution

7.3.5. Implementation Overview of the Fixture Coordinator Software

Similar to the module software, the application for the fixture coordinator is a skeleton program which means that it is not limited to the test bed hardware. Instead, it can be

configured with the position and orientation of arbitrary numbers of transport components, fixture modules and the fixture design information. Consequently, changes of the fixture hardware do not require additional programming effort for the fixture coordinator software. The first step during the start of the application, is the generation of the object model for the transport components, based on the provided configuration data. For the test bed, it was decided to provide this information directly in the source code, instead of utilising XML-files. This decision reduced the programming effort by avoiding the development of a second XML interpreter, yet preserved the ability to test different transport component configurations by defining a set of test cases in the source code. At the start of the application one of these test cases must be selected and depending on that, the program generates the object model for the transport components. The listing unterhalb shows extracts for the object generation for a test case, labelled TESTCASE_6.

```
1    case TESTCASE_5:
     ...
3    case TESTCASE_6:
4       //first build the transport component objects...
5       TransportComponent * rail1 = new TransportComponent(idCounter->getNextId(),
6                                                     continuous, onedimensional);
7       //set the spatial description of the TC relative to the global coordinate system
8       rail1->setSpatialDescriptions(0,0,0,0,0,0);
9       //create a slot object for the rail...
10      Slot * slot1 = new Slot(*rail1, idCounter->getNextId());
     ...
21      RepositionCapability * repositionCap = new RepositionCapability(idCounter->getNextId(),
22                                                     rail1->getId(), slot1);
23      WorkSpace * ws = new WorkSpace();
24      ws->linearRange_x.max = 548.5;
25      ws->linearRange_x.min = 212.5;
26      ws->linearRange_x.resolution = 0.002;
27      ws->linearRange_x.unit = UNIT_MILLIMETER;
     ...
```

Listing 17: Configuration with Transport Component Details

First, one object of the class TransportComponent is created in line 5 and its spatial relation to the global coordinate system is defined in line 8. In this case, both coordinate systems are coincident, since the values for the translational and rotational parts of the coordinate transformation are all zero. Line 10 shows the creation of an object of the class Slot which stands for one carrier. The creation of a capability object is outlined in the lines 21 to 27. When all objects for one transport component are generated, the attached capability objects initialise their publisher/subscriber objects. After this, a new thread is started which continuously iterates through the capability list of the transport component and invokes the perform()-methods.

The second essential step consists of the initialisation of the subscriber for the module descriptions. This procedure is similar to the steps outlined for the publisher, described in the previous section and is therefore not explained in detail. Whenever a new module description is retrieved, the information is used to create a new object of the class FixtureModule and to attach it with adequate capability objects. All created objects are displayed immediately in two separate lists on the main window of the GUI which is shown in Figure 7-10.
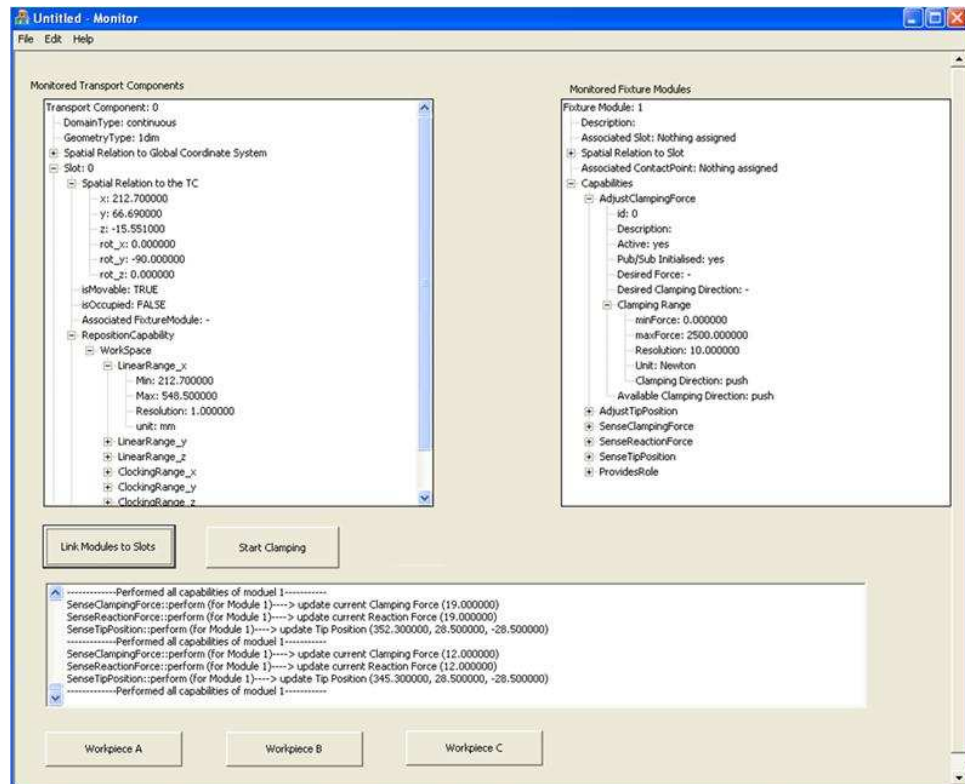
Figure 7-10: Screen Shot of the Main Screen of the GUI

The left list contains the existing transport components while the right list displays all discovered fixture modules with their capabilities. Below the list boxes, an area with two buttons and a text field exists. The left button opens another dialog window which allows the operator to link the fixture modules with the slot objects. The button on the right is used to start and stop the clamping of a workpiece. The text field was used during the development phase to display debug messages. Finally, on the bottom of the dialog three buttons have been placed to demonstrate the reconfiguration procedure for different test workpieces. If one of the buttons is activated, the software retrieves the predefined design information for the workpiece and starts the reconfiguration method as described in section

5.3. Whereas in an industrial environment the design information should be retrieved from a data base, it was decided to implement different sets of design parameters directly in the source code. In this way, the effort for the development of a data base interface could be saved.

The second dialog window of the application appears when the operator clicks on the button, labelled with "Link Modules to Slots". As described in section 5.2.3., this step is necessary because without being linked to the slots, the module positions and orientations are unknown. The layout of the dialog is shown in Figure 7-11.
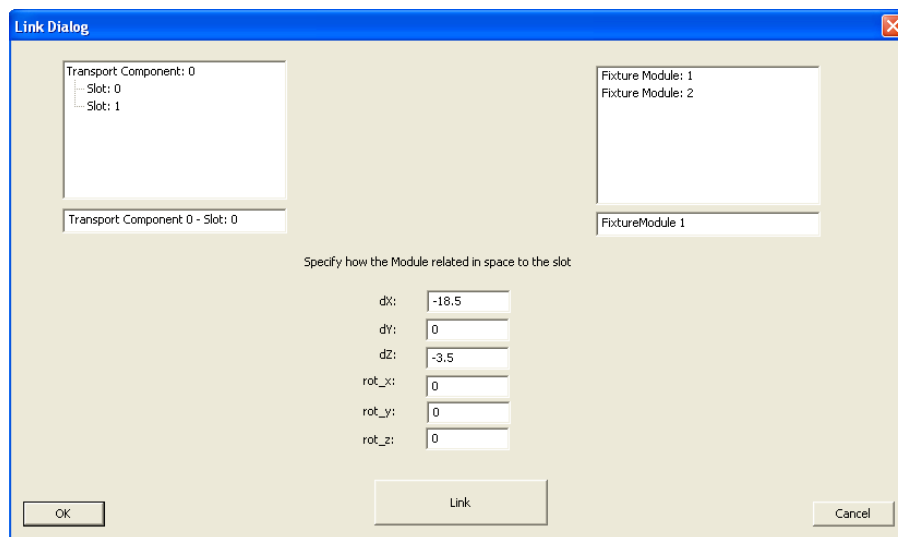
Figure 7-11: The GUI Dialog to Link Fixture Modules with Slots

The list on the right shows all free slots of the transport components, while the list on the right displays the unlinked fixture modules. The operator can select one slot and one fixture module at a time which are additionally shown in the text fields below the list boxes. The essential step is to provide the software with the spatial description of the fixture module, relative to the slot's local coordinate system. For this purpose, six input fields are provided to enter the translational and rotational parts, required by the coordinate transformation. When the "Link"-button in the middle is pressed, both objects are taken away from the lists in order to avoid multiple links. Additionally, as described in section 5.2.3, the fixture module is extended with the capabilities to adjust and sense the body position, based on its connection with the slot.

7.4. *Testing of the Fixture Reconfiguration with one Transport Component*

7.4.1. Objectives

In the first experiment, the fixture setup consists of just one rail which carries the two fixture modules. The example workpiece is a steel plate with a dimension of 250.0 mm x 51.1 mm x 10.0 mm which is rigidly screwed on a frame. As a result, the fixture modules are not required to secure and clamp the workpiece. Instead, they are used to apply dynamically changing clamping forces on two different points on the plate. Therefore, the aim of the experiment is not to demonstrate a complete fixture consisting of locators and clamps. Instead, it validates:

- The automatic discovery of fixture modules by the fixture coordinator.
- The automatic reconfiguration of the fixture modules from arbitrary initial positions to predefined target positions, including the avoidance of collisions
- The configuration of the fixture modules with predefined force profiles from the fixture design.
- The dynamic adaptation of the force over time.

7.4.2. Configuration Details

The software programs for the two fixture modules are configured with the files ModuleDescription_Module1.xml and ModuleDescription_Module2.xml, respectively. Both files are identical, except that different module-ids are provided, namely 1 for the first module and 2 for the second. Additionally, the identifiers for the motion axis and the ADC-channels differ. Appendix A contains the contents of these files.
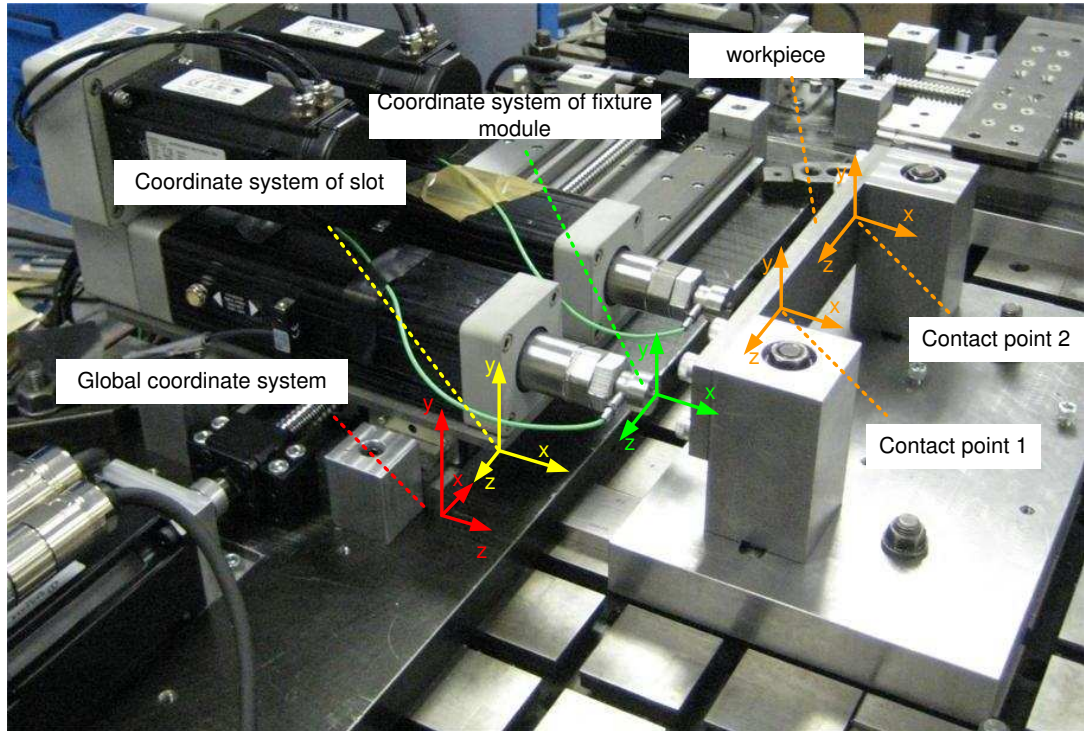
Figure 7-12: Test Setup for the First Experiment

The fixture coordinator is configured with the following information.

- The origin of the global coordinate system is set in the bottom corner of the rail plate as shown in Figure 7-12

- The local coordinate system of the transport component is defined as coincident with the global coordinate system. Consequently, all values for the spatial description of the transport component object are zero.

- Configuration data for first slot

  o The local coordinate system is placed in the bottom corner of the carrier plate as shown in Figure 7-12 and is rotated by 90° about the y-axis in clockwise direction. This location has been selected to simplify the measurement of the distances to the global coordinate system, using a micrometer and high-accuracy gage blocks from the company Cromwell Metrology. The results of these measurements are reflected by the values for x, y and z, given below. The value for x is based on the distance between the global coordinate system and the carrier after the latter has been moved to its home position during the initialisation routine of the transport component.

When the home switch of the rail triggers, it can be measured that this distance is 13.5mm.

- x: 13.5mm   rot_x: 0°
- y: 35.0mm   rot_y: -90°
- z: 25.0mm   rot_z: 0°

o Based on the dimension of the carrier plate, the attribute boundingBox is defined by the following points

- p1: (0.0/0.0/0.0)
- p2: (-250.0/5.0/-64.0)

o The Reposition-capability is instantiated with the following details. As can be seen, the minimum and maximum values for the attributes linearRange_y and linearRange_z are equal, since the slot can only be repositioned along the x-axis. Consequently, the resolution values for these attributes are not relevant and have been set to a default of 1mm. The maximum value for the displacement along the x-axis is 336.0, as a result of subtracting the width of the carrier (64mm) from the length of the rail (400mm).

- linearRange_x: 13.5 mm to 336.0 mm, resolution: 0.002mm
- linearRange_y: 35.0 mm to 35.0 mm, resolution: 1.0mm
- linearRange_z: 25.0 mm to 25.0 mm, resolution: 1.0mm
- The setup does not allow any rotations of the slot itself or the mounted fixture module. Consequently, all elements for the attribute slotClockingRanges and moduleClockingRanges are set to zero.

o The SensePosition-capability is instantiated with the following details:

- posX: 13.5 mm to 336.0 mm , resolution: 0.002mm
- posY: 35.0 mm to 35.0 mm , resolution: 1mm
- posZ: 25.0 mm to 25.0 mm, resolution: 1mm
- Since the setup does not allow any rotations of the slot or the module, all elements of the attributes slotClockingX, slotClockingY, slotClockingZ, moduleClockingX, moduleClockingY, and moduleClockingZ are set to zero

- Configuration data for second slot

- o The local coordinate system is placed in the bottom corner of the carrier plate as shown in Figure 7-12 and is rotated by 90° in clockwise direction. At the start of the experiment the carrier is manually moved to a start position, defined by the spatial description details below.
  - x: 97.5mm   rot_x: 0°
  - y: 35.0mm   rot_y: -90°
  - z: 25.0mm   rot_z: 0°
- o Based on the dimension of the carrier plate, the attribute boundingBox is defined by the following points
  - p1: (0.0/0.0/0.0)
  - p2: (-250.0/5.0/-64.0)
- o The Reposition-capability is instantiated with the following details:
  - linearRange_x: 13.5 mm to 336.0 mm, resolution: 1.0.mm
  - linearRange_y: 35.0 mm to 35.0 mm, resolution: 1.0mm
  - linearRange_z: 25.0 mm to 25.0 mm, resolution: 1.0mm
  - The setup does not allow any rotations of the slot itself or the mounted fixture module. Consequently, all elements for the attribute slotClockingRanges and moduleClockingRanges are set to zero.
- o The SensePosition-capability is instantiated with the following details:
  - posX: 13.5 to 336.0, resolution: 1.0mm
  - posY: 35.0 to 35.0, resolution: 1.0mm
  - posZ: 25.0 to 25.0, resolution: 1.0mm
  - Since the setup does not allow any rotations of the slot or the module, all elements of the attributes slotClockingX, slotClockingY, slotClockingZ, moduleClockingX, moduleClockingY, and moduleClockingZ are set to zero

Since the repositioning of the second slot is performed manually, the related capability object is created with a resolution of 1mm which is an estimate of what is achievable by manually moving the carrier. The Reposition-capability is implemented as a dummy which opens a dialog box, asking the operator to move the slot to the target position. After the

dialog is closed, it is assumed that the slot has been repositioned correctly and the values for the current position are updated in the data model. Similarly, the SensePosition-capability is also implemented as a dummy. Instead of accessing a sensor device, it simply returns the current values for the slot position from the data model. The resolution for the feedback is also set to 1mm.

Furthermore, the fixture coordinator has been configured with the following fixture design details, in the form of ContactPoint objects. The values for the spatial description relative to the global coordinate system have been retrieved through manual measurements, using a micrometer and gage blocks.

- ContactPoint 1
    - Spatial Description
        - x: 120.0mm   rot_x: 0°
        - y: 68.6mm   rot_y: -90°
        - z: 112.2mm   rot_z: 0°
    - Role: Clamp
    - ForceProfile: time-dependent step function as shown in Figure 7-13
- ContactPoint 2
    - Spatial Description
        - x: 220.0mm   rot_x: 0°
        - y: 68.6mm   rot_y: -90°
        - z: 112.2mm   rot_z: 0°
    - Role: Clamp
    - ForceProfile: time-dependent step function as shown in Figure 7-13
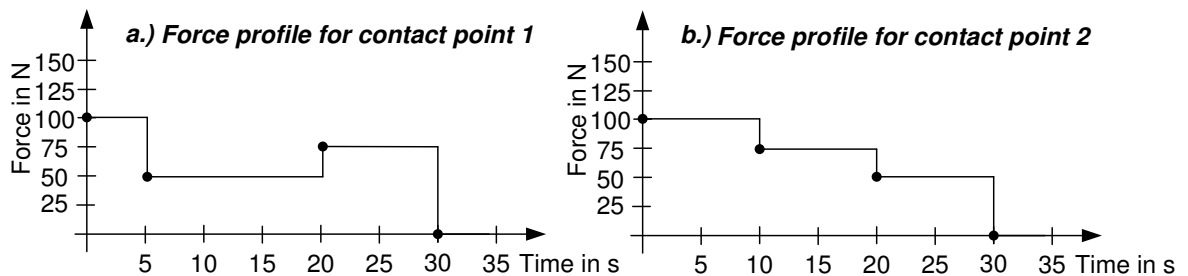
Figure 7-13: Force Profiles for (a) Contact Point 1 and (b) Contact Point 2

7.4.3. Testing Procedure

Table 7-4 shows the sequence of actions during the experiment and the expected behaviour. As can be seen, the first module is started before the fixture coordinator is launched (steps 1 and 2). This has been done in order to demonstrate the capability of the communication infrastructure to redistribute the module capabilities, as described in section 6.3.3. When the operator links the fixture modules with the slots (steps 3 and 6), the following values are used.

- x: 64.1 mm   rot_x: 0°
- y: 33.6 mm   rot_y: 0°
- z: -31.8 mm   rot_z: 0°

These values have been retrieved using the gage blocks described before and are based on the settings for the local coordinate systems of the fixture module (see section 7.4.2) and the slots. Furthermore, step 4 renders a negative test, proving that the framework is not only able to correctly reconfigure a fixture, but can also recognise situations where the design parameters cannot be satisfied. Only after the second fixture module is correctly linked with the slot, the reconfiguration procedure succeeds (step 7).

Action	Expected Behaviour
1.) Start of fixture module 1	• The module initialises and extends the actuator to find the home position
2.) Start of the fixture coordinator	• The transport component is initialised and the first slot is moved to the home position • The transport component is displayed with all details in the GUI • The fixture module is automatically recognised and displayed with all details in the GUI
3.) Link Slot 1 with Fixture module 1	• The fixture module object of the coordinator is enhanced with 2 additional capabilities, namely the SenseBodyPosition and AdjustBodyPosition
4.) Click on Button "Workpiece A"	• The design parameters are retrieved and the reconfiguration procedure aborts with an error message, indicating that the

	current fixture setup cannot satisfy the design criteria. The reason for this is that there are more contact points than fixture modules.
5.) Launch Fixture module 2	• The fixture coordinator discovers the module and displays its properties on the GUI
6.) Link Slot 2 with Fixture Module 2	• Same as step 3.
7.) Click on Button "Workpiece A"	• The design parameters are retrieved and the reconfiguration procedure finishes successfully. Contact point 1 is assigned with fixture module 1 and contact point 2 is assigned with fixture module 2. In order to avoid a collision between both modules, the reconfiguration commands are executed such that module 2 is moved first to its target position.
8.) Click on the "Start Clamping"-button	• The clamping process is started and the force profiles are followed by the associated fixture modules.
9.) Click on the "Stop Clamping" button	• Both modules retract fully to their initial home position

Table 7-4: Experiment Procedure and Expected Behaviour

The test procedure above has been carried out 15 times over a period of 3 days. To obtain an initial verification of the correct positioning of the fixture modules, the target counts of the motors for the rail and the linear actuators have been calculated manually, as illustrated in Figure 7-14. Based on this, these values were compared with the final counts of both motors after the completion of the reconfiguration procedure.
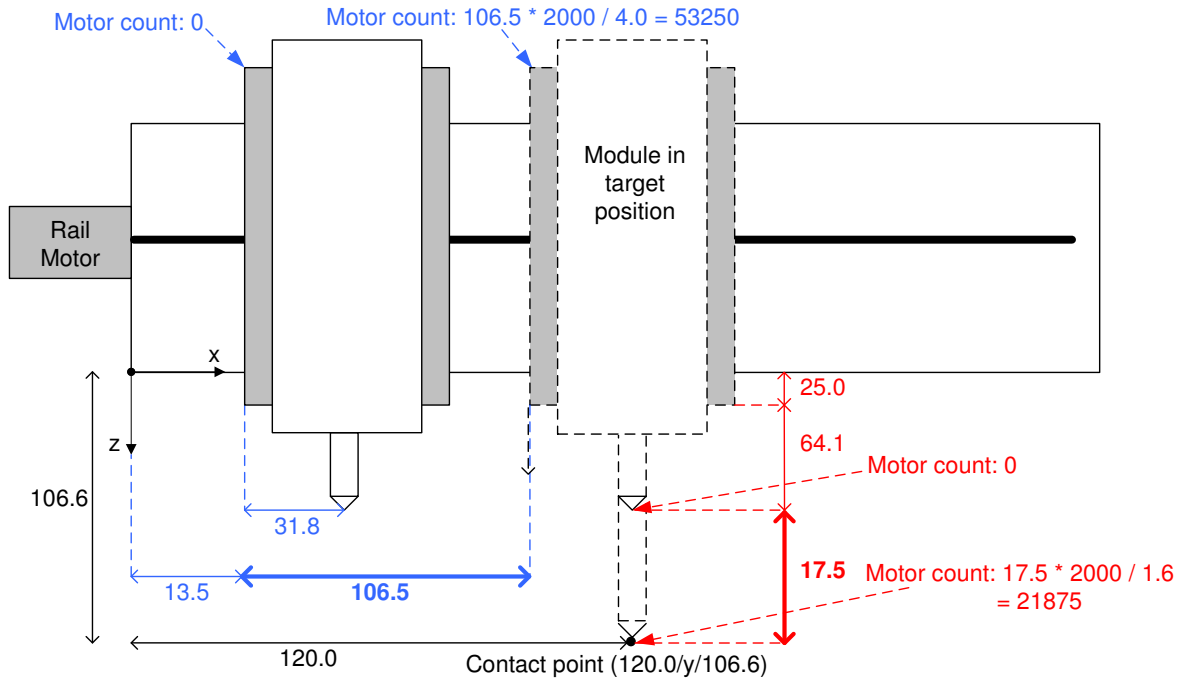
Figure 7-14: Calculating Motor Counts for the Rail Motor (blue) and the Actuator (red)

Figure 7-14 shows a top-down view of the transport component with a fixture module mounted on the carrier plate (gray). All distances are provided in mm and have been obtained using gauge blocks. The details relevant for the positioning of the module on the rail are marked in blue colour, while red is used for the calculation of the motor count for the linear actuator. When the carrier plate is in its home position, the left corner of the plate is 13.5mm away from the global coordinate system. In this position, the rail motor has its initial motor count of 0. Due to the dimension of the actuator and the way it has been mounted on the carrier plate, the carrier must be moved 106.5mm along the rail in order to ensure that the actuator tip can reach the contact point. This distance equals a motor count of 53250, as a result of equ. 7-1. The same strategy was applied to calculate the target values for the linear actuator. When the actuator is in its home position, the distance between its tip and the global coordinate system is 89.1 mm (25.0 + 64.1). Consequently, the actuator must extend by 17.5mm in order to reach the contact point. This equals a motor count of 21875, based on the pitch of the actuator (16mm), the gear factor (1:10) and the positional resolution (2000). Additionally, the locations of the contact points have been marked on the workpiece. This allows to visually inspect if the workpiece is approached correctly.

To verify if the force profiles are followed, the fixture module software has been extended with the capability to store the measured force sensor values, together with a time stamp. Additionally, the software stores the time when it receives a new target force from its subscriber. The time stamp consists of the clock count of the PC's CPU which operates at a frequency of 2999980000 ticks per second. Based on this, the elapsed time in milliseconds between two samples can be calculated, using the equation below

$$t_{elapsed} = (CLK_2 - CLK_1) * 1000 / 2999980000 \qquad \textbf{(Equ. 7-3)}$$

, where CLK_1 and CLK_2 stand for the clock counts of the first and second sample, respectively. Additionally, the fixture coordinator software retains the clock count when the "Start Clamping"-button is pressed. Since both software applications run on the same PC, the reaction time of the fixture module can be obtained using (Equ. 7-3). The reaction time is influenced by the delay for the publish/subscribe communication, the cycle time of the module software and the time delay until a motor movement results in a change of the force sensor readings. These delays are further discussed in the next section. All measurements are stored in a text file with the CSV-format which can be opened by Microsoft Excel in order to draw diagrams.

7.4.4. Test Results

During the execution of the test procedure, the expected behaviour of the fixture coordinator could be observed. The fixture modules were discovered automatically and the details of their capabilities were displayed by the GUI. Furthermore, in step 4 of the test sequence the reconfiguration process was aborted as expected with an error message, indicating that the fixture design cannot be satisfied. Once the second module was discovered and linked to its slot, the reconfiguration process was carried out successfully. In particular, the list of reconfiguration commands was reordered automatically by the fixture coordinator to avoid the detected collision between the two modules. Thus, during the execution of this list it could be observed that the command to move module 2 along the rail was carried out before module 1 was repositioned. The accurate approach of the fixture modules towards the contact points was examined for module 1 only, since the second module is moved manually on the transport component. For this, the final counts of both, the motor for the rail and the linear actuator were retrieved after the completion of the

reconfiguration sequence, using the "Measurement And Automation Explorer (MAX)" from National Instrument, Inc. This software can be used to display the details of the motion controller, including the current motor counts on all axis. As a result, it can be stated that the expected motor count values as calculated in section 7.4.3 were achieved, indicating that the contact points were accurately approached. This was confirmed by visually inspecting the workpiece, which is shown in Figure 7-15.
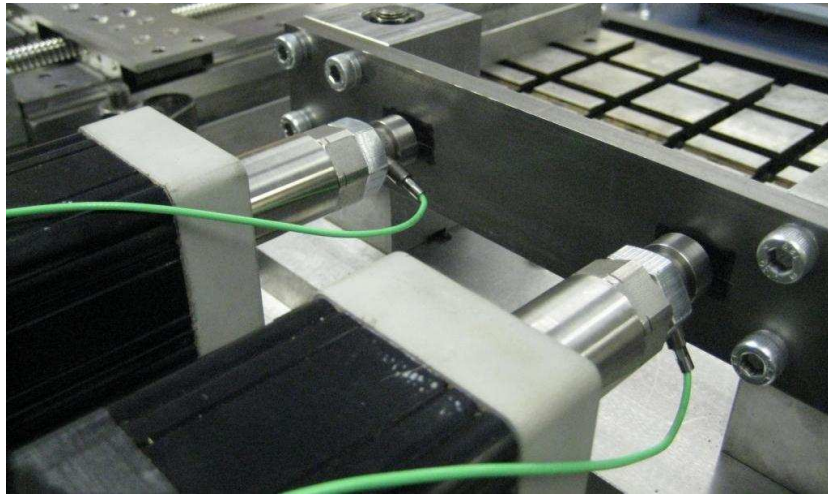
Figure 7-15: The Tip of the Linear Actuator after the Reconfiguration Sequence

The general results for the force adaptation are summarised in the diagram provided by Figure 7-16. The diagram shows the measured forces and the target forces for the first fixture module during the 6th test run. All other test runs have shown similar results and are therefore not discussed in the subsequent sections. As can be seen, the force profile is followed by the fixture module throughout the entire duration of the test. Target forces are reached within less than 300ms after they have been published by the fixture coordinator. After this time span the measured values stabilise with small fluctuations of less than 2%. The fluctuations can be explained by the noise of the sensor feedback which results in minimal motor movements when the force control algorithm tries to compensate the alleged error. Just after 10 seconds there is a clearly visible increase in the measured forces on this module which results from the effects when the second actuator decreases its own clamping force from 100 N to 75 N. However, as can be seen in the diagram, the force adaptation of the module compensates for this error and stabilises again after about 300ms.
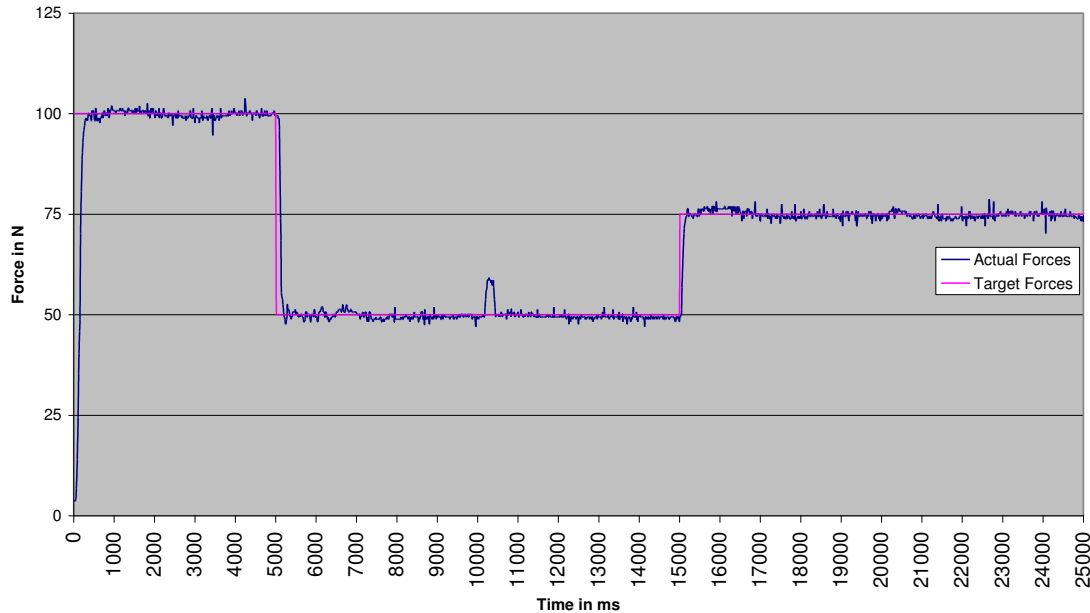
Figure 7-16: Comparison of Actual Force vs. Target Force for Fixture Module 1

Figure 7-17 shows a more detailed graph of the adaptation for the first target force by module 1. At time 0, the fixture coordinator has just published the target force of 100N. This is received 4.5ms later by the fixture module, which subsequently triggers the actuator to move in order to adapt the force. This delay is caused by the transmission time for the data transfer via DDS and the cycle time of the fixture module program. However, based on the reported performance benchmarks in chapter 3, the latency induced by DDS is significantly smaller than 1ms. Consequently, the main reason for the delay is the cycle time of the fixture module program. This is supported by the measured sample data which show an average cycle time of around 9ms. Hence, in the worst case a delay equal to a full cycle can occur when the target force is published just after the AdjustClampingForce-capability of the module has been performed. However, it is pointed out that the cycle times of the fixture modules in the experiments are negatively affected by the fact that all programs were operated on the same PC, thereby taking away processor resources from each other. Secondly, the measurement of the sample data itself takes time, typically in the range of 300-400 microseconds. Thirdly, due to the Windows operating system, a number of other processes are executed in parallel, consuming processor time. Hence, the observed delays can be drastically reduced by implementing the concept on dedicated processors for each fixture module. The average cycle time of the fixture coordinator program is lower at 3-4ms because the fixture coordinator does not interact with any hardware. However, in the

experiment it is also slowed down due to the previously mentioned reasons. Further delays are introduced as a result of the integrated GUI and the thread for the transport components. In an industrial environment, these tasks would be implemented as separate applications, thereby significantly increasing the performance of the coordinator software.
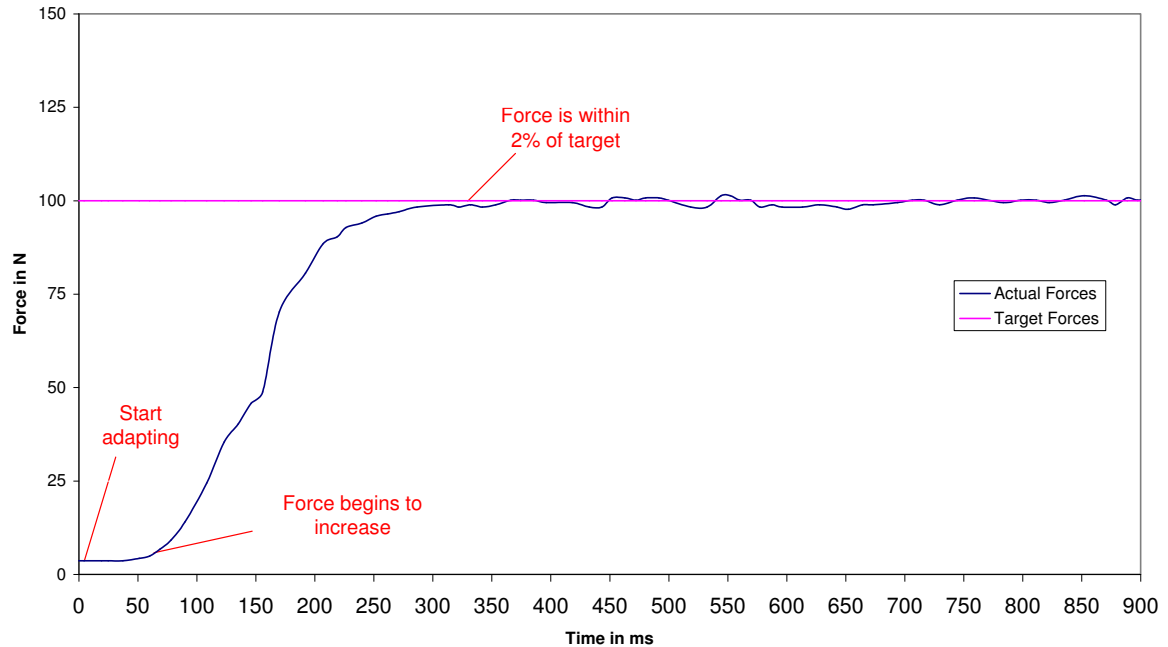
Figure 7-17: Detailed Comparison of Force Adaptation for Fixture Module 1

After the actuator starts moving, a further delay of approximately 50ms can be observed until the measured clamping force values start to increase. This delay can be explained with the inertia the motor has to overcome when it is acting against the workpiece and potential backlash effects. Therefore, this effect is caused by the motor characteristics, rather than being related with the presented concepts of this research. Moreover, it shows that the overhead as a result of the communication infrastructure is significantly smaller than the delays, induced by the equipment itself.

7.5. Testing of the Fixture Reconfiguration with two Transport Components

7.5.1. Objectives

For the second experiment, two prismatic parts with different geometries and dimensions were clamped. For this, the fixture setup was extended with the second transport

component which is positioned perpendicular to the first rail. One of the fixture modules was mounted on the carrier of the second rail, resulting in a new fixture layout consisting of two transport components with one fixture module each. Hence, the objectives of this experiment can be summarised as follows:

- To demonstrate the ability of the framework to adapt to a new fixture setup without the need for reprogramming
- To demonstrate the automatic reconfiguration of the fixture modules for different prismatic workpieces

Additionally, the test demonstrates the ability of the communication infrastructure to detect when fixture modules become disconnected.

7.5.2. Configuration Details

Figure 7-18 shows the photographs and the dimensions for both test workpieces. The material of these parts is aluminium. As can be seen, workpiece A has a dimension of 320 mm x 320 mm x 50 mm. Conversely, workpiece B is smaller with a dimension of 300mm x 300mm x 50mm. The characteristics of the cut-out can be obtained from the picture.
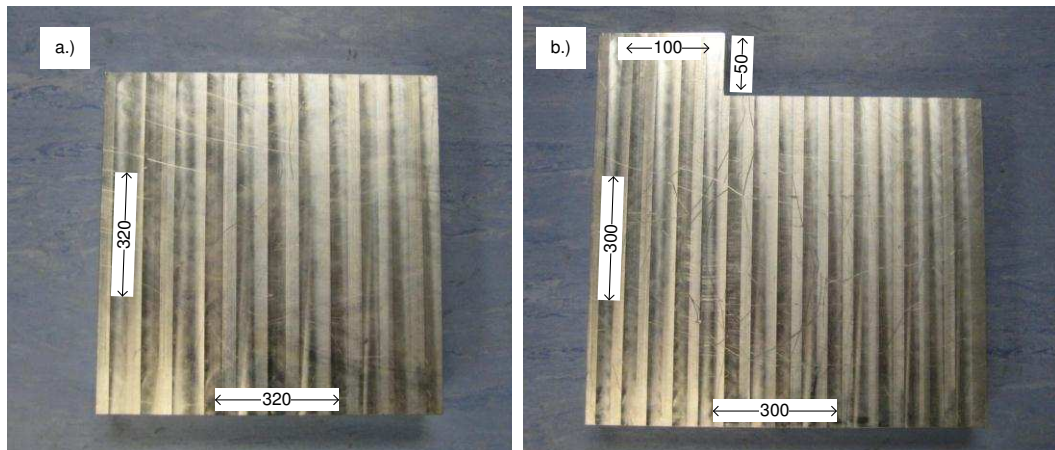
Figure 7-18: Photographs and Dimensions for (a) Workpiece A   (b) Workpiece B

The software programs for the two fixture modules are configured with the same files as in the first test, since the internal device structure of each module remained the same. The fixture coordinator is configured with the following information.

- The origin of the global coordinate system is set in the same position as in the previous test. It is positioned in the bottom corner of the first rail plate, as shown in Figure 7-19.
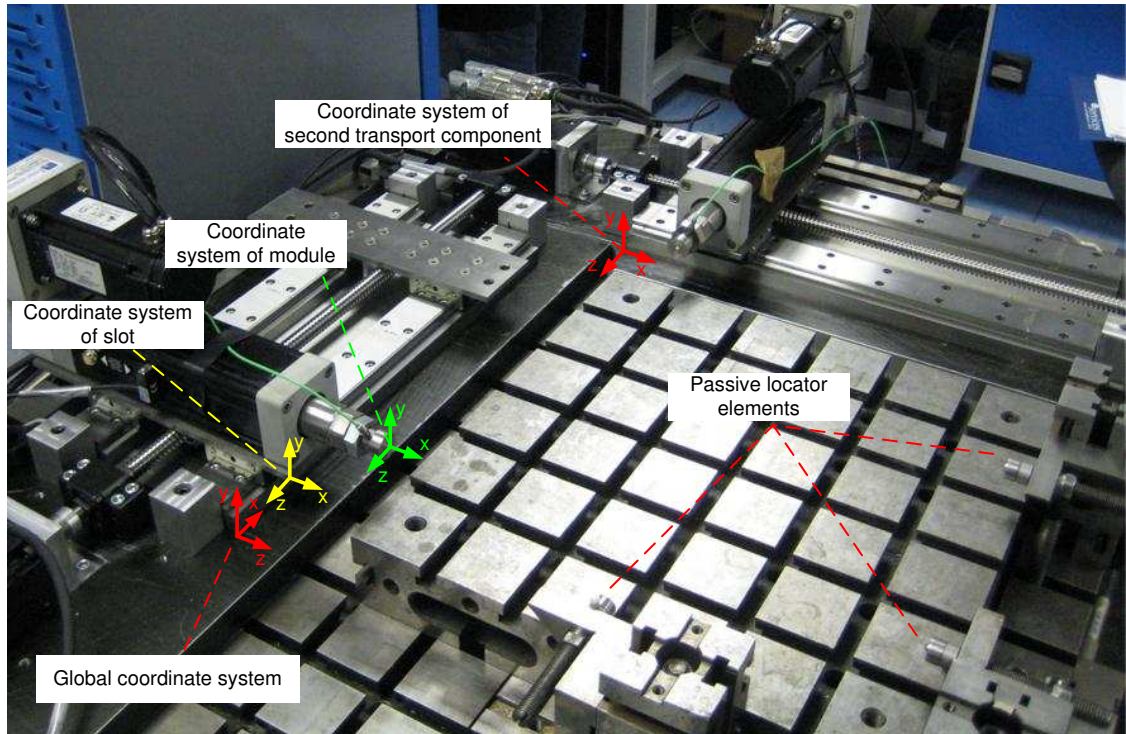
Figure 7-19: Test Setup for the Second Experiment

- Configuration details for the first transport component
  - o All details are the same as in section 7.4.2. However, the second slot has been manually moved to the far end of the rail, as can be seen in Figure 7-19. This is reflected by the spatial description for slot 2, which is set to
    - x: 300.0 mm   rot_x: 0°
    - y: 35.0 mm   rot_y: -90°
    - z: 25.0 mm   rot_z: 0°
- Configuration details for the second transport component
  - o The local coordinate system is placed in the bottom corner of the ground plate, as shown in Figure 7-19. The values for the spatial description define the relative position and orientation of the local coordinate system to the global coordinate system. These values are results of measurements with a calibrated micrometer and are summarised below,

- x: 462.4 mm   rot_x: 0°
- y: 0.0 mm   rot_y: -90°
- z: 54.0 mm   rot_z: 0°

- Configuration data for the slot
  - All configuration details are identical with those of the first slot of transport component 1.

As can be seen in Figure 7-19, there are three additional passive locator elements to confine the clamped workpiece. Since these elements are not controlled by the software framework, the fixture coordinator is not informed about their existence. Furthermore, the fixture coordinator is configured with the following fixture design details, in the form of ContactPoint objects. Figure 7-20 shows both sample workpieces when they are clamped, indicating the positions of the subsequently defined contact points.

- For workpiece A
  - ContactPoint 1
    - Spatial Description
      - x: 185.0 mm   rot_x: 0°
      - y: 68.6 mm   rot_y: -90°
      - z: 112.8 mm   rot_z: 0°
    - Role: Clamp
    - ForceProfile: constant clamping force at 200N
  - ContactPoint 2
    - Spatial Description
      - x: 344.6 mm   rot_x: 0°
      - y: 68.6 mm   rot_y: -180°
      - z: 260.0 mm   rot_z: 0°
    - Role: Clamp
    - ForceProfile: constant clamping force at 200N
- For workpiece B
  - ContactPoint 1
    - Spatial Description

- x: 170.0 mm   rot_x: 0°
- y: 68.6 mm   rot_y: -90°
- z: 113.0 mm   rot_z: 0°
  - Role: Clamp
  - ForceProfile: constant clamping force at 200N
- ContactPoint 2
  - Spatial Description
    - x: 343.7 mm   rot_x: 0°
    - y: 68.6 mm   rot_y: -180°
    - z: 350.0 mm   rot_z: 0°
  - Role: Clamp
  - ForceProfile: constant clamping force at 200N
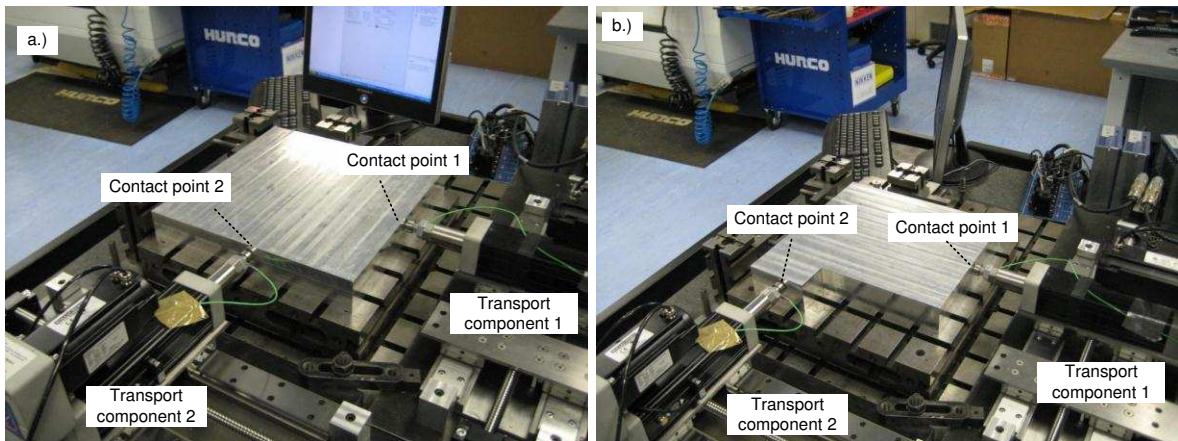
Figure 7-20: Contact Points for (a) Workpiece A and (b) Workpiece B

7.5.3. Testing Procedure

Table 7-5 shows the sequence of actions during the experiment and the expected behaviour. To link the fixture modules with the slots, the same values as in the first experiment are used, which are summarised below.

- X: 64.1 mm   rot_x: 0°
- y: 33.6 mm   rot_y: 0°
- z: -31.8 mm   rot_z: 0°

Action	Expected Behaviour
1.) Start of fixture	• Both modules initialise and extend their actuator to

module 1 and 2	find the home position
2.) Start of the fixture coordinator	• Both transport components are initialised and the slots, connected to the ball screw are moved to the home position • Both transport components are displayed with all details in the GUI • Both fixture modules are automatically recognised and displayed with all details in the GUI
3.) Kill process of fixture module 1	• The fixture coordinator software discovers the disconnected module within 1 second and immediately destroys its related software objects
4.) Restart fixture module 1	• Fixture module 1 initialises by finding its home position and publishing its capability description • The fixture coordinator software discovers the new module and displays its details in the GUI
5.) Link Slots with fixture modules	• Both fixture module objects of the coordinator are enhanced with 2 additional capabilities, namely SenseBodyPosition and AdjustBodyPosition
6.) Click on Button "Workpiece A"	• The design parameters are retrieved and the reconfiguration commands are executed. After both modules have been repositioned on the rails, the operator is asked to position the part against the passive locators. After that, both modules approach the workpiece.
7.) Click on the "Start Clamping"-button	• Both fixture modules start applying the specified constant clamping force
8.) Click on the "Stop Clamping" button	• Both modules retract fully to their respective home positions
9.) Click on Button "Workpiece B"	• Same as step 4

10.) Click on the "Start Clamping" button	• Both fixture modules start applying the specified constant clamping force
11.) Click on the "Stop Clamping" Button	• Both modules retract fully to their respective home positions

Table 7-5: Experiment Procedure and Expected Behaviour

The test procedure above has been carried out 15 times over a period of 3 days. As in the first experiment, the applied clamping forces were recorded in CSV-files. To verify the positioning of the fixture modules, the target motor counts were calculated manually, using the same approach as in the first experiment. The expected values are listed in Table 7-6 and were compared with the real values, as signalled by the feedback devices of the motors.

	Fixture Module 1	*Fixture Module 2*
Workpiece A	Rail motor 1: 69850 counts	Rail motor 2: 80352 counts
	Actuator motor 1: 29625 counts	Actuator motor 2: 35875 counts
Workpiece B	Rail motor 1: 62349 counts	Rail motor 2: 125350 counts
	Actuator motor 1: 29875 counts	Actuator motor 2: 37000 counts

Table 7-6: Predicted Motor Counts for Workpieces A and B

Additionally, the correct approach of the fixture modules towards the contact points was inspected visually.

7.5.4. Test Results

As expected, each fixture module moved its actuator to the home position during the initialisation routine. Similarly, the fixture coordinator instantiated two objects for the transport components which moved their carrier to its home position. After that, the fixture coordinator discovered both fixture modules, since they have previously published their capability descriptions. Subsequently, the details of their capabilities could be obtained from the GUI. When the process of fixture module 1 was aborted, the coordinator software reacted correctly by informing the operator with an immediate error message and deleting the software objects related to module 1. This showed the ability of the communication infrastructure to be the backbone of robust industrial fixtures with the ability of failure recovery. Linking both modules with their slots in the fixture coordinator, resulted in the enhancement of their capabilities with AdjustBodyPosition and SenseBodyPosition capability objects. After pressing the button "Workpiece A", the reconfiguration sequence

succeeded with the actuator tips of both modules approaching the workpiece. Similar to the first experiment, the achieved motor counts were compared with the pre-calculated target values, revealing that the fixture reconfiguration was performed as expected. Visual inspection of the parts showed that the actuator tips were positioned at the contact point. After the clamping of workpiece A, both modules retracted to their actuator home positions. When the button "Workpiece B" was clicked, the new design parameters were correctly retrieved by the fixture coordinator and the reconfiguration process was initiated. First, both fixture modules were repositioned on the rails, before the operator was asked to load the part and position it against the passive locator elements. After that, the fixture modules approached the part until the linear actuator reached its target position. Again, a comparison of the motor count and visual inspection showed agreement with the expected outcomes. Figure 7-21 shows a detailed picture of the fixture after the completion of the reconfiguration process for workpiece B.

Figure 7-21: Clamping of Workpiece B

With regards to the force control, the same effects as in the first experiments could be observed for all workpieces and fixture modules. Therefore, for the discussion of these results it is referred to section 7.4.4. Detailed diagrams of the force profiles are provided in Appendix D.

7.6.   Chapter Summary

The key research elements have been verified using a set of experiments. The experimental results show that the research results can be applied to automate the reconfiguration and clamping process of different fixturing systems. The results also demonstrate that the

initially defined use cases (see section 3.3) are satisfied by implementing the proposed model and the methodology.

With regard to the use case "Initialise Fixture", it was shown that the communication infrastructure is able to discover the different components in the system and represent their capabilities. Fixture modules are discovered regardless of whether or not they have been launched before the fixture coordinator. Additionally, the communication infrastructure discovers disconnected fixture modules. In the context of the use case "Reconfigure Fixture", it can be stated that the change-over from the fixture layout for the first to the second experiment required approximately 30 minutes, due to the manual labour needed to mount one module on the second transport component. However, no changes were necessary in any of the software applications that operate the fixture. This is a significant improvement over existing systems which typically require reprogramming and recompiling in order to be adapted for a new fixture. Moreover, the automatic reconfiguration for two different workpieces was accomplished in less than 10 seconds.

Finally, it was shown that the proposed software framework can be used for the realisation of adaptive fixtures. In the experiments, target forces were reached in less than 300ms. This is comparable to the reaction times of the adaptive fixture, developed at the National University of Singapore [49] which has been characterised as the state-of-the-art in adaptive fixturing. However, in addition to the force adaptation capability the concepts proposed by this research render the fixture not only adaptive, but also reconfigurable. This provides evidence that this research is indeed a promising approach towards the realisation of reconfigurable and adaptive fixturing system for complex manufacturing processes.

8. Conclusions and Future Work

8.1. Introduction

The research described in this thesis was motivated by the ongoing trend towards the utilisation of advanced computer technology and sensor feedback for the development of fixtures that are both, adaptive and reconfigurable. However, as identified in chapter 2, existing fixturing solutions satisfy at best only one of the aforementioned characteristics. The main barriers for this are (1) the lack of a data model for the representation of the capabilities of adaptive fixtures; (2) a missing fixture reconfiguration approach that is applicable to a wider range of different fixturing systems and (3) a lack of a communication infrastructure that recognises the need for flexible and platform-independent information exchange between the participating components.

According to a detailed research framework, presented in chapter 3, the knowledge gaps were first translated into clear research objectives. Additionally, a detailed use case analysis was conducted and available technologies for the realisation of the communication infrastructure were compared. Based on this, the key concepts of a software framework for the operation of reconfigurable and adaptive fixturing systems were developed and finally demonstrated in a prototype application.

This chapter provides a summary of the key knowledge contributions in section 8.2 and discusses potential application areas in industry in section 8.3. Furthermore, section 8.4 focuses on the future work that needs to be carried out in order to guarantee industrial uptake of the proposed framework.

8.2. Original Contribution to Knowledge

A new data model for the representation of the capabilities of reconfigurable and adaptive fixturing systems has been developed

In contrast to existing data models for fixture reconfiguration which appear to concentrate on the design phase and treat fixtures as purely mechanical, passive devices, the developed

data model uses object-oriented modelling techniques that are able to capture the changing capabilities of adaptive fixtures during their operation. In addition to conventional object-oriented techniques such as inheritance, the model makes heavy use of software delegation and a number of object-oriented design patterns to accomplish the dynamic access and flexible substitution of the model elements during the operation of the fixture. In this way, the research does not only contribute to the fixturing domain by the introduction of a new data model, but also to the field of computer science through the application of existing concepts to a new application area. For the formalisation and definition of the relationships between the model elements the Unified Modelling Language (UML) has been used which guarantees a platform-independent definition of the data model.

A generic methodology for the automatic reconfiguration of adaptive fixturing systems has been developed

A novel decision-making methodology for fixture reconfiguration has been developed which consists of two interrelated parts. Firstly, the capability recognition method describes how the elements of the object-oriented data model are instantiated by both, the fixture module software and the fixture coordinator, in order to reflect the capabilities of a given fixture setup. This results in a layered object hierarchy where model elements of higher layers delegate requests to the model elements of subordinate layers during the operation of the fixture. Secondly, the setup adaptation method defines the steps for the reconfiguration of an existing fixture layout to accommodate the next workpiece. The approach is based on matching the software objects representing the physical setup with the objects representing the predefined fixture design parameters. This assignment allows to delegate the generation of the reconfiguration sequence to each individual fixture module, using the Command pattern approach. As a result, the proposed methodology is independent from the number and type of the existing fixture modules and can therefore be adapted for a plethora of different setups.

A flexible communication infrastructure for the operation of reconfigurable and adaptive fixturing systems has been developed

A flexible communication infrastructure has been proposed which allows the platform-independent communication between the various parts of the fixturing system through the

adoption of a publish/subscribe mechanism. In contrast to existing approaches for adaptive fixtures which rely on hardwired connections between the devices, the proposed infrastructure allows to dynamically establish communication channels when components are added, removed or replaced. For this an emerging middleware standard (DDS) has been applied to the fixturing domain which so far lacks any standardised communication infrastructure. The required data topics and data types were defined, using the platform-independent Interface Definition Language (IDL). Additionally, standardised device library interfaces and method interfaces for the data model elements were defined which are the key for the platform-independent and flexible operation of the fixture.

In addition, the following secondary contributions have been achieved:

- A comprehensive requirement analysis of reconfigurable and adaptive fixturing system was carried out, based on a use case study.
- A detailed assessment of different middleware concepts for the use as a communication infrastructure for fixturing systems was conducted.
- A prototype software application for the operation of an exemplary fixturing system has been developed, based on the proposed core knowledge contributions.

8.3. Areas of Application

The key knowledge contributions support a common software framework which can significantly reduce the efforts for the development of adaptive and reconfigurable fixturing systems in future applications. As demonstrated by the prototype application in chapter 7, ready-to-use skeleton programs for the fixture coordinator and the fixture modules can be used and configured with information of the particular fixture setup. This will benefit system integrators as it alleviates the need for programming of the overall application architecture and recurring tasks, such as the recognition of equipment capabilities, information exchange and the realisation of the reconfiguration procedure. As a result, engineers will be able to focus on their core competencies, such as the generation of clamping strategies and the mechanical design of the fixturing system. Moreover, the research results of this study are expected to be applicable to a wide range of applications in the fixturing domain, from assembly operations to fixtures for machining operations. Apart

from the prototype application, the research outcomes have partly been implemented in a reconfigurable fixturing system for the assembly and disassembly of Rolls-Royce aero-engines. At the time of writing this thesis, large parts of the software for this system have been completed, which utilise the reconfiguration methodology, described in chapter 5. Other companies, including Airbus have shown interest in the communication infrastructure as a basis for the development of a new generation of adaptive and reconfigurable fixtures.

8.4.   Future Work

While the reported research is regarded as a significant step towards the successful realisation of reconfigurable and adaptive fixtures, it also opens new avenues for further research. The main areas where these opportunities arise are summarised below.

Extension of the data model for other fixturing scenarios and equipment

For the definition of the data model only the most common equipment types and associated capabilities have been included, in order to reduce the complexity of the proposed model. However, the described object-oriented structure can be extended with additional classes and attributes to represent more equipment, like other clamping types, sensor devices or locator types, as well as their associated capabilities. An example for an additional equipment type would be a temperature sensor. The latter could be integrated by defining new classes for the sensor device itself and its associated capability. Additionally, a new data topic for the communication of the measured temperature data would have to be defined. Similarly, the existing classes of the model can be extended with further attributes in order to arrive at a more detailed representation of the fixture. For example, the device classes could be extended with attributes for the weight, rigidity and material.

Investigation of distributed, collaborative fixturing approach

The proposed publish/subscribe communication can easily be extended by further data topics in the future and it facilitates the integration of the fixture with other subsystems of the shop floor. For example, a Human Machine Interface (HMI) can easily participate in the data exchange by registering publishers or subscribers for the appropriate data topics.

Additionally, the proposed communication infrastructure can act as a starting point for further research towards the development of a distributed system with autonomous fixture modules. In such an approach, the fixture coordinator would become obsolete as a central instance to ensure correct functioning. Instead, the modules would subscribe to all topics and hence get informed about each other's existence and current states. Based on this information, a collaborative methodology for the fixture reconfiguration and clamping procedure could be developed to adapt the system without the need of a central coordinator. The advantage of this approach is the elimination of the fixture coordinator as a single-point-of-failure.

Extension of the framework for repositionable transport components

The proposed software framework is based on the assumption that only the fixture modules can change positions during the operation of the fixture while the transport components are fixed. While this limitation reflects the physical characteristics of many existing fixturing systems, there are scenarios conceivable where transport components can be repositioned automatically, too. For example, consider a setup where the rails from the prototype described in chapter 7 are mounted on a stage that can lift the rails up and down. For such cases, the framework needs to be extended with the option to link transport components with each other. Additionally, new capability classes for the transport components need to be created which represent their ability to be repositioned. Finally, the generation of the reconfiguration commands needs to be extended by command classes for the repositioning of transport components and further strategies are required to determine whether or not the repositioning of transport components is required in order to align the fixture modules with the contact points.

Extension of the fixture reconfiguration algorithm with the capability to make proposals

Furthermore, it is possible to extend the fixture reconfiguration methodology with the ability to actively propose changes in the event that an existing fixture layout cannot be transformed according to given design requirements. A possible solution would involve a data base which contains information about the available fixture modules and transport

components. Based on this, the system could search the data base for replacement components which better satisfy the requirements of a given fixture design. Similarly, it could propose the addition of more fixture modules and other changes of the fixture layout in order to accommodate the next workpiece.

8.5.   Concluding Remarks

Fixtures play an important role in both assembly and machining operations. Their significance is reflected by the myriad of research activities aimed at improving various aspects of their behaviour. However, despite recent efforts towards increased reconfigurability and adaptability, fixtures still appear to be major bottlenecks of reconfigurable manufacturing systems.

The research presented in this thesis rooted in the observation that a major obstacle for the successful realisation of fixturing systems with reconfigurable and adaptive capabilities, is the lack of flexible software concepts for the operation of such devices. The overarching idea was to define the core concepts for a software framework that minimises the need for application programming when a new fixture is developed or an existing system is adapted for new requirements. Starting from an extensive literature review and a detailed requirement analysis, the core knowledge contributions of the research have been developed and presented in the chapters 4, 5 and 6. These are (1) an object-oriented data model; (2) a generic fixture reconfiguration methodology and (3) a publish/subscribe communication infrastructure. While the developed framework is not claimed to be a complete industrial solution, it presents a significant step towards the successful and cost-effective development of reconfigurable fixturing systems in future applications. The fundamental principle of the developed framework has been demonstrated in a prototype application in chapter 7 while parts of the data model and reconfiguration methodology have been implemented for an industrial testcase, as described in section 8.3.

As the work has been done in close collaboration with industry, there is a good chance that the research outcomes will be accepted and adopted as a platform for the development of next-generation fixtures. However the success of the work also depends on the

dissemination of results to the wider manufacturing community and the inclusion of extra features to create a commercial product that system integrators can use for industrial projects.

References

1.   Bi, Z.M. and Zhang, W.J., 2001, "Flexible fixture design and automation: Review, issues and future directions", International Journal of Production Research, vol. 39, n. 13, pp. 2867-2894.

2.   Consalter, L.A. and Boehls, L., 2004, "An Approach to Fixture Systems Management in Machining Processes", Journal of the Brazilian Society of Mechanical Science & Engineers, vol. 26, n. 2, pp. 145-152.

3.   Perremans, P., 1996, "Feature-based description of modular fixturing elements: The key to an expert system for the automatic design of the physical fixture", Advances in Engineering Software, vol. 25, n. 1, pp. 19-27.

4.   Bi, Z.M., Lang, Y.T.S., Verner, M. and Orban, P., 2007, "Development of reconfigurable machines", International Journal of Advanced Manufacturing Technology, vol. 39, n. 11-12, pp. 1227-1251.

5.   Mohamed, Z.M., 1996, "A flexible approach to (re)configure Flexible Manufacturing Cells", European Journal of Operational Research, vol. 95, n. 3, pp. 566-576.

6.   Oxford University Press, 2010, "Oxford English Dictionary", Available from: www.oed.com, April 2010.

7.   Riehle, D., 2000, "Framework design: A Role model approach", PhD Thesis, ETH Zurich, pp. 230.

8.   Fayad, M. and Schmidt, D.C., 1997, "Object-oriented application frameworks", Communications of the ACM, vol. 40, n. 10, pp. 32-38.

9.   Shirinadeh, B., Lin, G. and Chan, K., 1995, "Strategies for planning and implementation of flexible fixturing systems in a computer-integrated manufacturing environment", Proceedings of International Conference on Computer Integrated Manufacturing, Singapore.

10.   Lin, G. and Du, H., "Design and development of an automated flexible fixture", Proceedings of the 4th Internatonal Conference on Automation Technology (AUTOMATION '96), Hsinchu, Taiwan.

11.   Hoffman, G.H., 1987, "Modular fixturing", Manufacturing Technology Press, Lake Geneva, Wisconsin, ISBN: 978-0932819000, pp. 186.

12.   Gandhi, M.V. and Thompson, B., 1986, "Automated design of modular fixturing for flexible manufacturing systems", Journal of Manufacturing Systems, vol. 5, n. 4, pp. 243-252.

13.   Lewis, G., 1983, "Modular fixturing system", Second International conference on Flexible Manufacturing Systems (IFS), London.

14.   Lin, C.I., 1994, "A systematic conceptual design of modular fixtures", International Journal of Advanced Manufacturing Technology, vol. 9, n. 4, pp. 217-224.

15.   Ngoi, B.K.A., 1990, "Computer aided design of modular fixture assembly", PhD Thesis, University of Canterbury, New Zealand, pp. 225.

16.   Sela, M.N., Gaundry, O., Dombre, E. and Benhabib, B., 1997, "A reconfigurable modular fixturing system for thin-walled flexible objects", International Journal of Advanced Manufacturing Technology, vol. 13, n. 9, pp. 611-617.

17. Zheng, Y. and Qian, W.-H., 2008, "A 3-D modular fixture with enhanced localization accuracy and immobilarization capability", International Journal of Machine Tools and Manufacture, vol. 48, n. 6, pp. 677-687.

18. Xu, Y.-C., Liu, G., Tang, Y., Zhang, R., Dong, R. and Wu, M., 1985, "A modular fixturing system for flexible manufacturing", in "Flexible Manufacturing Systems", IFS Publications, Bedford, pp. 227-233.

19. Kusiak, A., 1992, "Intelligent design and manufacturing", John Wiley & Sons, New York, ISBN: 978-0471534730, pp. 776.

20. Shirinzadeh, B., 1995, "Flexible and automated workholding systems", Industrial Robot: An International Journal, vol. 22, n. 2, pp. 29-34.

21. Nee, A.Y.C., Whybrew, K. and Kumar, A.S., 1995, "Advanced Fixture Design For FMS", Advanced Manufacturing Series, Springer-Verlag, London, ISBN: 978-1848827387, pp. 481.

22. Hazen, F.B. and Wright, P.K., 1990, "Workholding automation: innovations in analysis, design and planning", Manufacturing Review, vol. 3, n. 4, pp. 224-236.

23. Aoyama, T., 2004, "Development of Gel Structured Electrorheological Fluids and their Application for the Precision Clamping Mechanism of Aerostatic Sliders", CIRP Annals - Manufacturing Technology, vol. 53, n. 1, pp. 325-328.

24. Rong, Y., Tao, R. and Tang, X., 2000, "Flexible Fixturing with phase-change materials. Part 1. Experimental study on magnetorheological fluids", International Journal of Advanced Manufacturing Technology, vol. 16, n. 11, pp. 822-829.

25. Aoyama, T. and Kakinuma, Y., 2005, "Development of Fixture Devices for Thin and Compliant Workpieces", CIRP Annals - Manufacturing Technology, vol. 54, n. 1, pp. 325-328.

26. Ahn, S.A. and Wright, P.K., 2002, "Reference free part encapsulation (RFPE): An investigation of material properties and the role of RFPE in a taxonomy of fixturing systems", Journal of Manufacturing Systems, vol. 21, n. 2, pp. 101-110.

27. Choi, D.S., Lee, S.H., Shin, B.S., Wang, K.H., Yoon, K.K. and Sarma, S.E., 2001, "A new rapid prototyping system using universal automated fixturing with feature-based CAD/CAM", Journal of Materials Processing Technology, vol. 113, n. 1-3, pp. 285-290.

28. Lee, E. and Sarma, S.E., 2007, "Reference free part encapsulation: Materials, machines and methods", Journal of Manufacturing Systems, vol. 26, n. 1, pp. 22-36.

29. Phuah, H.L., 2005, "Part-fixture behaviour prediction methodology for fixture design verification", PhD Thesis, University of Nottingham, pp. 210.

30. Hurtado, J.F. and Melkote, S.N., 1998, "A Model for the Prediction of Reaction Forces in a 3-2-1 Machining fixture", 26th North American Manufacturing Research Conference NAMRC XXVI, Atlanta, Georgia.

31. Englert, P.J. and Wright, P.K., 1986, "Application of artificial intelligence and the design of fixtures for automated manufacturing", IEEE International Conference on robotics and automation, San Francisco.

32. Cutkovsky, M.R., Kurawa, E. and Wright, P.K., 1985, "Programmable conformable clamps", AUTOFACT 4, Dearborn, Michigan.

33. Al-Habaibeh, A., Gindy, N. and Parkin, R.M., 2003, "Experimental Design and Investigation of a pin-type reconfigurable clamping system for manufacturing

aerospace components ", Journal of Engineering and Manufacture - Proceedings of the Institution for Mechanical Engineers Part B, vol. 217, n. 12, pp. 1771-1777.

34. Tuffentsammer, K., 1981, "Automatic loading of machining system and automatic clamping of workpieces", Annals of the CIRP, vol. 30, n. 2, pp. 553-558.

35. Lin, G.C.I. and Du, H., 1996, "Design and development of an automated flexible fixture", 4th International Conference on Automation Technology, Hsinchu, Taiwan.

36. Du, H. and Lin, G.C.I., 1998, "Development of an automated flexible fixture for planar objects", Robotics and Computer-Integrated Manufacturing, vol. 14, n. 3, pp. 173-183.

37. Youcef-Toumi, K. and Buitrago, J.H., 1989, "Design and implementation of robot-operated adaptable and modular fixtures", Robotics & Computer-Integrated Manufacturing, vol. 5, n. 4, pp. 343-356.

38. Benhabib, B., Chan, K.C. and Dai, M., 1991, "A modular programmable fixturing system", Journal of Engineering for Industry, vol. 113, n. 1, pp. 93-100.

39. Chan, K., Benhabib, B. and Dai, M., 1990, "A reconfigurable fixturing system for robotic assemble", Journal of Manufacturing Systems, vol. 9, n. 3, pp. 206-221.

40. Kurz, K., Craig, K. and Wolf, B., 1993, "Design and development of a flexible, automated fixturing device for manufacturing", Proceedings of 1993 ASME Winter Annual Meeting, New Orleans.

41. Kurz, K., Craig, K., Wolf, B. and Stolfi, F., 1994, "Developing a flexible automated fixturing device", Mechanical Engineering, vol. 116, n. 7, pp. 59-63.

42. Lu, S.-S., Chu, J.-L. and Jang, H.-C., 1997, "Development of a novel coordinate transposing fixture system", International Journal of Advanced Manufacturing Technology, vol. 13, n. 5, pp. 350-358.

43. Chan, K. and Lin, C., 1996, "Development of a computer numerical control [CNC] modular fixture machine design of a standard multifinger module", International Journal of Advanced Manufacturing Technology, vol. 11, n. 1, pp. 18-26.

44. Tao, Z.J., Kumar, A.S. and Nee, A.Y.C., 1999, "Automatic generation of dynamic clamping forces for machining fixtures", International Journal of Production Research, vol. 37, n. 12, pp. 2755-2776.

45. Tao, Z.J., Kumar, A.S., Nee, A.Y.C. and Mannan, M.A., 1997, "Modelling and experimental investigation of a sensor-integrated workpiece-fixture system", International Journal of Computer Applications in Technology, vol. 10, n. 3-4, pp. 236-250.

46. Gupta, S., Bagchi, A. and Lewis, R., 1988, "Sensor-based fixturing system", in "Recent Developments in Production Research", Elsevier, Amsterdam, pp. 11-16.

47. Nee, A.Y.C., Kumar, A.S. and Tao, Z.J., 2000, "An Intelligent Fixture with a Dynamic Clamping Scheme", Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture, vol. 214, n. 3, pp. 183-196.

48. Nee, A.Y.C., Tao, Z.J. and Kumar, A.S., 2004, "An Advanced Treatise on Fixture Design and Planning", Series on Manufacturing Systems and Technology, World Scientific Publishing, Singapore, ISBN: 978-9812560599, pp. 248.

49. Mannan, M.A. and Sollie, J.P., 1997, "A Force-Controlled Clamping Element for Intelligent Fixturing", Annals of the CIRP, vol. 46, n. 1, pp. 256-268.

50. Wang, Y.F., Wong, Y.S. and Fuh, J.Y.H., 1999, "Off-line modelling and planning of optimal clamping forces for an intelligent fixturing system", International Journal of Machine Tools and Manufacture, vol. 39, n. 2, pp. 253-271.

51. Wang, Y.F., Fuh, J.Y.H. and Wong, Y.S., 1997, "A model-based online Control of optimal fixturing process", IEEE International Conference on Robotics and Automation, Albuquerque, USA.

52. Rashid, A. and Mihai Nicolescu, C., 2006, "Active vibration control in palletised workholding system for milling", International Journal of Machine Tools and Manufacture, vol. 46, n. 12-13, pp. 1626-1636.

53. Nnaji, B.O. and Lyu, P., 1990, "Rules for an expert fixturing system on a CAD screen using flexible fixtures ", Journal of Intelligent Manufacturing, vol. 1, n. 1, pp. 31-48.

54. Gaoliang, P., Xu, H., Haiquan, Y., Xin, H. and Alipour, K., 2008, "Precise manipulation approach to facilitate interactive modular fixture assembly design in a virtual environment ", Assembly Automation, vol. 28, n. 3, pp. 216-224.

55. Sun, S.H. and Chen, J.L., 1996, "A Fixture Design System using Case-based Reasoning", Engineering Applications of Artificial Intelligence, vol. 9, n. 5, pp. 533-540.

56. Li, W., Li, P. and Rong, Y., 2002, "Case-based agile fixture design", Journal of Materials Processing Technology, vol. 128, n. 1-3, pp. 7-18.

57. Wang, H. and Rong, Y.K., 2008, "Case based reasoning method for computer aided welding fixture design", Computer-Aided Design, vol. 40, n. 12, pp. 1121-1132.

58. Trappey, A.C. and Matrubhutam, S., 1993, "Fixture configuration using projective geometry", Journal of Manufacturing Systems, vol. 12, n. 6, pp. 486-495.

59. Kang, Y., Rong, Y. and Yang, J.A., 2003, "Geometric and Kinetic Model Based Computer-Aided Fixture Design Verification", Journal of Computing and Information Science in Engineering, vol. 3, n. 3, pp. 187-200.

60. Kang, Y., Rong, Y. and Yang, J.C., 2003, "Computer-Aided Fixture Design Verification. Part 1. The Framework and Modelling", International Journal of Advanced Manufacturing Technology, vol. 21, n. 10-11, pp. 827–835.

61. Wu, Y., Rong, Y., Ma, W. and LeClair, S.R., 1998, "Automated modular fixture planning: Accuracy, clamping, and accessibility analyses", Robotics and Computer-Integrated Manufacturing, vol. 14, n. 1, pp. 17-26.

62. Wu, Y., Rong, Y., Ma, W. and LeClair, S.R., 1998, "Automated modular fixture planning: Geometric analysis", Robotics and Computer-Integrated Manufacturing, vol. 14, n. 1, pp. 1-15.

63. King, L.S.B. and Hutter, I., 1993, "Theoretical Approach for Generating Optimal Fixturing Locations for Prismatic Workparts in Automated Assembly", Journal of Manufacturing Systems, vol. 12, n. 5, pp. 409-416.

64. Menassa, R.J. and DeVries, W.R., 1991, "Optimisation Methods Applied to Selecting Support Positions in Fixture Design", Journal of Engineering for Industry, Transactions of ASME, vol. 113, n. 1, pp. 412-418.

65. Wu, N.H. and Chan, K.C., 1996, "A Genetic Algorithm Based Approach to Optimal Fixture Configuration", Computers and Industrial Engineering, vol. 31, n. 3-4, pp. 919-924.

66.   Krishnakumar, K. and Melkote, S.N., 2000, "Machining fixture layout optimization using the genetic algorithm", International Journalof Machine Tools and Manufacture, vol. 40, n. 4, pp. 579-598.

67.   Krishnakumar, K., Satyanarayana, S. and Melkote, S.N., 2002, "Iterative fixture layout and clamping force optimization using the genetic algorithm", Journal of Manufacturing Science and Engineering, vol. 124, n. 1, pp. 119-126.

68.   Vallapuzha, S., DeMetere, C., Choudhuri, S. and Khetan, R.P., 2002, "An investigation into the use of spatial coordinates for the genetic algorithm based solution of the fixture layout optimization problem", International Journal of Machine Tools and Manufacture, vol. 42, n. 2, pp. 265 – 275.

69.   Kaya, N., 2006, "Machining fixture locating and clamping position optimization using genetic algorithms", Computers in Industry, vol. 57, n. 2, pp. 112-120.

70.   Aoyama, T., Kakinuma, Y. and Inasaki, I., 2006, "Optimization of fixture layout by means of the genetic algorithm", I*PROMS - Intelligent Production Machines and Systems, Virtual International Conference.

71.   Huang, B., Gou, H., Liu, W., Li, Y. and Xie, M., 2002, "A framework for virtual enterprise control with the holonic manufacturing paradigm", Computers in Industry, vol. 49, n. 3, pp. 299-310.

72.   Van Leeuwen, E.H. and Norrie, D., 1997, "Holons and holarchies", Manufacturing Engineer, vol. 76, n. 2, pp. 86-88.

73.   Valckenaers, P., Van Brussel, H., Bongaerts, L. and Wyns, J., 1997, "Holonic manufacturing systems", Integrated Computer-Aided Engineering, vol. 4, n. 3, pp. 191-201.

74.   Sugi, M. and Maeda, Y., 2003, "A Holonic architecture for easy reconfiguration of robotic assembly systems", IEEE Transactions on Robotics and Automation, vol. 19, n. 3, pp. 457-464.

75.   Leitao, P. and Restivo, F., 2006, "ADACOR: A holonic architecture for agile and adaptive manufacturing control", Computers in Industry, vol. 57, n. 2, pp. 121-130.

76.   Leitão, P. and Restivo, F., 2008, "A holonic approach to dynamic manufacturing scheduling", Robotics and Computer-Integrated Manufacturing, vol. 24, n. 5, pp. 625-634.

77.   Babiceanu, R.F., Chen, F.F. and Sturges, R.H., 2005, "Real-time holonic scheduling of material handling operations in a dynamic manufacturing environment", Robotics and Computer-Integrated Manufacturing, vol. 21, n. 4-5, pp. 328-337.

78.   Gou, L., Luh, P.B. and Kyoya, Y., 1998, "Holonic manufacturing scheduling: architecture, cooperation mechanism, and implementation", Computers in Industry, vol. 37, n. 3, pp. 213-231.

79.   Jarvis, J., Ronnquist, R., McFarlane, D. and Jain, L., 2006, "A team-based holonic approach to robotic assembly cell control", Journal of Network and Computer Applications, vol. 29, n. 2-3, pp. 160-176.

80.   Ferber, J., 1999, "Multi-agent Systems: An Introduction to Distributed Artificial Intelligence", ADDISON-WESLEY, London, ISBN: 978-0201360486, pp. 528.

81.   Tang, H.P. and Wong, T.N., 2005, "Reactive multi-agent system for assembly cell control", Robotics and Computer-Integrated Manufacturing, vol. 21, n. 2, pp. 87-98.

82. Nee, A.Y.C., Kurnar, A.S., Prombanpong, S. and Puah, K.Y., 1992, "A Feature-Based Classification Scheme for Fixtures", CIRP Annals - Manufacturing Technology, vol. 41, n. 1, pp. 189-192.

83. Shirinzadeh, B., 1996, "A CAD-Based hierarchical approach to interference detection among fixture modules in a reconfigurable fixturing system", Robotics & Computer-Integrated Manufacturing, vol. 12, n. 1, pp. 44-53.

84. Jeng, Y.C. and Gill, K.F., 1997, "A CAD-based approach to the design of fixtures for prismatic parts ", Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture, vol. 211, n. 7, pp. 523-538.

85. Subrahmanyam, S.R., 2002, "Fixturing features selection in feature-based systems", Computers in Industry, vol. 48, n. 2, pp. 99-108.

86. Subrahmanyam, S.R., 2002, "A method for generation of machining and fixturing features from design features", Computers in Industry, vol. 47, n. 3, pp. 269-287.

87. Liquing, F. and Kumar, A.S., 2005, "XML-based Representation in a CBR System for Fixture Design", Computer-Aided Design & Applications, vol. 2, n. 1-4, pp. 339-348.

88. Mervyn, F., Kumar, A.S., Bok, S.H. and Nee, A.Y.C., 2003, "Development of an Internet-enabled Interactive Fixture Design System", Computer-Aided Design, vol. 35, n. 10, pp. 945-957.

89. Hunter, A., R., Ríos, C., J., Pérez García, J.M. and Vizán Idoipe, A., 2010, "Fixture knowledge model development and implementation based on a functional design approach", Robotics and Computer-Integrated Manufacturing, vol. 26, n. 1, pp. 56-66.

90. Hunter, R., Rios, J., Perez, J.M. and Vizan, A., 2006, "A functional approach for the formalization of the fixture design process", International Journal of Machine Tools and Manufacture, vol. 46, n. 6, pp. 683-697.

91. Zha, X.F., Du, H. and Lim, Y.E., 2001, "Knowledge intensive Petri net framework for concurrent intelligent design of automatic assembly systems", Robotics and Computer-Integrated Manufacturing, vol. 17, n. 5, pp. 379-398.

92. Lohse, N., Ratchev, S. and Chrisp, A., 2004, "Function-behaviour-structure model for modular assembly equipment", Proceedings of the International Precision Assembly Seminar IPAS 2004, Bad Hofgastein, Austria.

93. Lohse, N., 2006, "Towards an ontology framework for the integrated design of modular assembly systems", PhD Thesis, University of Nottingham, pp. 234.

94. Meljer, B.R., Tomlyama, T., van der Hoist, B.H.A. and van der Werff, K., 2003, "Knowledge Structuring for Function Design", CIRP Annals - Manufacturing Technology, vol. 52, n. 1, pp. 89-92.

95. Zhang, M., Fisher, W., Webb, P. and Tarn, T.-J., 2003, "Functional Model Based Object-Oriented Development Framework for Mechatronic Systems", IEEE International Conference on Robotics & Automation, Taipei, Taiwan.

96. Prabhakar, S. and Goel, A.K., 1998, "Functional modeling for enabling adaptive design of devices for new environments", Artificial Intelligence in Engineering, vol. 12, n. 4, pp. 417-444.

97. Kovács, G.L., Kopácsi, S., Nacsa, J., Haidegger, G. and Groumpos, P., 1999, "Application of software reuse and object-oriented methodologies for the modelling

and control of manufacturing systems", Computers in Industry, vol. 39, n. 3, pp. 177–89.

98. Schäfer, C. and López, O., 2004, "An Object-Oriented Robot Model and Its Integration into Flexible Manufacturing Systems ", in "Multiple Approaches to Intelligent Systems", Springer, Berlin / Heidelberg, pp. 820-829.

99. Bruccoleri, M., Pasek, Z.J. and Koren, Y., 2006, "Operation management in reconfigurable manufacturing systems: Reconfiguration for error handling", International Journal of Production Economics, vol. 100, n. 1, pp. 87-100.

100. Bruccoleri, M., 2007, "Reconfigurable control of robotized manufacturing cells", Robotics and Computer-Integrated Manufacturing, vol. 23, n. 1, pp. 94-106.

101. Alexander, C., 1979, "The Timeless Way of Building", Oxford University Press, New York, ISBN: 978-0201360486, pp. 552.

102. Coad, P., 1992, "Object-Oriented Patterns", Communications of the ACM, vol. 35, n. 9, pp. 152-159.

103. Gamma, E., Helm, R., Johnson, R.E. and Vlissides, J., 1993, "Design Patterns: Abstraction and Reuse of Object-Oriented Design", in "Lecture Notes in Computer Science ", Springer-Verlag, Kaiserslautern, pp. 406-431.

104. Gamma, E., Helm, R., Johnson, R.E. and Vlissides, J., 1995, "Design Patterns. Elements of Reusable Object-Oriented Software." Addison-Wesley Longman, Amsterdam, ISBN: 978-0582844421, pp. 395.

105. Thiry, L., Perronne, J.-M. and Thirion, B., 2004, "Patterns for behavior modeling and integration", Computers in Industry, vol. 55, n. 3, pp. 225-237.

106. Soundararajan, K. and Brennan, R.W., 2005, "A proxy design pattern to support real-time distributed control system benchmarking", in "Holonic and Multi-Agent Systems for Manufacturing", Springer, Berlin/Heidelberg, pp. 133-143.

107. Soundararajan, K. and Brennan, R.W., 2008, "Design patterns for real-time distributed control system benchmarking", Robotics and Computer-Integrated Manufacturing, vol. 24, n. 5, pp. 606-615.

108. Pont, M.J. and Banner, M.P., 2004, "Designing embedded systems using patterns: a case study", Journal of Systems and Software, vol. 74, n. 3, pp. 201-213.

109. Sanz, R. and Zalewski, J., 2003, "Pattern-based control systems engineering ", IEEE Control Systems Magazine, vol. 23, n. 3, pp. 43-60.

110. Buschmann, R., Meunier, H., Rohnert, P. and Sommerland, M., 1996, "Pattern-oriented Software Architecture - A System of Patterns", John Wiley & Sons, Chichester, ISBN: 978-9971514211, pp. 476.

111. Neumann, P., 2007, "Communication in industrial automation—What is going on?" Control Engineering Practice, vol. 15, n. 11, pp. 1332-1347.

112. Hurwitz, J., 1998, "Sorting out middleware", DBMS Archive, vol. 11, n. 1, pp. 10-12.

113. Amoretti, M. and Reggiani, M., 2009, "Architectural paradigms for robotics applications", Advanced Engineering Informatics, vol. 24, n. 1, pp. 4-13.

114. Object Management Group, 2004, "Common Object Request Broker Architecture: Core Specification, Version 3.0.3", Available from: www.omg.org, August 2007.

115. Object Management Group, 2002, "OMG IDL Syntax and Semantics", Available from: www.omg.org, October 2008.

116. Object Management Group, 2005, "Real-Time CORBA Specification, Version 1.2", Available from: www.omg.org, March 2010.
117. Object Management Group, 2007, "Data Distribution Service for Real-Time Systems, Version 1.2", Available from: www.omg.org, June 2007.
118. Shin, J., Park, S., Ju, C. and Cho, H., 2003, "CORBA-based integration framework for distributed shop floor controlsmall star", Computers & Industrial Engineering, vol. 45, n. 3, pp. 457-474.
119. Sanz, R., 2003, "A CORBA-based architecture for strategic process control", Annual Reviews in Control, vol. 27, n. 1, pp. 15-22.
120. Haber, R.E., Cantillo, K. and Jiménez, J.E., 2005, "Networked sensing for high-speed machining processes based on CORBA", Sensors and Actuators A: Physical, vol. 119, n. 2, pp. 418-426.
121. Joshi, J., 2007, "Data-Oriented Architecture", Real-Time Innovations, Inc., Whitepaper, Available from: www.rti.com, March 2010.
122. Object Computing, Inc., "OPEN DDS", Available from: http://www.opendds.org/, March 2010.
123. Real-Time Innovations, Inc., 2009, "Applications of the RTI Data Distribution Service", Available from: http://www.rti.com/industries/, 20th October 2009.
124. Object Management Group, 2009, "The Real-time Publish-Subscribe Wire Protocol DDS Interoperability Wire Protocol Specification, Version 2.1", Available from: http://www.omg.org/spec/DDSI/2.1, October 2009.
125. Veiga, G., Pires, J.N. and Nilsson, K., 2009, "Experiments with service-oriented architectures for industrial robotic cells programming", Robotics and Computer-Integrated Manufacturing, vol. 25, n. 4-5, pp. 746-755.
126. Ahn, S.C., Kim, J.H., Lim, K., Kwon, Y. and Kim, H., 2005, "UPnP approach for robot middleware", Proceedings of the 2005 IEEE International Conference on Robotics and Automation, Barcelona, Spain.
127. Estrem, W.A., 2003, "An evaluation framework for deploying Web Services in the next generation manufacturing enterprise", Robotics and Computer-Integrated Manufacturing, vol. 19, n. 6, pp. 509-519.
128. Ha, Y.-G., Sohn, J.-C., Cho, Y.-J. and Yoon, H., 2007, "A robotic service framework supporting automated integration of ubiquitous sensors and devices", Information Sciences, vol. 177, n. 3, pp. 657-679.
129. Sun Microsystems, Inc, 2002, "Java Message Service Specification, Version 1.1", Available from: www.sun.com, August 2007.
130. Sánchez, E., Portas, A., Pereira, A. and Vega, J., 2006, "Applying a message oriented middleware architecture to the TJ-II remote participation system", Fusion Engineering and Design, vol. 81, n. 15-17, pp. 2063-2067.
131. Sachs, K., Kounev, S., Bacon, J. and Buchmann, A., 2009, "Performance evaluation of message-oriented middleware using the SPECjms2007 benchmark", Performance Evaluation, vol. 66, n. 8, pp. 410-434.
132. Urdaneta, G., Colmenares, J.A., Queipo, N.V., Arapé, N., Arévalo, C., Ruz, M., Corzo, H. and Romero, A., 2007, "A reference software architecture for the development of industrial automation high-level applications in the petroleum industry", Computers in Industry, vol. 58, n. 1, pp. 35-45.

133. Mervyn, F., Kumar, A.S., Bok, S.H. and Nee, A.Y.C., 2004, "Developing distributed applications for integrated product and process design", Computer-Aided Design, vol. 36, n. 8, pp. 679-689.

134. Dugenske, A., Fraser, A., Nguyen, T. and Voitus, R., 2000, "The National Electronics Manufacturing Initiative (NEMI) plug and play factory", International Journal of Computer Integrated Manufacturing, vol. 13, n. 3, pp. 225-244.

135. Delamer, I.M. and Martinez Lastra, J.L., 2006, "Evolutionary multi-objective optimization of QoS-Aware Publish/Subscribe Middleware in electronics production", Engineering Applications of Artificial Intelligence, vol. 19, n. 6, pp. 593-697.

136. Association Connecting Electronics Industries (IPC), 2003, "IPC-2501 - Definition for Web-based Exchange of XML Data", Northbrook, USA.

137. Association Connecting Electronics Industries (IPC), 2001, "IPC2541 - Generic Requirements for Electronics Manufacturing Shop-Floor Equipment Communication Messages (CAMX) ", Northbrook, USA.

138. Association Connecting Electronics Industries (IPC), 2005, "IPC2546 - Sectional Requirements for Shop-Floor Equipment Communication Messages (CAMX) for Printed Circuit Board Assembly", Northbrook, USA.

139. Association Connecting Electronics Industries (IPC), 2001, "IPC2547 - Sectional Requirements for Shop-Floor Equipment Communication Messages (CAMX) for Printed Circuit Board Test, Inspection and Rework", Northbrook, USA.

140. Delamer, I.M. and Martinez Lastra, J.L., 2006, "Quality of service for CAMX middleware", International Journal of Computer Integrated Manufacturing, vol. 19, n. 8, pp. 784-804.

141. Delamer, I.M., Martinez Lastra, J.L. and Tuokko, R., 2004, "Design of QoS-aware framework for industrial CAMX systems", Proceedings of the Second IEEE International Conference on Industrial Informatics INDIN 2004, Berlin, Germany.

142. Object Management Group, 2005, "Unified Modeling Language: Infrastructure, version 2.0", Available from: http://www.omg.org/spec/UML/2.0/, March 2010.

143. Object Management Group, 2005, "Unified Modelling Language: Superstructure, version 2.0", Available from: http://www.omg.org/spec/UML/2.0/, March 2010.

144. Weilkiens, T. and Oestereich, B., 2007, "UML 2 Certification Guide - Fundamental and Intermediate Exams", Morgan Kaufmann Publishers, San Francisco, ISBN: 978-0-12-373585-0, pp. 320.

145. Joshi, J., 2006, "A comparison and mapping of Data Distribution Service (DDS) and Java Messaging Service (JMS) ", Real-Time Innovations, Inc., Whitepaper, Available from: www.rti.com, March 2009.

146. Ryll, M., 2006, "Entwicklung einer CORBA-basierten Applikation zur Überwachung und Visualisierung von modularen Produktionslinien", Diploma Thesis, University of Applied Sciences, pp. 110.

147. Li, Y., Zou, F., Wu, Z. and Ma, F., 2004, "PWSD: A Scalable Web Service Discovery Architecture Based on Peer-to-Peer Overlay Network ", in "Advanced Web Technologies and Applications", Springer Berlin / Heidelberg, pp. 291-300.

148. Makris, C., Panagis, Y., Sakkopoulos, E. and Tsakalidis, A., 2006, "Efficient and adaptive discovery techniques of Web Services handling large data sets", Journal of Systems and Software, vol. 79, n. 4, pp. 480-495.

149. Sun, Y., He, S. and Leu, J.Y., 2007, "Syndicating Web Services: A QoS and user-driven approach", Decision Support Systems, vol. 43, n. 1, pp. 243-255.

150. Sánchez, E., Portas, A., Pereira, A., Vega, J. and Kirpichev, I., 2007, "Remote control of data acquisition devices by means of message oriented middleware", Fusion Engineering and Design, vol. 82, n. 5-14, pp. 1365-1371.

151. Buccafurri, F., De Meo, P., Fugini, M., Furnari, R., Goy, A., Lax, G., Lops, P., Modafferi, S., Pernici, B., Redavid, D., Semeraro, G., Ursino, D., 2008, "Analysis of QoS in cooperative services for real time applications", Data & Knowledge Engineering, vol. 67, n. 3, pp. 463-484.

152. Cardoso, J., Sheth, A., Miller, J., Arnold, J. and Kochut, K., 2004, "Quality of service for workflows and web service processes", Web Semantics: Science, Services and Agents on the World Wide Web, vol. 1, n. 3, pp. 281-308.

153. Schmidt, D.C. and O'Ryan, C., 2003, "Patterns and performance of distributed real-time and embedded publisher/subscriber architectures", The Journal of Systems and Software, vol. 66, n. 3, pp. 213-223.

154. Tselikas, N.D., Dellas, N.D., Koutsoloukas, E.A., Kapellaki, S.H., Prezerakos, G.N. and Venieris, I.S., 2007, "Distributed service provision using open APIs-based middleware: OSA/Parlay vs. JAIN performance evaluation study", The Journal of Systems and Software, vol. 80, n. 5, pp. 765-777.

155. Tuma, P. and Buble, A., 2001, "Open CORBA Benchmarking", International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2001), Orlando, Florida.

156. Distributed Systems Research Group, Charles University, Prague, 2008, "Open CORBA Benchmarking", Available from: http://dsrg.mff.cuni.cz/~bench/, May 2010.

157. Gokhale, A.S. and Schmidt, D.C., 1998, "Measuring and Optimizing CORBA Latency and Scalability Over High-speed Networks", IEEE Transaction on Computers, vol. 47, n. 4, pp. 391-413.

158. Gray, N.A.B., 2004, "Comparison of Web Services, Java-RMI, and CORBA service implementations", Fifth Australasian Workshop on Software and System Architectures, Melbourne, Australia.

159. Juric, M.B., Rozman, I., Brumen, B., Colnaric, M. and Hericko, M., 2006, "Comparison of performance of Web services, WS-Security, RMI, and RMI–SSL", Journal of Systems and Software, vol. 79, n. 5, pp. 689-700.

160. Distributed Object Computing (DOC) Group for Distributed Real-time and Embedded (DRE) Systems, 2008, "Real-Time DDS Examination & Evaluation Project (RT-DEEP)", Available from: http://www.dre.vanderbilt.edu/DDS/, 07.10.2008.

161. Parsons, J., Xiong, M., Schmidt, D.C., Edmondson, J., Nguyen, H. and Ajiboye, O., 2006, "Evaluating the performance of Pub/Sub Platforms for Tactical Information Management", Whitepaper, Available from: http://www.omgwiki.org/dds/, May 2010.

162. Xiong, M., Parsons, J., Edmondson, J., Nguyen, H. and Schmidt, D.C., 2006, "Evaluating the performance of Publish/Subscribe platforms information management in distributed real-time and embedded systems", Whitepaper, Available from: http://www.omgwiki.org/dds/, May 2010.

163. Real-Time Innovations, Inc., 2008, "RTI Data Distribution Service 4.2 Architectural Overview", Real-Time Innovations, Inc., Whitepaper, Available from: www.rti.com, April 2010.

164. Gottschalk, S., Lin, M.C. and Manocha, D., 1996, "OBBTree: a hierarchical structure for rapid interference detection", Proceedings of the 23rd annual conference on Computer graphics and interactive techniques (SIGGRAPH), New Orleans, USA.

165. UNC Research Group on Modeling, Physically-Based Simulation and Applications, 2009, "RAPID - Robust and Accurate Polygon Interference Detection", Available from: http://gamma.cs.unc.edu/OBB/, April 2010.

166. Pardo-Castellote, G., 2005, "OMG Data-Distribution Service: Architectural overview", Real-Time Innovations, Inc., Whitepaper, Available from: www.omg.org, April 2010.

167. Real-Time Innovations, Inc., 2006, "Can Ethernet be Real time?" Real-Time Innovations, Inc., Whitepaper, Available from: www.rti.com, April 2010.

168. S200 High Performance Compact Brushless Servo Drives - Reference Manual,2008, Danaher Motion, I., Available from: www.danahermotion.com, April 2010.

169. Butenhof, D.R., 1997, "Programming with POSIX Threads", Addison-Wesley, Boston, ISBN: 0-201-63392-2, pp. 400.

170. Shepherd, G. and Wingo, S., 1996, "MFC Internals - Inside the Microsoft Foundation Class Architecture", Addison-Wesley, ISBN: 0-201-40721-3, pp. 736.

171. Geeknet, Inc., "TinyXml ", Available from: http://sourceforge.net/projects/tinyxml/, April 2010.

172. Techsoft, Inc., "Matrix TCL Lite 2.0", Available from: www.techsoftpl.com/, April 2010.

173. Real-Time Innovations, Inc., 2007, "RTI Data Distribution Service - User Manual", Available from: www.rti.com, April 2010.

Appendix A:

Listings of Module Configuration Files in XML-Format

Contents of the File: ModuleDescription_module1.xml

```xml
<?xml version="1.0" ?>
<FixtureModule>
      <id>1</id>
      <OccupiedSpace>
            <p1>
                  <x>-334.3</x>
                  <y>-28.5</y>
                  <z>28.5</z>
            </p1>
            <p2>
                  <x>60.0</x>
                  <y>28.5</y>
                  <z>-28.5</z>
            </p2>
      </OccupiedSpace>
      <device>
            <kind>LINEAR_CLAMP</kind>
            <id>1</id>
            <description>Nothing</description>
            <spatialdesc>
                  <x>0.0</x>
                  <y>0.0</y>
                  <z>0.0</z>
                  <rotx>0</rotx>
                  <roty>0</roty>
                  <rotz>0</rotz>
            </spatialdesc>
            <isLockable>true</isLockable>
            <applyforce>
                  <clampingrange>
                        <min>0</min>
                        <max>2500</max>
                        <unit>N</unit>
                        <resolution>1</resolution>
                        <clampingdirection>push</clampingdirection>
                  </clampingrange>
            </applyforce>
            <linearactuation>
                  <stroke_range>
                        <min>0</min>
                        <max>60</max>
                        <unit>MM</unit>
                        <resolution>0.0008</resolution>
                  </stroke_range>
            </linearactuation>
            <locate>
                  <maxReactionForce>5000</maxReactionForce>
            </locate>
            <library>NI_UMI7774_S200VTS.dll</library>
            <library-parameters>
                  <board_id>1</board_id>
                  <axis_id>1</axis_id>
                  <adcChannel>1</adcChannel>
                  <enc_resolution>2000</enc_resolution>
```

```xml
                    <pitch>1.6</pitch>
            </library-parameters>
            <feedbackdevices>
                    <device>2</device>
                    <device>3</device>
            </feedbackdevices>
        </device>
        <device>
            <kind>DISPLACEMENT_SENSOR</kind>
            <id>2</id>
            <description>SFD of Actuator</description>
            <sensedisplacement>
                    <sensing_info>
                            <min>0</min>
                            <max>1000</max>
                            <unit>MM</unit>
                            <resolution>0.0008</resolution>
                    </sensing_info>
            </sensedisplacement>
            <library>DisplacementSensor_EncoderS200Lib.dll</library>
            <library-parameters>
                    <board_id>1</board_id>
                    <axis_id>1</axis_id>
                    <enc_resolution>2000</enc_resolution>
                    <pitch>1.6</pitch>
            </library-parameters>
        </device>
        <device>
            <kind>FORCE_SENSOR</kind>
            <id>3</id>
            <description>Force sensor on actuator tip</description>
            <senseforce>
                    <sensing_info>
                            <min>0</min>
                            <mæx>2500</mæx>
                            <unit>N</unit>
                            <resolution>1.0</resolution>
                    </sensing_info>
            </senseforce>
            <library>KistlerForceSensor_UMI_ADC.dll</library>
            <library-parameters>
                    <board_id>1</board_id>
                    <device_channel>1</device_channel>
                    <minVolt>0</minVolt>
                    <mæxVolt>10</mæxVolt>
                    <mæxForce>2500</mæxForce>
            </library-parameters>
        </device>
</FixtureModule>
```

Contents of the file: ModuleDescription_module2.xml

```xml
<?xml version="1.0" ?>
<FixtureModule>
      <id>2</id>
      <OccupiedSpace>
            <p1>
                  <x>-334.3</x>
                  <y>-28.5</y>
                  <z>28.5</z>
            </p1>
            <p2>
                  <x>60.0</x>
                  <y>28.5</y>
                  <z>-28.5</z>
            </p2>
      </OccupiedSpace>
      <device>
            <kind>LINEAR_CLAMP</kind>
            <id>1</id>
            <description>Nothing</description>
            <spatialde  c>
                  <x>0.0</x>
                  <y>0.0</y>
                  <z>0.0</z>
                  <rotx>0</rotx>
                  <roty>0</roty>
                  <rotz>0</rotz>
            </spatialde u99 ?>
            <isLockable>true</isLockable>
            <applyforce>
                  <clampingrange>
                        <min>0</min>
                        <max>2500</max>
                        <unit>N</unit>
                        <resolution>1</resolution>
                        <clampingdirection>push</clampingdirection>
                  </clampingrange>
            </applyforce>
            <linearactuation>
                  <stroke_range>
                        <min>0</min>
                        <max>60</max>
                        <unit>MM</unit>
                        <resolution>0.0008</resolution>
                  </stroke_range>
            </linearactuation>
            <locate>
                  <maxReactionForce>5000</maxReactionForce>
            </locate>
            <library>NI_UMI7774_S200VTS.dll</library>
            <library-parameters>
                  <board_id>1</board_id>
                  <axis_id>2</axis_id>
                  <adcChannel>2</adcChannel>
                  <enc_resolution>2000</enc_resolution>
                  <pitch>1.6</pitch>
```

```xml
            </library-parameters>
            <feedbackdevices>
                    <device>2</device>
                    <device>3</device>
            </feedbackdevices>
        </device>
        <device>
            <kind>DISPLACEMENT_SENSOR</kind>
            <id>2</id>
            <description>SFD of Actuator</description>
            <sensedisplacement>
                    <sensing_info>
                            <min>0</min>
                            <max>1000</max>
                            <unit>MM</unit>
                            <resolution>0.0008</resolution>
                    </sensing_info>
            </sensedisplacement>
            <library>DisplacementSensor_EncoderS200Lib.dll</library>
            <library-parameters>
                    <board_id>1</board_id>
                    <axis_id>2</axis_id>
                    <enc_resolution>2000</enc_resolution>
                    <pitch>1.6</pitch>
            </library-parameters>
        </device>
        <device>
            <kind>FORCE_SENSOR</kind>
            <id>3</id>
            <description>Force sen_or on actuator tip</description>
            <senseforce>
                    <sensing_info>
                            <min>0</min>
                            <max>2500</max>
                            <unit>N</unit>
                            <resolution>1.0</resolution>
                    </sensing_info>
            </senseforce>
            <library>KistlerForceSensor_UMI_ADC.dll</library>
            <library-parameters>
                    <board_id>1</board_id>
                    <device_channel>2</device_channel>
                    <minVolt>0</minVolt>
                    <maxVolt>10</maxVolt>
                    <maxForce>2500.0</maxForce>
            </library-parameters>
        </device>
</FixtureModule>
```

Appendix B:
Data Type Definitions in IDL-format

Contents of the File: exampleApp.idl

```
enum ClampingDirection {
      push = 0,
      pull = 1,
      both = 2,
      unknown = 3
};

struct Force {
      long module_id;
      ClampingDirection clampingDirection;
      double value;
};

struct Position{
      long module_id;
      double x;
      double y;
      double z;
};

struct Point{
      double x;
      double y;
      double z;
};

struct OccupiedSpace {
      Point p1;
      Point p2;
};

struct SpatialDescription{
      double x;
      double y;
      double z;
      double rot_x;
      double rot_y;
      double rot_z;
};

struct Clocking{
      double rot_x;
      double rot_y;
      double rot_z;
};

struct BodyPositionInfo{
      long module_id;
      long tc_id;
      long slot_id;
      Point position;
      Clocking slotClocking;
      Clocking moduleClocking;
};
```

```
struct SensingInfo {
      double min;
      double max;
      long unit;
      double resolution;
};

struct BodyPosSensingInfo{
      SensingInfo posX;
      SensingInfo posY;
      SensingInfo posZ;
      SensingInfo moduleClockingX;
      SensingInfo moduleClockingY;
      SensingInfo moduleClockingZ;
      SensingInfo slotClockingX;
      SensingInfo slotClockingY;
      SensingInfo slotClockingZ;
};

struct ClampingRange{
      ClampingDirection clampingDirection;
      double minForce;
      double maxForce;
      long unit;
      double resolution;
};

struct StrokeRange{
      double min;
      double max;
      long unit;
      double resolution;
};

struct SwingRange{
      long axis;
      double cw_max;
      double ccw_max;
      long unit;
      double resolution;
};

struct ClockingRange{
      double cw_max;
      double ccw_max;
      long unit;
      double resolution;
};

struct ClockingRanges{
      ClockingRange clockingRange_x;
      ClockingRange clockingRange_y;
      ClockingRange clockingRange_z;
};
```

```
struct WorkSpace{
      StrokeRange linearRange_x;
      StrokeRange linearRange_y;
      StrokeRange linearRange_z;
      ClockingRange clockingRange_x;
      ClockingRange clockingRange_y;
      ClockingRange clockingRange_z;
};

struct ClampWorkSpace{
      StrokeRange strokeRange_x;
      StrokeRange strokeRange_y;
      StrokeRange strokeRange_z;
      SwingRange swingRange;
};

struct SenseTipPositionCapability{
      SensingInfo sensingInfo_x;
      SensingInfo sensingInfo_y;
      SensingInfo sensingInfo_z;
      boolean isSupported;
};

struct AdjustTipPositionCapability{
      ClampWorkSpace workspace;
      boolean isSupported;
};

struct SenseReactionForceCapability{
      SensingInfo sensingInfo;
      boolean isSupported;
};

struct SenseClampingForceCapability{
      SensingInfo sensingInfo;
      boolean isSupported;
};

struct AdjustClampingForceCapability{
      ClampingRange clampingRangePush;
      ClampingRange clampingRangePull;
      ClampingDirection clampingDirection;
      boolean isSupported;
};

struct SlotLinkInfo{
      long module_id;
      long tc_id;
      long slot_id;
      boolean isLink;
      SpatialDescription sdModule;
};

struct ClampRoleInfo{
      boolean isSupported;
};
```

```
struct LocatorRoleInfo{
      boolean isSupported;
      double maxForce;
};

struct SupportRoleInfo{
      boolean isSupported;
      double maxForce;
};

struct ProvidesRoleCapability{
      ClampRoleInfo     clampRoleInfo;
      LocatorRoleInfo locatorRoleInfo;
      SupportRoleInfo supportRoleInfo;
      boolean isSupported;
};

struct ModuleCapDefinition{
      long id;
      OccupiedSpace occupiedSpace;
      SenseTipPositionCapability senseTipPositionCapability;
      AdjustTipPositionCapability adjustTipPositionCapability;
      SenseReactionForceCapability senseReactionForceCapability;
      AdjustClampingForceCapability adjustClampingForceCapability;
      SenseClampingForceCapability senseClampingForceCapability;
      ProvidesRoleCapability providesRoleCapability;
};
```

Appendix C:

Source Code for the Device Libraries used in the

Prototype Application

Device Library for the Force Sensor Access –

Contents of the File: KistlerForceSensor_UMI_ADC.cpp

```cpp
// KistlerForceSensor_UMI_ADC.cpp : Defines the entry point for the DLL
application.
//
#include "stdafx.h"
#include "KistlerForceSensor_UMI_ADC.h"
#include "flexmotn.h"
#include "ForceSensor_UMI7774_ADCLib.h"


BOOL APIENTRY DllMain( HANDLE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                                 )
{
     switch (ul_reason_for_call)
     {
     case DLL_PROCESS_ATTACH:
     case DLL_THREAD_ATTACH:
     case DLL_THREAD_DETACH:
     case DLL_PROCESS_DETACH:
          break;
     }
   return TRUE;
}

void*  KISTLERFORCESENSOR_UMI_ADC_API  createLibraryInstance(TiXmlNode *
node){
     //if there is no parent node
     if ( !node )
          return NULL;

     //init
     int boardId = 0;
     int channelNumber = NIMC_ADC1;
     double minVolt = 0.0;
     double maxVolt = 10.0;
     double maxForce = 2500.0;


     do{
       //get type of node
       int t = node->Type();
       switch ( t ){
       case TiXmlNode::ELEMENT:
           if (strcmp(node->Value(), "board_id") == 0){
                 sscanf(node->FirstChild()->Value(), "%d", &boardId);
           } else if (strcmp(node->Value(), "device_channel") == 0){
                 sscanf(node->FirstChild()->Value(), "%d",
                                             &channelNumber);
           } else if (strcmp(node->Value(), "minVolt") == 0) {
                 sscanf(node->FirstChild()->Value(), "%lf", &minVolt);
```

```cpp
            } else if (strcmp(node->Value(), "maxVolt") == 0){
                    sscanf(node->FirstChild()->Value(), "%lf", &maxVolt);
            }else if (strcmp(node->Value(), "maxForce") == 0) {
                    sscanf(node->FirstChild()->Value(), "%lf",
                                                    &maxForce);
            }
                break;
        case TiXmlNode::UNKNOWN:
            break;
        }
    }while ((node = node->NextSibling()) != 0);

    //put the right constant for the channel_number
    switch(channelNumber){
        case 1:
                channelNumber = NIMC_ADC1;
                break;
        case 2:
                channelNumber = NIMC_ADC2;
                break;
        case 3:
                channelNumber = NIMC_ADC3;
                break;
        case 4:
                channelNumber = NIMC_ADC4;
                break;
        default:
                channelNumber = NIMC_ADC1;
                break;
    }


    return static_cast< void* > (new ForceSensor_UMI7774_ADCLib
                                    (boardId, channelNumber,
                                        minVolt, maxVolt,
                                            maxForce));
}
```

Device Library for the Force Sensor Access –
Contents of the File: KistlerForceSensor_UMI7774_ADCLib.cpp

```cpp
#include "StdAfx.h"
#include ".\forcesensor_umi7774_adclib.h"

#include <iostream>
#include <fstream>
#include <math.h>
using namespace std;

#include "flexmotn.h"
#ifndef NIMCEXAMPLE_H_INCLUDE
#define NIMCEXAMPLE_H_INCLUDE

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//constructor
ForceSensor_UMI7774_ADCLib::ForceSensor_UMI7774_ADCLib(int boardId, int
channelNumber, double minVolt, double maxVolt, double maxForce)
: boardId(boardId),
  channelNumber(channelNumber),
  minVolt(minVolt),
  maxVolt(maxVolt),
  maxForce(maxForce)
{
      //nothing
}

//destructor
ForceSensor_UMI7774_ADCLib::~ForceSensor_UMI7774_ADCLib(void){
      //nothing
}

//see header
bool ForceSensor_UMI7774_ADCLib::initialise(){
      //set adc range to 0..10V
       flex_set_adc_range(this->boardId, this->channelNumber,
                                             NIMC_ADC_UNIPOLAR_10);
      u16 adcMap = 3; //should be 0b0000000000000011 -> enable adc1 + 2
      flex_enable_adcs(this->boardId, 0, adcMap);
      //set adc range to 0..10V again..to make sure
      flex_set_adc_range(this->boardId, this->channelNumber,
                                             NIMC_ADC_UNIPOLAR_10);

      return true;
}

//see header
bool ForceSensor_UMI7774_ADCLib::closeDevice(){
      return true;
}

//see header
double ForceSensor_UMI7774_ADCLib::getCurrentValue(void){
      i16 adcValue = 0;
      i32 err;
```

- 231 -

```
        double forceNewton = 0.0;

        // Read the ADC channel
        err = flex_read_adc_rtn(this->boardId, this->channelNumber,
                                                    &adcValue);

        //transform the adc-value to a Newton-value...
        // 2^12 -1          2500         10V
        // ----          =      ----   =      ----
        //   x                y           z
        forceNewton = (double)((double)(adcValue * this->maxForce) /
                                            (double)4095.0);

        return forceNewton;
}
```

Device Library for the Displacement Sensor Access –
Contents of the File: DisplacementSensor_EncoderS200Lib.cpp

```cpp
// DisplacementSensor_EncoderS200Lib.cpp : Defines the entry point for
the DLL application.
//
///~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~###Includes~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#include "stdafx.h"
#include "DisplacementSensor_EncoderS200Lib.h"
#include ".\encoders200_umi7774lib.h"
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

BOOL APIENTRY DllMain( HANDLE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                                   )
{
      switch (ul_reason_for_call)
      {
      case DLL_PROCESS_ATTACH:
      case DLL_THREAD_ATTACH:
      case DLL_THREAD_DETACH:
      case DLL_PROCESS_DETACH:
            break;
      }
    return TRUE;
}

//see header
void* DISPLACEMENTSENSOR_ENCODERS200LIB_API createLibraryInstance(
                              TiXmlNode * node)
{
      //if there is no parent node
      if ( !node )
            return NULL;

      //init
      int boardId = 0;
      int axisId = 0;
      double encoder_resolution = 0.0;
      double pitch = 0.0;

      do{
        //get type of node
        int t = node->Type();
        switch ( t ){
        case TiXmlNode::ELEMENT:
            if (strcmp(node->Value(), "board_id") == 0){
                sscanf(node->FirstChild()->Value(), "%d", &boardId);
            } else if (strcmp(node->Value(), "axis_id") == 0) {
                sscanf(node->FirstChild()->Value(), "%d", &axisId);
            } else if (strcmp(node->Value(), "enc_resolution") == 0){
                sscanf(node->FirstChild()->Value(), "%lf",
                                            &encoder_resolution);
            }else if (strcmp(node->Value(), "pitch") == 0) {
                sscanf(node->FirstChild()->Value(), "%lf", &pitch);
```

```
        }
        break;

    case TiXmlNode::UNKNOWN:
        break;
    }
}while ((node = node->NextSibling()) != 0);

return static_cast< void* > (new EncoderS200_UMI7774Lib(boardId,
                                   axisId, encoder_resolution,
                                        pitch));
}
```

Device Library for the Displacement Sensor Access –
Contents of the File: EncoderS200_UMI774Lib.cpp

```cpp
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~IncludeS~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#include "StdAfx.h"
#include ".\encoders200_umi7774lib.h"
#include <stdio.h>
#include <iostream>
using namespace std;

//for hardware access
#include "flexmotn.h"
#ifndef NIMCEXAMPLE_H_INCLUDE
#define NIMCEXAMPLE_H_INCLUDE

//constructor
EncoderS200_UMI7774Lib::EncoderS200_UMI7774Lib(int boardId, int axisId,
                                double encoder_resolution, double pitch)
: boardId(boardId),
  axisId(axisId),
  encoder_resolution(encoder_resolution),
  pitch(pitch),
  stepOffset(0)
{
      //nothing
}

//destructor
EncoderS200_UMI7774Lib::~EncoderS200_UMI7774Lib(void){
      //nothing
}

//see header
bool EncoderS200_UMI7774Lib::initialise(){
      return true;
}

//see header
bool EncoderS200_UMI7774Lib::closeDevice(){
      return true;
}

//see header
double EncoderS200_UMI7774Lib::getCurrentValue(void){
      i32 positionInSteps;    // Current position of axis
      i32 err;
      //try to read the current axis position
      err = flex_read_pos_rtn(this->boardId,this->axisId,
                                      &positionInSteps);
      //transform the retrieved value in millimeters...
      //now it is dreisatz 1.6mm = 2000 Steps
      //                    ----   ---------
      //                    x mm = y Steps
      return ((this->pitch * positionInSteps) /
                            this->encoder_resolution);
}
```

Device Library for the Linear Actuator Access –
Contents of the File: NI_UMI7774_S200VTS.cpp

```cpp
//  NI_UMI7774_S200VTS.cpp  :  Defines  the  entry  point  for  the  DLL
application.
//
#include "stdafx.h"
#include "flexmotn.h"
#include "NI_UMI7774_S200VTS.h"
BOOL APIENTRY DllMain( HANDLE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                                  )
{
      switch (ul_reason_for_call)
      {
      case DLL_PROCESS_ATTACH:
      case DLL_THREAD_ATTACH:
      case DLL_THREAD_DETACH:
      case DLL_PROCESS_DETACH:
            break;
      }
    return TRUE;
}


//see header
void* NI_UMI7774_S200VTS_API createLibraryInstance(TiXmlNode * node){
      //if there is no parent node
      if ( !node )
            return NULL;

      //init
      int boardId = 0;
      int axisId = 0;
      double encoder_resolution = 0.0;
      double pitch = 0.0;
      int adcChannelNumber = 1;

      do{
            //get type of node
            int t = node->Type();
            switch ( t ){
            case TiXmlNode::ELEMENT:
              if (strcmp(node->Value(), "board_id") == 0){
                  sscanf(node->FirstChild()->Value(), "%d", &boardId);
              } else if (strcmp(node->Value(), "axis_id") == 0) {
                  sscanf(node->FirstChild()->Value(), "%d", &axisId);
              } else if (strcmp(node->Value(), "enc_resolution") == 0){
                  sscanf(node->FirstChild()->Value(), "%lf",
                                                &encoder_resolution);
              }else if (strcmp(node->Value(), "pitch") == 0) {
                  sscanf(node->FirstChild()->Value(), "%lf", &pitch);
              } else if (strcmp(node->Value(), "adcChannel") == 0) {
                  sscanf(node->FirstChild()->Value(), "%d",
                                                &adcChannelNumber);
              }
```

```
                break;

            case TiXmlNode::UNKNOWN:
                break;
        }
    }while ((node = node->NextSibling()) != 0);

    //put the right constant for the channel_number
    switch(adcChannelNumber){
        case 1:
            adcChannelNumber = NIMC_ADC1;
            break;
        case 2:
            adcChannelNumber = NIMC_ADC2;
            break;
        case 3:
            adcChannelNumber = NIMC_ADC3;
            break;
        case 4:
            adcChannelNumber = NIMC_ADC4;
            break;
        default:
            adcChannelNumber = NIMC_ADC1;
            break;
    }

    return static_cast< void* > (new NI_UMI774_S200VTSLib(boardId,
                                axisId, encoder_resolution, pitch,
                                             adcChannelNumber)
                                        );
}
```

**Device Library for the Linear Actuator Access –
Contents of the File: NI_UMI7774_S200VTSLib.cpp**

```cpp
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~####IncludeS~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#include "StdAfx.h"
#include ".\ni_umi774_s200vtslib.h"

// basic file operations
#include <iostream>
#include <fstream>
#include <math.h>
using namespace std;

#include "flexmotn.h"
#ifndef NIMCEXAMPLE_H_INCLUDE
#define NIMCEXAMPLE_H_INCLUDE
#include <conio.h>

//see header
NI_UMI774_S200VTSLib::NI_UMI774_S200VTSLib(int boardId, int axisId,
            double encoder_resolution, double pitch, int adcChannel)
: boardId(boardId),
  axisId(axisId),
  encoder_resolution(encoder_resolution),
  pitch(pitch),
  adcChannel(adcChannel),
  stepOffset(0)
{
    //nothing
}

//destructor
NI_UMI774_S200VTSLib::~NI_UMI774_S200VTSLib(void){
    //nothing
}


//see header
bool NI_UMI774_S200VTSLib::initialise(){
    f64 acceleration  =100; // Acceleration value in RPS/S
    f64 velocity      =200; // Velocity value in RPM
    u16 found, finding;     // Check Reference Statuses
    u16 axisStatus;         // Axis Status
    u16 csr=0;              // Communication Status Register
    i32 position;           // Current position of axis
    i32 scanVar;            // Scan variable to read in values
                            // not supported by the scanf function

    //Variables for modal error handling
    u16 commandID;                          //  The  commandID  of  the
function
    u16 resourceID;              // The resource ID
    i32 errorCode;                      // Error code
    i32 err;

    //Check if the board is at power up reset condition
```

```
err = flex_read_csr_rtn(this->boardId, &csr);

if (csr & NIMC_POWER_UP_RESET ){
     printf("\nThe FlexMotion board is in the reset condition.
                      Please initialize the board.");
     return false;
}

//Load acceleration and deceleration to the axis selected
err = flex_load_rpsps(this->boardId, this->axisId, NIMC_BOTH,
                                          acceleration, 0xFF);
//Load velocity to the axis selected
err = flex_load_rpm(this->boardId,this->axisId, velocity, 0xFF);

//configures the find_reference function, to automatically `reset
//IF a home position is found
flex_load_reference_parameter(this->boardId,this->axisId,
     NIMC_FIND_HOME_REFERENCE, NIMC_ENABLE_RESET_POSITION ,1);
flex_load_reference_parameter(this->boardId,this->axisId,
     NIMC_FIND_HOME_REFERENCE, NIMC_PRIMARY_RESET_POSITION, 0);
flex_load_reference_parameter(this->boardId,this->axisId,
     NIMC_FIND_HOME_REFERENCE,NIMC_SMART_ENABLE ,TRUE);

//configures the find reference to initially search reverse for
//the home position
flex_load_reference_parameter(this->boardId, this->axisId,
     NIMC_FIND_HOME_REFERENCE,NIMC_INITIAL_SEARCH_DIRECTION,
          true);

//Start the Find Reference move
err = flex_find_reference(this->boardId, this->axisId, 0,
     NIMC_FIND_HOME_REFERENCE);

//Wait for find reference to complete on the axis
do{
     //Read the current position of axis
     err = flex_read_pos_rtn(this->boardId,this->axisId,
               &position);
     err = flex_read_axis_status_rtn(this->boardId,this->axisId,
                                          &axisStatus);

     //Check if the reference has finished finding
     err = flex_check_reference(this->boardId, this->axisId, 0,
                         &found, &finding);
     //Read the Communication Status Register - check the
     //modal error bit
     err = flex_read_csr_rtn(this->boardId, &csr);
     if (csr & NIMC_MODAL_ERROR_MSG)
     {
          flex_stop_motion(boardId, NIMC_AXIS1,
                    NIMC_DECEL_STOP, 0);//Stop the Motion
          err = csr & NIMC_MODAL_ERROR_MSG;
     }
}while ( !(axisStatus & (NIMC_FOLLOWING_ERROR_BIT |
                         NIMC_AXIS_OFF_BIT)) && finding);
```

```cpp
        //wait a bit until he is really at position 0
        Sleep(5000);

        if (found){
              printf("\nAxis Found Home Position: Axis %d position:
                    %10d", this->axisId, position);
        }else{
              printf("\nAxis Did not Find Home Position: Axis %d
                    position: %10d  --- Please abort program",
                          this->axisId, position);
              char buf[500];
              scanf("%s", buf);
              return false;
        }

        //~~~~~~~~Initialise the acc, dec and velocity so we don't have
        //to do it during the operation all the time
        // Set the velocity for the move (in counts/sec)
        err = flex_load_velocity(this->boardId, this->axisId, 10000,
                                                            0xFF);
        // Set the acceleration for the move (in counts/sec^2)
        err = flex_load_acceleration(this->boardId, this->axisId,
                                      NIMC_ACCELERATION, 100000, 0xFF);
        // Set the deceleration for the move (in counts/sec^2)
        err = flex_load_acceleration(this->boardId, this->axisId,
                                      NIMC_DECELERATION, 100000, 0xFF);
        // Set the jerk (s-curve value) for the move (in sample periods)
        err = flex_load_scurve_time(this->boardId, this->axisId, 100,
                                      0xFF);
        err = flex_set_op_mode(this->boardId, this->axisId,
                                          NIMC_RELATIVE_POSITION);
        //initialise the ADC settings...just to be on the safe side....
        //set adc range to 0..10V
        flex_set_adc_range(this->boardId,
                          this->adcChannel,NIMC_ADC_UNIPOLAR_10);
        u16 adcMap = 1; //should be 0b0000000000000001
        flex_enable_adcs(this->boardId, 0, adcMap);

        return found;              // Finish
}

//see header
bool NI_UMI774_S200VTSLib::closeDevice(){
        return true;
}

//see header
bool NI_UMI774_S200VTSLib::applyForce(double targetForce, long
                                      desiredDirection)
{
        u16 axisStatus;                  // Axis status
        i32 constant;                        // Constant force
        i16 adcValue;                        // ADC value read
        i32 err;                          // Error code

        // constant force as an adc value that needed to be maintained
        constant = ((long)targetForce * 4095) / 2500;
```

- 240 -

```cpp
        // Check the move complete status/following error/axis off status
        err = flex_read_axis_status_rtn(this->boardId, this->axisId,
                                                &axisStatus);

        if(!(axisStatus & NIMC_AXIS_OFF_BIT)){
                //check if the move is complete - only do something if the
                //axis is currently not moving
                if(!(axisStatus & NIMC_MOVE_COMPLETE_BIT)){
                        return true;
                }

                err = flex_read_adc_rtn(this->boardId, this->adcChannel,
                                                        &adcValue);

                if( (constant - adcValue) != 0){
                        //adjust new relative position
                        int diff = constant - adcValue;

                        err = flex_set_op_mode(this->boardId, this->axisId,
                                NIMC_RELATIVE_POSITION);

                        err = flex_load_target_pos(this->boardId, this-
                                        >axisId, diff, 0xFF);
                        // Move based on delta force
                        err =  flex_start(this->boardId, this->axisId, 0);
                }
        }

        return true;
}

//see header
bool NI_UMI774_S200VTSLib::actuate(double targetActuation){
        //translate the desiredActuation into steps
        //now it is dreisatz 1.6mm = 2000 Steps
        //                   ----   ---------
        //                   x mm = y Steps
        long positionInSteps = (targetActuation *
                                this->encoder_resolution) / this->pitch;

        //initialise some variables
        u16 csr     = 0;
        u16 axisStatus;
        u16 moveComplete;
        i32 err;

        // Set the operation mode
        err =  flex_set_op_mode (this->boardId, this->axisId,
                                        NIMC_ABSOLUTE_POSITION);
        // Load Position as giving by the parameter
        err = flex_load_target_pos (this->boardId,this->axisId,
                                        positionInSteps, 0xFF);
        // Start the move
        err = flex_start(this->boardId, this->axisId, 0);

        do
        {
```

```
            axisStatus = 0;
            // Check the move complete status
            err = flex_check_move_complete_status(this->boardId,
                               this->axisId, 0, &moveComplete);
            // Check the following error/axis off status for the axis
            err = flex_read_axis_status_rtn(this->boardId,
                                  this->axisId, &axisStatus);
      }while (!moveComplete && !(axisStatus & NIMC_FOLLOWING_ERROR_BIT)
             && !(axisStatus & NIMC_AXIS_OFF_BIT));
      return moveComplete;          // Finish
}
```

Appendix D:

Diagrams for Force Profiles Followed by the Fixture Module During the Tests

Force Profiles for Fixture Module 2 during the first test:

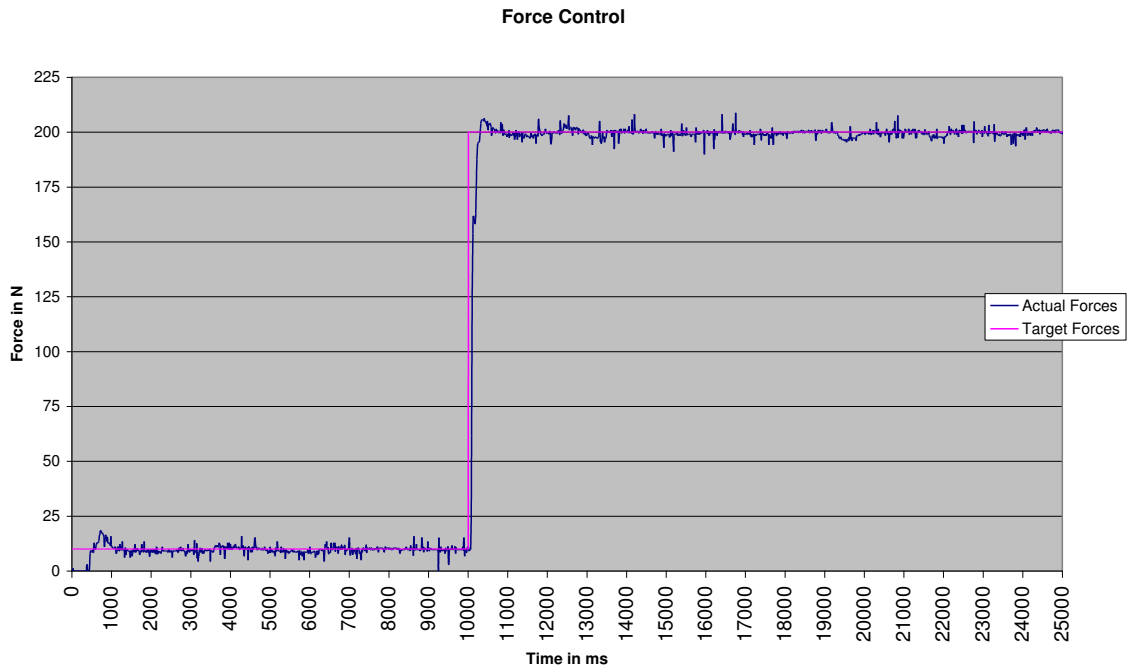**Force Control**
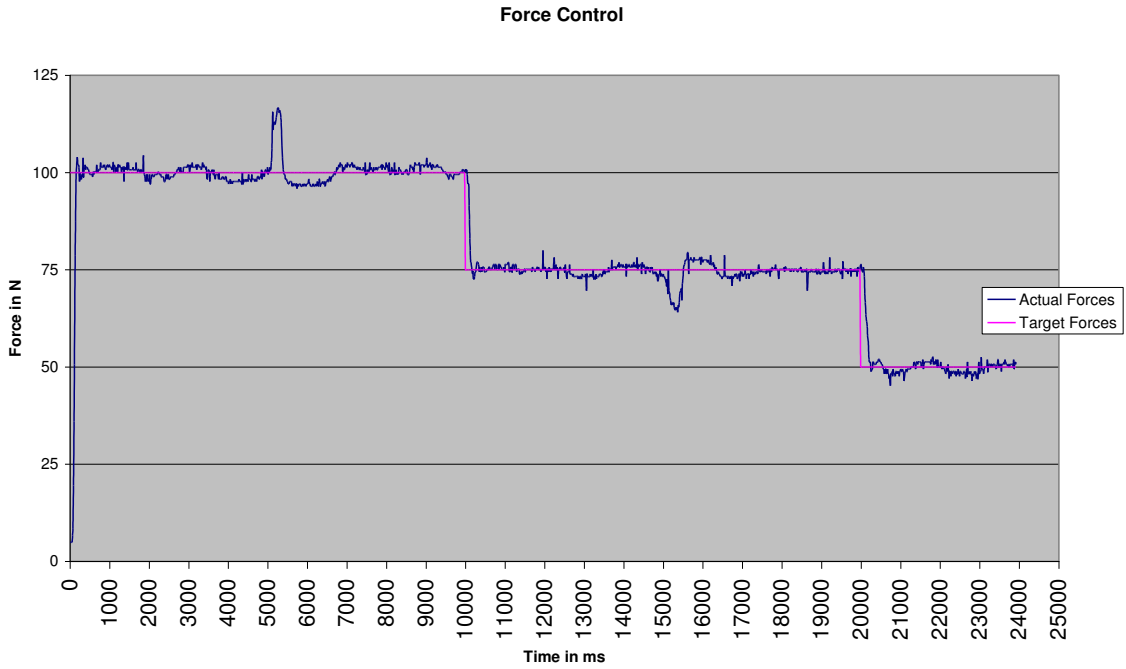
Figure D.1: Overall Force Profile of Fixture Module 2 during the First Test
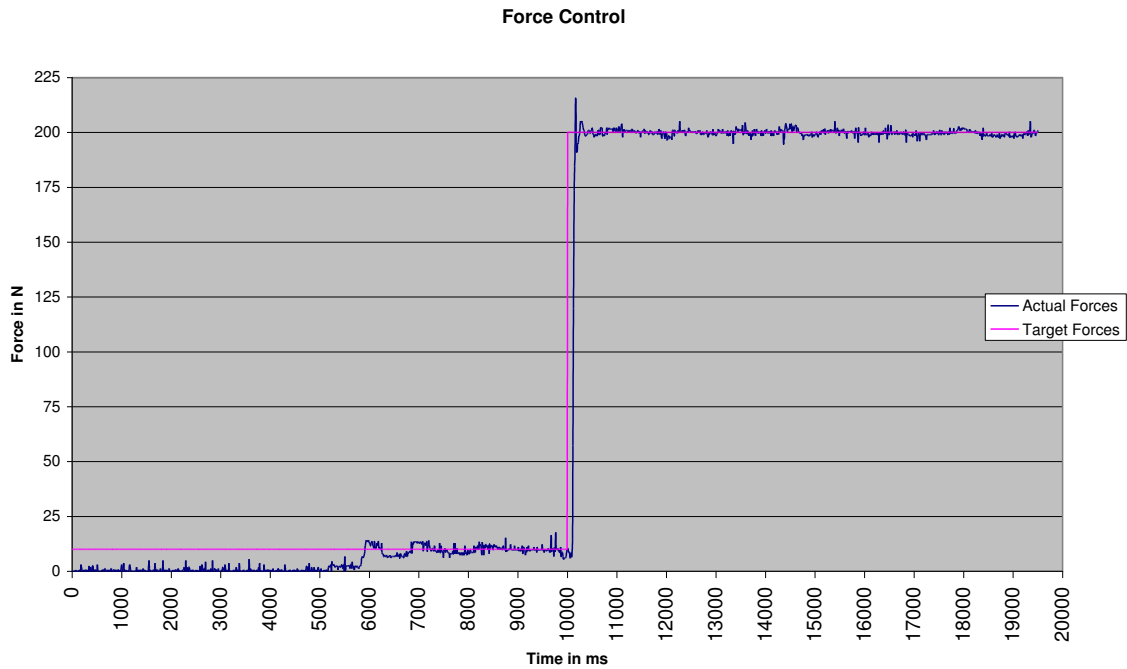
Force Control
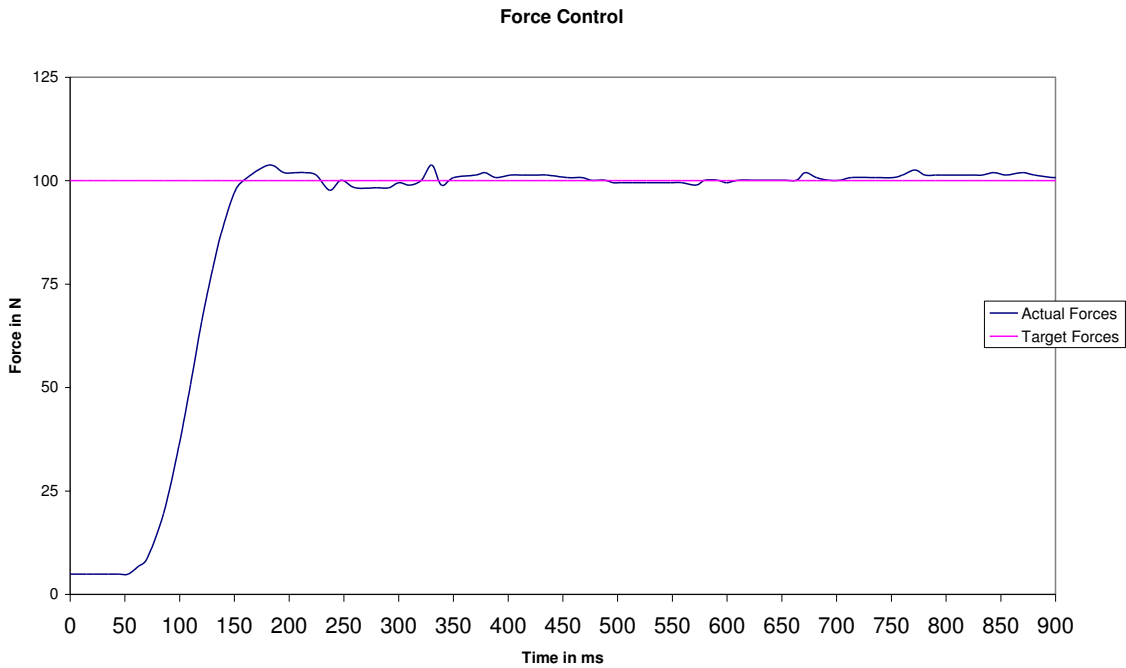
Figure D.2: Zoomed-in Force Profile of Fixture Module 2 during the First Test
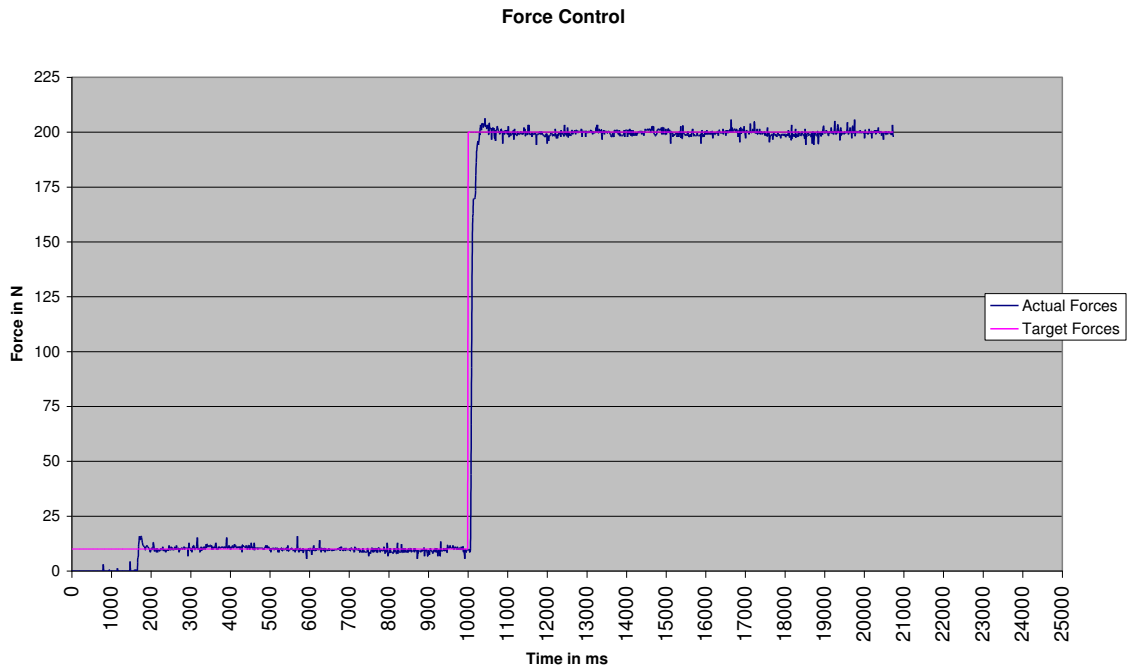
Force profiles during the second test (Workpiece A):

Force Control

Figure D.3: Force Profile of Fixture Module 1 during the Second Test

Force Control
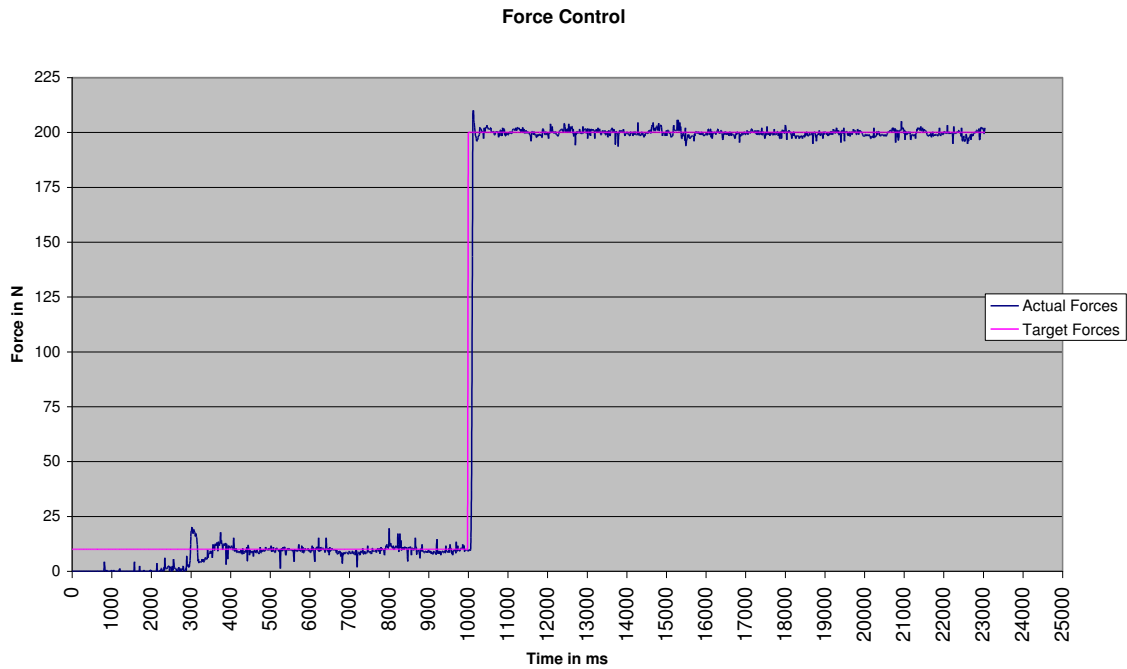

Figure D.4: Force Profile of Fixture Module 2 during the Second Test

Force profiles during the second test (Workpiece B):

Force Control

Figure D.5: Force Profile of Fixture Module 1 during the Second Test

Force Control

Figure D.6: Force Profile of Fixture Module 2 during the Second Test