DIPLOMA THESIS

Virtual Assembly of an Optical System and MEMS Components in a Minifactory

Christoph Bergler

2006/33

Prof. Dr. Ralph L. Hollis
The Robotics Institute
Carnegie Mellon University (CMU)
Adviser

Prof. Dr-Ing. Michael Zäh
Institut für Werkzeugmaschinen und Betriebswissenschaften
Technische Universität München (TUM)

Task

Thesis Title:

Virtual Assembly of an Optical System and MEMS Components in a Minifactory

Inv.- Nr.:	2006/33		
Author:	Christoph Bergler	**Adviser:**	Stella Clarke
Start date:	01.09.06	**End date:**	15.03.07

Background:
AAA Interface Tool as developed by the Carnegie Mellon University MSL is a tool for simulating, programming and controlling mechanical assembly work in a novel way. For the assembly of small size components, a 3D virtual Minifactory has been developed based on AAA Interface Tool.

The assembly of 3D MEMS structures goes beyond the actual capabilities of the virtual minifactory. The micro components are manufactured in a plane, from which they have to be collected, rotated by 90° and pushed into holders on another plane.
The 90° rotation requires an additional degree of freedom which is currently not available in the existing minifactory.

Objective:
The work consists of designing a virtual minifactory which has the additional degree of freedom. By this, the virtual minifactory shall be enabled to cope with the assembly of the MEMS structures. To do this, several virtual end effectors of the multi-agent minifactory have to be modelled and programmed. In order to show proper minifactory functionality, the operation of the assembly shall be shown in a simulation.

Method:
In order to get familiar with the concept of AAA and especially with the 3D virtual minifactory, a new assembly project will be simulated which bases on already existing components. After that, new end effectors shall be introduced, which comprise the additional rotational axis. Programming of these end effectors and of the whole minifactory will then lead to a complete simulation of the MEMS structures assembly.

Preface

I would like to thank Prof. Ralph Hollis for his guidance, his support and the opportunity to write this thesis at the Microdynamic System Laboratory and for the warm reception in his team. I also would like to thank Prof. Michael Zäh for offering the opportunity to do my diploma thesis abroad. Many thanks go to Stella Clarke for establishing the connection to the Microdynamic System Laboratory at Carnegie Mellon University and advising me in Germany. I would like to thank Sherif Zaidan for taking over the second part of the mentoring of this thesis.

Many thanks go to Cornelius Niemeyer for the many fruitful discussions, his drive and the good collaboration as lab mate as well as Jay Gowdy, Mark Dzmura, Jacob Thomas and Bertram Unger for their support and valuable counsels.

Abstract

This thesis describes the construction and programming of two three-dimensional assembly systems in the framework of the Minifactory. The Minifactory is a modular, flexible and agile assembly system that is developed by Microdynamic Systems Laboratory (MSL) at Carnegie Mellon University (CMU). It consists of two main bodies: the physical representation of an assembly system, which is used to automatically assemble different mechanical systems with high precision; and a virtual representation of the same station, which is used to set up and test the workflow for the assembly in the real world.

The Interface Tool as a component of the Minifactory concept, provides an environment for creating and programming virtual factories, which could be afterwards directly ported to the physical Minifactories.

Within this thesis, a first project was realized in cooperation with an optical devices manufacturer. The existing Minifactory set-up was herein used to model the assembly of telescopic sights.

In a second project, limitations of the existing Minifactory set-up were reduced. For the assembly of three dimensional MEMS (Micro-Electro-Mechanical-System) structures, the number of degrees of freedom in the existing Minifactory system did not suffice to handle the task at hand. Thus, an assembly tool was developed that provides an additional controllable movement axis and thereby an additional degree of freedom. This newly designed axis had to be implemented to the Interface Tool. To accomplish this, the source code of the Interface Tool had to be accordingly modified and upgraded.

The newly implemented axis does not only allow the set up of the MEMS assembly system in this project, but also expands the application field of the Minifactory for the future.

Zusammenfassung

Diese Arbeit beschreibt die Konstruktion und Programmierung von zwei virtuellen dreidimensionalen Montageanlagen im Rahmen des „Minifactory“- Projekts. „Minifactory“ ist ein modular aufgebautes, flexibles und schnell anpassbares Montagesystem für Produkte in einer Größenanordnung von Mikrometer bis Dezimeter, das im Microdynamic Systems Laboratory (MSL) an der Carnegie Mellon University (CMU) entwickelt wird. Es besteht im Wesentlichen aus zwei Teilen: die physikalische Verwirklichung einer Montageanlage, die verschiedenste mechanische Systeme automatisch und mit hoher Präzision zusammensetzt; und die virtuelle Abbildung derselben Anlage, die dazu benutzt wird, die physikalische Montage vorzubereiten und zu testen.

Das Interface Tool ist ein Teil des Minifactory Konzepts, das die Umgebung schafft, um virtuelle Fabriken zu erzeugen und Programme für die physikalischen Minifactories zu schreiben.

Innerhalb dieser Diplomarbeit wurde zunächst ein Projekt im Auftrag eines Herstellers von optischen Systemen bearbeitet. Aufgabe war die Erstellung einer Simulation für die Montage eines Zielfernrohrs.

In einem zweiten Projekt wurden Limitierungen, denen der bis dahin existierende Minfactory Aufbau unterlag behoben. Für die Zusammensetzung von dreidimensionalen MEMS- (Mikro-Elektro-Mechanisches-System) Strukturen reichte bis zu diesem Zeitpunkt die Anzahl Freiheitsgrade in der existierenden Minifactory nicht aus. Im Rahmen dieser Arbeit wurde daher ein Montagewerkzeug entwickelt, das eine zusätzliche kontrollierbare Bewegungsachse ermöglicht, und somit den zusätzlich benötigten Freiheitsgrad liefert. Diese neu entwickelte Achse musste in das Interface Tool integriert werden. Dazu musste der Quellcode des Interface Tools analysiert, modifiziert und ergänzt werden.

Die neu eingeführte Achse erlaubt nicht nur den Aufbau einer MEMS- Montageanlage, wie es in diesem Projekt behandelt wurde, sondern erweitert das Anwendungsgebiet der Minifactory auch für zukünftige Projekte.

Contents

1 Introduction

Nowadays, three trends become more and more apparent for the development of products from the high tech industries. On the one hand, product life cycles shorten rapidly. There are several explanations for this development. The pressure of competition increases constantly and new products have to be brought to market in shorter time intervals. Also, as a second trend, the fast progression of technical innovations leads to the fast obsolescence of many products. In particular, the electronic industry has to face that their products are subjected to a permanent technological change. On the other hand, the number of product variants increases continuously. Due to the rising degree of individualization the customers care to choose between several variations of a product or order a custom product. The miniaturization of products presents the third trend. More features have to be integrated on smaller areas. To save space and weight, products shrink and their attributes increase in quantity at the same time. Break throughs in the sector of nanotechnology fueled this trend. The MEMS (Micro-Electro-Mechanical System), for example, is a result of this development. Each of these trends poses a challenge for the fabrication systems. Some products are affected by all three trends. In these cases restrictions usually have to be accepted, since few assembly systems can meet all the latter demands.

The Agile Assembly Architecture (AAA) philosophy developed at the Microdynamic Systems Laboratory (MSL) at Carnegie Mellon University (CMU) and its instantiation in the Minifactory is an answer to these challenges. With its modular structure, easy manageability and high accuracy, Minifactory can meet the described demands.

Minifactory is still a prototype that is subject to permanent development and enhancement. During this process, there are research projects together with companies which could be future users or buyers. This thesis deals with the latest two projects. The first project was undertaken in cooperation with a telescopic-sight manufacturer. In a precision assembly, differently sized and

shaped lenses had to be placed in a collimator housing and subsequently fixed with screwed-in retainer rings. In this thesis, the generation of a virtual version of this Minifactory is described. This project proves the practical applicability of Minifactory and demonstrates the qualification of Minifactory for critical assembly tasks that require a high degree of accuracy. The second project describes the cooperation between MSL and a company specialized on the fabrication and assembly of MEMS. The assembly task consists of building a 3D MEMS structure by picking up a micro component, rotating it 90° and placing it perpendicularly in a retainer. However, up to then, Minifactory did not provide a fifth degree of freedom to rotate the part. Thus, besides creating a virtual factory for this task, a major part of this project was implementing an additional axis in the simulation environment of Minifactory. The additionally obtained degree of freedom did not only allow to finalize this project successfully, but also expands the application field of Minifactory in general.

This report is composed of three main chapters. In chapter 2, an introduction to the concept and the environment of Minifactory is given. The generation of the virtual version of the Minifactory that assembles the telescopic sights is described in chapter 3. Chapter 4contains the implementation of the fifth axis of the Minifactory and the creation of the virtual assembly of the 3D MEMS structures.

2 The Agile Assembly Architecture

The concept of the Agile Assembly Architecture (AAA) (Hollis 1995) was developed at the Microdynamic System Laboratory (MSL) at Carnegie Mellon University (CMU).

It presents a forward-looking approach to meet all demands that are made on recent automated assembly systems. Beyond a high degree of flexibility that allows versatile applications, AAA provides the agility to adapt to a rapidly changing product market.

To achieve these characteristics, the idea of AAA banks on a modular structure of the assembly system whose elements can be reused and reassembled to new systems over and over - like building blocks in a construction kit. Standardized basic modules guarantee a high reusability, which can be adapted to diverse applications easily and quickly. Standardized data protocols and standardized mechanical and electrical interfaces facilitate an arbitrary combination of all elements and a future extension of the assembly system.

An AAA assembly system is composed of robotic modules that are computationally independent and therefore do not depend on a central control unit during operation. These modules know about their capabilities and communicate with each other via network.

Another characteristic of the AAA philosophy is the close alliance between a real assembly system and an identical virtual version of it. An Interface Tool provides a basis for the virtual environment and the robotic modules supply information about their geometrical models and their behaviour. By means of the Interface Tool virtual assembly systems can be designed, programmed and simulated with actual module specifications, which can be loaded remotely via Internet. Standardized protocols and a huge library of routines as well as structured robotic agent autonomy simplify the generation of virtual systems and simulations. In these simulations processes can be tested and failures can be detected at an early stage.

Due to the modular character of AAA and standardized interfaces the real assembly system is built up easily. After the assembly and the alignment of the system elements the program, which has already been tested in the simulation can be uploaded to the robotic modules. A simple transition from simulated to real assembly is supported by the robotic modules' capability to self calibrate and to explore their environment.

Thus, long set-up times are avoided and a workable assembly system can be built up in a very short time.

2.1 Minifactory

Minifactory represents one way of implementing the Agile Assembly Architecture philosophy. Designed at the MSL, Minifactory is a modular assembly system cut to products in an order of magnitude of few micrometers up to some centimeters. Realized as a tabletop system, Minifactory consists of basic modules that can be equipped with robotic modules necessary for the respective application. A typical Minifactory set up is shown in *Figure 1*.

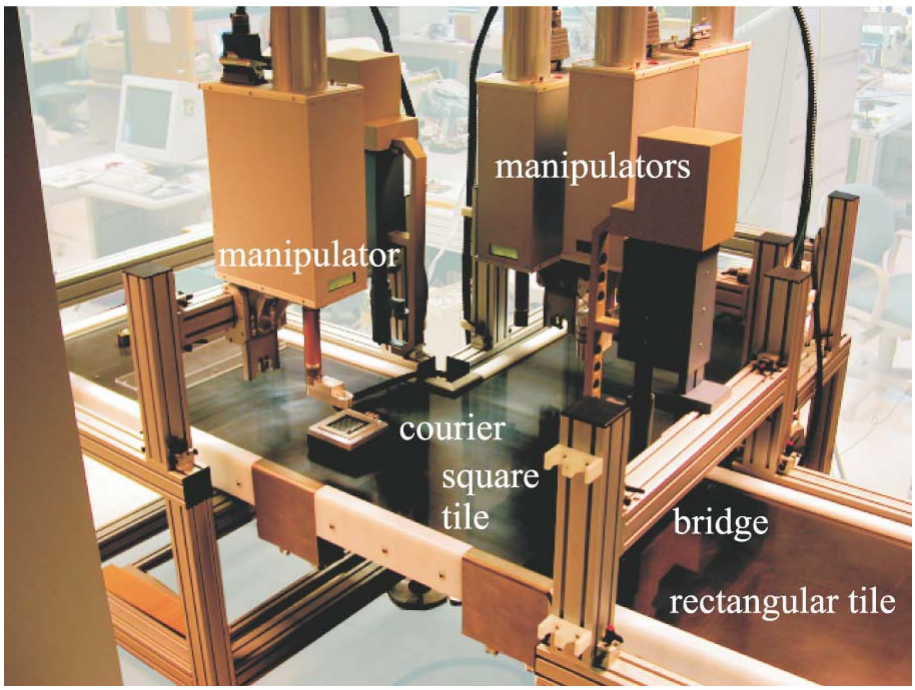


Figure 1: T-shaped Minifactory at MSL

Base frame modules (*Figure 2*), made of aluminum profiles, form the basic structure of the factory to which all other modules can be attached. Each *base*

frame includes a *base unit*, a service module, which supplies up to eight agent modules with power, pressured air, vacuum and network services. The connection between *base unit* and robot modules is made with a single multi-core cable and standardized connectors. The robots communicate via a global 100Mbit network using standard IP protocols and a local 100Mbit network that is adapted to real time capabilities.

On top of the *base frame* a *platen tile* is mounted, that functions as factory floor (*Figure 2*). The *platen tile* consists of a grid of ferromagnetic posts with edge lengths and pitches of 1 mm (in each case) embedded in epoxy to shape a planar platen surface. Polyethylene curbs surround the *platen tile* borders in order to prevent inadvertent falling of the transportation robot.

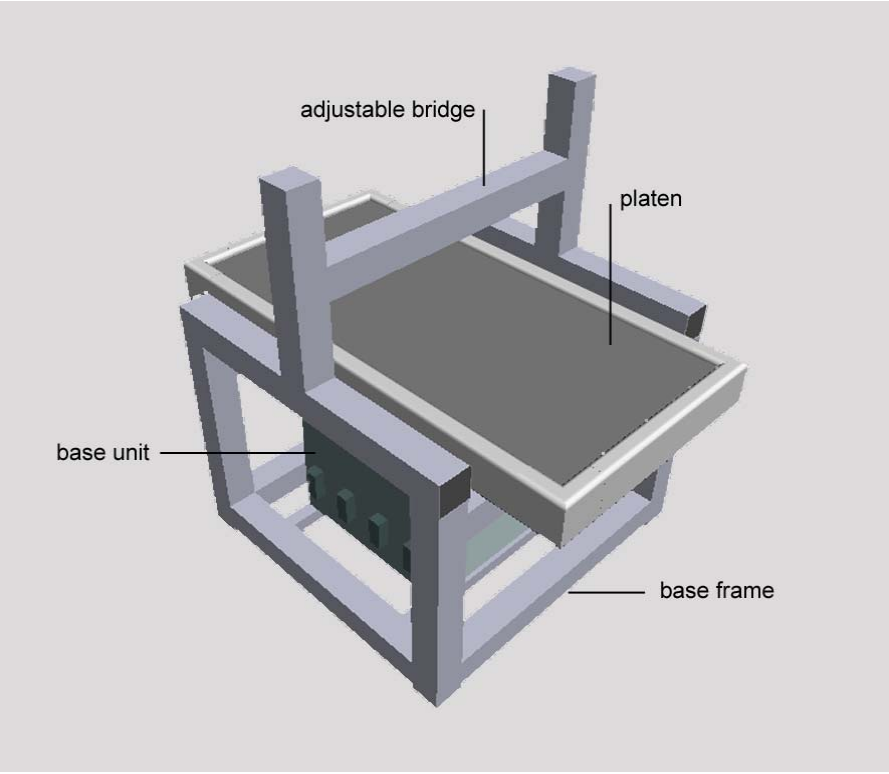


Figure 2: Minifactory unit

The robotic modules, referred to as *agents*, can be divided into two main classes. A characteristic that both types have in common is the small number of degrees of freedom (DOF) they provide. If these robots co-operate, however, one receives a higher DOF and is able to master complex tasks of assembly.

The first class summarizes the *courier agents*. These robots have two DOFs and can move freely on the table level. The task of the *couriers* is both the product transport within the factory and to transiently form cooperative three- to five-DOF manipulators.
The second class contains the *manipulator agents*. These agents are attached to *bridges* that are fixed to the *base frame* and span the tiles. Some manipulators are designed according to the functionality of a SCARA (Selective Compliance Assembly Robot Arm) assembly robot that is reduced to two DOF - the linear and one rotatory axis. Various end effectors can be attached easily to these manipulators and make them flexibly applicable. Apart from this type of manipulator, similar agents exist, which have only one linear axis or offer an additional rotatory axis. For some join technologies, as bolting small screws, there are specialized manipulators. The manipulator agents are responsible for the actual assembly process and accomplish the join process.
In the two projects presented in this report, courier agents and standard two-DOF overhead manipulator agents are used. In the following sections these two types of robot agents will be presented in detail.

The minimal layout of a Minifactory consists of a base frame with integrated base unit, a platen, a bridge and the respective courier and manipulator agents. This single unit can be upgraded with up to six other robot agents and several units can be combined to an entire assembly system. With the aid of connecting members and connecting plates, different layouts can be formed with a user-defined number of units.

2.1.2 Courier

In contrast to most conventional assembly systems, conveyor belts or similar systems are not used for the product transport in Minifactory. Courier agents carry the individual parts and sub-assemblies to the assembly stations. Since the couriers can move freely in X and Y direction on the platens, it is not required that the assembly stations are in a pipelined configuration or are arranged according to the processing sequence. Another field of functions of the

courier is co-operating with manipulators in assembly operations. Both couriers and manipulators have only a small number of DOFs, which is insufficient for most assembly tasks. But if a courier and a manipulator act like a team, complex processes that require more DOFs can be performed. For the co-operation, one of the agents gives up its autonomy and is controlled by the other agent's computer.

A courier unit consists of the mobile courier cube and a so called brain box, which is clamped to the side of the base frame. The brain box contains most of the electronics and the computing hardware and is plugged in one base unit with multi-core cable. A tether that is connected to the brain box manages the power and pressured air supply of the courier cubes. The length of the tether is the only thing that sets boundaries to the mobility of the courier.

The courier agents are floating on air bearings at an altitude of 10 -15 µm (Hollis 2003) over the platen factory floor. Four planar stepper motors exploiting the Sawyer principle in combination with the grid of ferromagnetic teeth in the platen, which provide the reaction force, activate the courier. For exact positioning, a platen sensor is used that enables close loop control at a resolution of 0.2 µm (Gowdy 1999) and at a speed of 1.5 m/s (Quaid 1998). Additionally the courier cube features an optical coordination sensor that has the ability to detect and measure the relative distance to LED beacons integrated in end effectors of manipulators with a resolution of 0.15 µm (Ma 2000).

On the topside of the courier cubes, mounting possibilities are located for task specific appliances.

2.1.3 Manipulator

The overhead manipulator with two axes (two DOF) represents the standard manipulator in the Minifactory modular system and is most flexibly applicable. It is able to move vertically along a Z-axis in a range of 150 mm with a resolution of 5 µm and to rotate 570° in θ around Z with a resolution of 0.0002° (Brown 2001). This type of manipulator agent is based on the principle of the common 4-DOF SCARA assembly robot, but presents a version which is reduced to two axes. A courier agent that co-operates with the manipulator pro-

vides the other two DOFs. With this division of labor, the disadvantage of SCARA robots in terms of insufficient accuracy is eliminated. By the omission of heavy robot arms and serial kinematic linkages with relatively flexible joints the accuracy and operation speed of the system increases.

The complete mechanics, electronics and computing hardware are placed in another brain box belonging to the manipulator. It is attached to bridges and can be positioned at arbitrary positions over the factory floor. An easy transportation and handling is guaranteed by the compact design of the robotic agents and is an implementation of the modularity concept.

For different applications the manipulators can be equipped with several end effectors, which can be interconnected by an electrical and pneumatic interface (*Figure 3*).

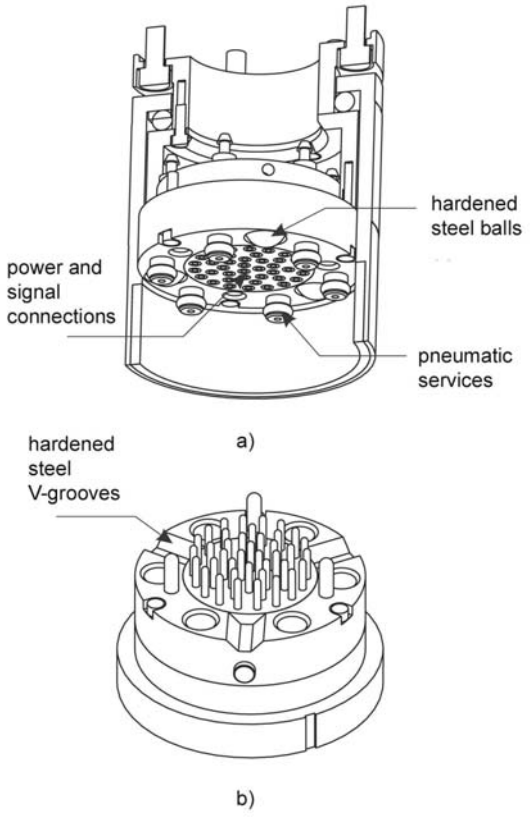


Figure 3: Connecting interface between manipulator and end effector: a) female connector, b) male connector

This interface supplies the end effectors with several power supply lines, pressured air and vacuum channels, which all can be actuated separately. Beside

the supply lines there are circuits for common signal input as well as for video transmission signals.
Due to a standardized quick connection at the interface, the end effectors can be replaced fast and easily.
Depending on the assembly task end effectors with different capabilities are chosen.
For instance, end effectors with vacuum grippers, tweezers or with a jammer tip have been designed and manufactured. Some end effectors are equipped with a video camera and enable vision based operations. Many other end effectors have been designed for diverse virtual factories.

2.2 Field of Application of Minifactory

Minifactory was developed for micro assembly tasks. Because of the high fidelity of the movements and accuracy of the robot agents, it is not only predestinated for very small products, as MEMS (Micro-Electro-Mechanical System) parts, but also for medium scaled products that feature very small tolerances, as optical products. The dimensions of the courier cubes and the workspace between courier topside and the overhead manipulators is the only limit for the maximum size of the products that can be assembled with Minifactory. Therefore, it is possible to assemble products that measure up to several decimeters, but a loss in throughput has to be taken into account, because one courier can only transport a single or a few parts at a time. It is the accuracy of the system which limits the minimum size of the products.
Due to the compact architecture of Minifactory, it would not be complex and expensive to set Minifactory up in a clean room. Furthermore, from the outset abrasion-resistant materials have been employed and mechanisms have been designed in a manner that no abrasion particles can escape the modules' housing. Thus, the requirement for clean working environment, which is indispensable for many parts in micrometer scale, can be met without problems.
The throughput of Minifactory depends on several factors. The number of assembly steps as well as the complexity of the assembly operations is respon-

sible for the time a product requires passing through the assembly system. Also, as aforementioned, the number of items transported by a courier affects the throughput. By dint of high accelerations and accurate retardations of the couriers and high maximum velocities of the manipulator agents, operations can be executed speedy. In general, Minifactory is designed for small to medium quantities, in certain cases large-scale production is expedient, too. In particular, Minifactory is the optimal solution for products that are subject to frequent technological change or have short life time cycles. Minifactory was planned according to the principle of agility and therefore can be adapted to new requirements very smoothly. Because of the short set-up times, the reusability of the system components and the high accuracy, it becomes often cost-efficient to automatize assembly tasks which are till now manually operated.

2.3 Interface Tool

In common proceeding, the development of an automated assembly system is divided into two distinct stages. First, the system is designed and programmed in a simulated environment “off-line”. After that, the results of the first stage are used “on-line” to reduce the deployment and integration time of the physical machines. The gap between on- and off-line systems is so great that usually two different software environments are used for the design phase and the deployment and operation phase (Gowdy 1999). Interface Tool combines both the environment for planning and creating virtual factories and the platform for programming the real factory.

Programming robots off-line offers theoretically the opportunity to minimize the effort and the time for setting up an assembly system. A virtual version of the planned factory can be programmed and tested in simulation even before the physical factory is installed. Thus, tests can show planning errors at an early stage and prevent later expensive modifications at the real system. Writing programs for a future task for an already existing system off-line while it is executing current task, reduces machine idle time. Performing test runs with

the virtual factory is not dangerous and cannot cause damages like dry runs with the physical factory. In practice, the off-line programming is only truly useful, if the virtual system matches the physical system to a sufficient degree in terms of geometrical structure and functional abilities. Otherwise, programs that have been created in off-line mode have to be modified and adapted to the real status. Accordingly, time saving is minimized and test results are then in many cases invalid.

AAA Interface Tool provides features that guarantee a highest possible correlation between both worlds, the virtual and the physical.

At first, all physical existing Minifactory modules are registered with their description in the so called component palette list of the Interface Tool (see *Figure 4*). That means, information about the body structure, the ability and the way of the connection to other components of each module is entered. The component palette is a list of iconified representations of the modules along with buttons to insert the representation into the virtual environment (Gowdy 1999). Since for creating diverse assembly systems only a small number of different standardized modules is necessary and the modules are assembled according to the same rules in the real as well as in the simulated environment, most big disparities are eliminated from the outset. The component palette list is not a passive catalog, but draws its data content via Internet from the several robotic agents, which have the ability to represent themselves to the Interface Tool. Thereby, the list can be updated easily whenever already uploaded agents get modified or new agents are included in the range of Minifactory modules.

To assure a high degree of analogy between the two systems during the complete lifetime of a Minifactory, Interface Tool offers the possibility of transitions between simulation and reality in both directions. This means that once the programming of the physical assembly system has to be modified, the changes can also be transferred to the simulated system easily. Contrary to most common simulation approaches, where transition from simulation to reality is just done once, with Interface Tool the full potential of simulation can be utilized. Lastly, the real robot agents have the capability to calibrate them-

selves and explore their physical environments. Thereby, little gaps between the simulated and the physical factory can be compensated.

With these features the approach of Interface Tool provides a high degree of agility – the over all goal of AAA.

2.3.1 Structure of the Interface Tool

The structure of Interface Tool consists of different layers that are constructed one on top of the other.

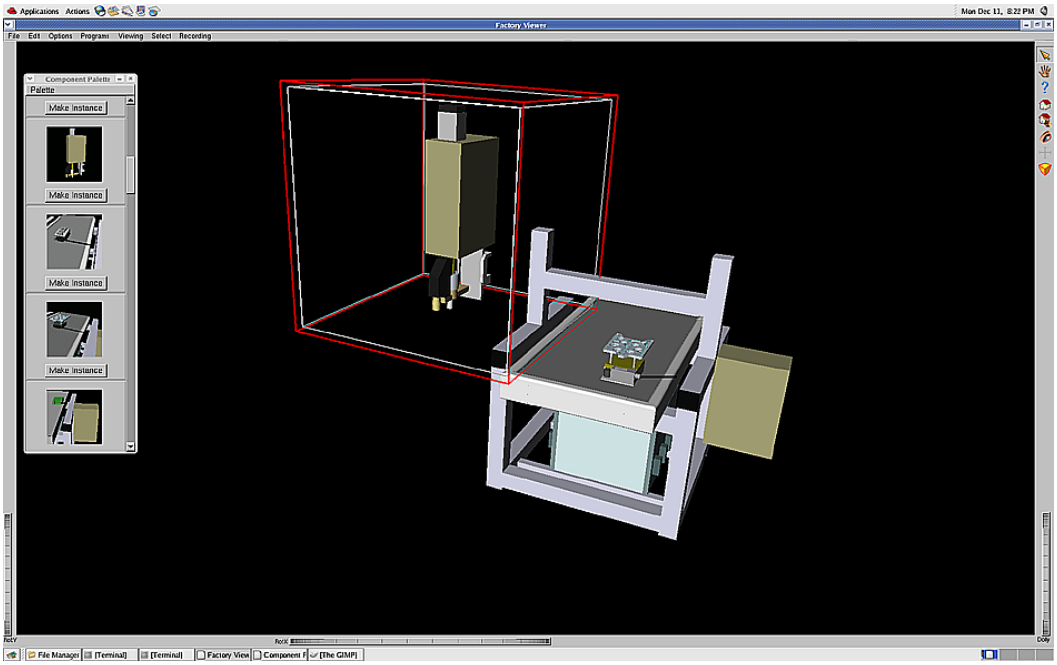


Figure 4: GUI of the Interface Tool with the component palette on the left

On the highest-level, Interface Tool provides a graphical user interface (GUI) that allows the factory designer to construct factories in a very convenient way (*Figure 4*). Based on the Open Inventor platform, this part of the Interface Tool offers all benefits of the windows design, as menu bar and dialog boxes. In this environment, Minifactories can be assembled (*Figure 5*) by choosing the required components from the component palette, which are then linked by a few commands. Further information about the building of a virtual factory with Interface Tool can be found in chapter 3.2.3.3.

The user interface also provides 3D rendering of the running simulated factory as a whole. The factory designer can change the viewing angle, zoom in/out

and regulate the execution speed/ simulation velocity. Once the real Minifactory has been set up and is operating, it does not depend on the Interface Tool anymore, as after uploading programs from the simulated factory to the real robot agents each one has its own controller and operates independently of a central control unit. The GUI, however, can be used to display the processes in real time and serve as a monitoring device.

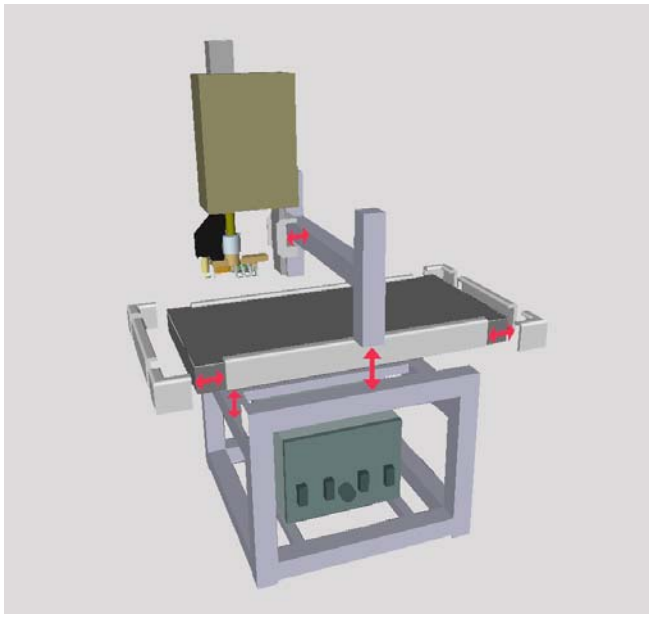


Figure 5: Assembly of the Minifactory components

The programs for the simulated and the physical Minifactories, respectively, are written in Python (a byte-coded, object orientated programming/script language) at a lower level. With the Python program, the user specifies the run-time behavior of the agents. These programs instantiate objects together with its required methods. The program objects have a *bind* method which is used to specify all of the global factory components the agents will use, and a *run* method which is the actual script that defines the run-time behavior of the agents (Gowdy 1999).

By assembling a virtual factory with the GUI, the basic structure of the Python program is already built. The chosen components are included with their position matrixes and parent-child-relations in the Python script automatically. The user just has to add the command lines which determine the behavior of the

agents to the script. There is a huge library with diverse commands for each type of robot agent to manage many different assembly tasks with Minifactory.
The Python level is based on data coded written in the source language C++. This architecture reduces the effort for programming the factories, as Python is a quite incomplex language, and only comparatively short text blocks have to be programmed.
All Minifactory component and product descriptions are objects that are represented by the Description Class (*Figure 6*).

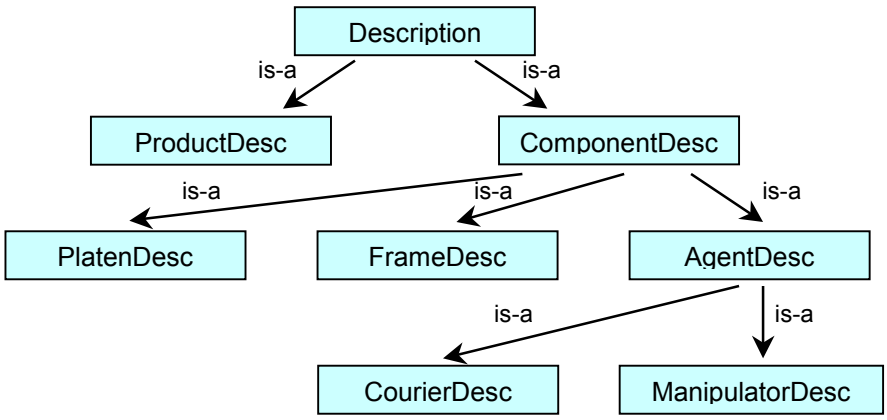


Figure 6: Class diagram for factory descriptions

In the product description (*ProductDesc*) subclass all products getting assembled in a virtual factory are defined. The descriptions of all factory components have to be subclasses of *ComponetDesc*. Static structure components, as base frames and platens, as well as agents are counted in these subclasses. Every agent description - courier and manipulator - has to be a subclass of *AgentDesc,* which contains a field named “interface” (Gowdy 1999).
The interface field itself is a database that encapsulates the actual implementation of both simulated agents running in the Interface Tool and physically initiated agents the Interface Tool is interacting with remotely. State variables, which can be monitored, and parameters which can be changed to influence the operation of simulated or physical agents, can be entered into this database. The value of the interface field must be a subclass of the *Interface* class (*Figure 7*).

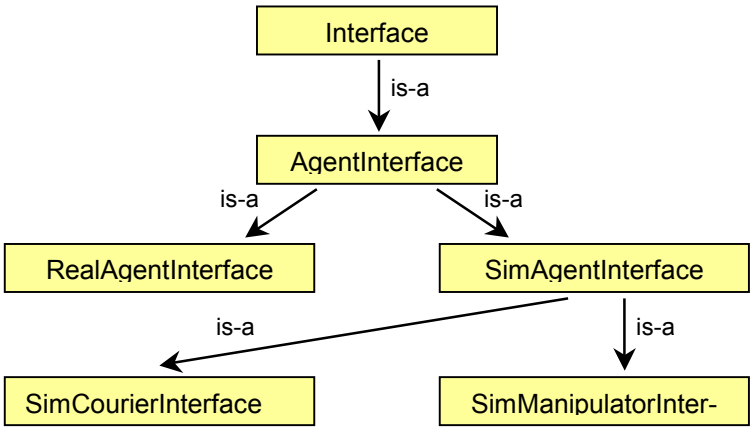


Figure 7: Class diagram for agents Interfaces

These agent description interfaces must implement an update method, which is called by the Interface Tool as often as possible. To move the rendered parts of the agents' description, as the graphic representation of a courier sliding on a platen, a particular implementation of the update method is necessary. The structure of the Interface Tool C++ programs is highly nested. This fact is attributed on the one hand to so many derived subclasses, on the other hand to the numerous use of macros (separately defined variables or small functions).

The object oriented structure of Interface Tool accommodates the modular character of Minifactory. Each factory component is a self-contained class/subclass that provides particular methods to describe its appearance. For instance, all objects offer a method that defines with which parts and how the component is assembled. Thus, the GUI can assure the correct assembly of the virtual factories by implementing these methods.

Currently, the user can choose between two different environments within the Interface Tool. On the one hand, there is the *tool* environment, which is used to provide the physical interaction with the agents as described above. On the other, hand Interface Tool offers the *sim* environment, which can solely be used to design and program virtual factories.

The GUI of the two environments are similarly designed and offer the same capabilities except for a dialog box for making pictures and recording movies of the simulated Minifactory, which is only a feature of the *sim* environment.

Both environments have access to the same information about the description of factory components and products. A point the *sim* and *tool* environment differ in is the coding of the run behavior of the assembly system in Python. Though the structure of the Python script is identical, the commands for the agents are different. As the virtual agents do not have to care about how to control the motors and valves exactly, the commands are simplified in the *sim* environment. For instance, a simulated overhead manipulator with a vacuum gripper picks a part by moving its virtual end effector until it reaches the part's surface in order to bind it virtually to itself without controlling the position of the effector or the real vacuum flow. That is the reason for the partition in two separate systems. The programming in the *sim* environment is more convenient and faster, because the commands are easier to handle. Therefore, the *sim* part of the Interface Tool is the ideal medium for users who only need a virtual version of their planned factory. The direct translation from a *sim*- to a *tool*-factory is not possible.

The dispartment is not part of the final solution of the Interface Tool. Since the representation of almost every agent existed in the virtual reality long before the agent is actually built in reality, *sim* environment offers the opportunity to include agents in a factory simulation without knowing the exact commands needed for the control of the physical machine.

As long as Interface Tool and Minifactory are still in the state of development, it makes no sense to unify both environments.

2.3.2 Designing and Programming a Virtual Minifactory

Since this thesis deals mainly with the creation of virtual factories, the following paragraph describes primarily the standard procedure for setting up a virtual Minifactory with the Interface Tool *sim* environment.

First, the factory designer opens a window with the graphical user interface of Interface Tool and assembles a factory from modules (see *Figure 4*). The required modules and parts can be chosen from the component palette or can be downloaded via WWW, the latter option will be of importance when Minifactory is marketable and a network of manufactures offer agents. In contrast to

basic modules, like base units and platens, which can be used universally for every kind of factory, agent modules have to be modified for each task in most instances. In many cases it is enough to equip a mounting on the courier, which carries products and/or the tools and to adapt specialized end effectors to standard manipulators. Component part retainers as well as end effectors are custom-made factory component. Often there are CAD (Computer Aided Design) construction plans that can be used to create virtual representations of these components in the Interface Tool environment. Otherwise, CAD files of the required parts have to be designed. When designing the parts, the level of abstraction is allowed to be relatively high. This means that only the surface and not the inner life of the parts, and only essential details have to be reproduced. The same proceeding also applies to component parts and products, which get assembled in the virtual Minifactory.
All types of CAD files have to be converted into Inventor files (.iv) so that they can be displayed by the GUI user interface (*Figure 8*).

Figure 8: ZYVEX mechanism in Pro/E and in Open Inventor format

After exporting the iv-files from the CAD program to the Interface Tool, they have to be adjusted to the simulated environment. These adjustments can be changes of the unit (inch, mm), of the transformation matrixes or of the material. The modified iv-files are then included in the Python files, the only file format Minifactory can work with. The in Python files encapsulated iv-files affect then the appearance of the used courier or manipulator.

As soon as all required components of the virtual factory are provided, the factory designer links the parts together by marking two parts with the cursor at a time and activating the assemble command. All possible connection locations of the components are hard-coded in their C++ files, so that invalid connections cannot be made. Since there are several available locations, the components have to be positioned and orientated to guarantee the desired assembly. The alignment of the modules can be realized by changing the cursor mode from marking to moving.

There are two alternatives to integrate the component parts of the future products into the simulated Minifactories. In case the parts are to be placed already in the retainers when the simulation is called, their iv-files have to be included later in couriers' Python files. If the parts are supposed to appear during the assembly simulation, their iv-files have to be included in the main program - also a Python file - of the Minifactory. Latter version is used for instance if adhesive application is to be simulated during join proceeding.

After implementing the described scheme, one obtains a static 3D image of the planned Minifactory, which can already be used to make pictures.

Up to this point the proceeding is in both environments, the *sim*-environment and the *tool*-environment, absolutely similar.

In the next step the behavior of the agent modules is programmed. In order to do that, the just generated factory (.fac) file has to be opened in any compiler. As previously mentioned, by designing a Minifactory with the GUI, the basic structure of the program is already built automatically.

```
file base_frame.aaa {
    children {
        file lg_platen.aaa {

            matrix [ 1 0 0 0 0 1 0 0 0 0 1 0 0 0 570 1 ]
        }
        file Osti_courier_1.aaa {
            matrix [ 1 0 0 0 0 1 0 0 0 0 1 0 450 -330 655 1 ]

            member home {
                matrix [ 1 0 0 0 0 1 0 0 0 0 1 0 -450 330 15 1 ]
            }
```

Source code 1: Code extract of OSTI fac-file

The structure is characterized by parent-child-relations, as a base frame is the parent of the platen and of all couriers attached to it, or as an overhead manipulator is the child of a bridge (*Source code 1*).
Each child exhibits as a member field a matrix that defines its position relative to the zero point of the global coordinate system.
The actual code for the assembly movements is included under the member fields of the agent children. The included move commands can been seen in the complete OSTI factory file in appendix: A OSTI Files. After saving the upgraded fac-file, the factory designer opens it with the GUI again. Now the simulation can be played, analyzed and tested for failures. As there is no built-in collision detection control, there is no automatic prevention of agents colliding with basic modules or other robot agents. Thus in the run-mode the user has to monitor the assembly system closely not only to scrutinize the sequence of the assembly but also to detect collisions. The program must be modified until the virtual Minifactory works properly. Of course the testing phase is also an essential part of the factory building procedure in the *tool environment*. There it is even of higher importance because it can prevent damage of the physical factory.

2.4 Setting Up a Working System

Before the start of this thesis, hardware and software of Minifactory were modified or upgraded. The computer hardware of the physical agents (courier and standard overhead manipulators) were changed to embed Intel Celeron M systems and the operating system was switched from LynxOs to QNX. Furthermore, a new end effector system with a novel optical system has been designed and equipped.
The Interface Tool was moved and ported from a Silicon Graphics machine to a new X86 PC computer running on Linux. This operation however could not be accomplished completely in the planned time and thus some of the modifications were not finished by the start of this project. For this reason, the adaption of the system was integrated into this thesis as a first part. Since this the-

sis deals with simulated factories, problems concerning the Interface Tool are presented in the following paragraph. After the transfer of the Interface Tool to the Linux environment, comprehensive testing of the system was not carried out in order to save time for the main part of the thesis.

Occurring problems throughout the project due to the missing testing especially with functions that were not needed in the beginning, were solved on the fly. At first, a problem with time variables, which are responsible for the workflow of the simulated Minifactory, had to be solved. Thereafter, it appeared that all simulations ran very slowly and jerkily. This was caused by an incorrect installation of the graphics card, whose hardware first was not used for the 3D rendering of any Linux applications. Also, the GUI did not work correctly in the beginning, as the factory components could not be connected in the intended places.

Most of the ordinary problems were caused by missing libraries or wrong default paths, a result of the previous computer system change. At least almost the entire functionality of the Interface Tool could be restored and only unimportant features, like the option for shortcut control of the GUI, were not repaired.

Another difficulty was, that a significant part of the existing documentation, which refers exclusively to the Interface Tool on the Silicon Graphics machine, is not valid for the Interface Tool on the X86 PC system. Thus, in many cases the handling with Interface Tool had to be learned by trial and error.

3 OSTI Project: Assembly of Telescopic Sight

Optical System Technology, Inc. (OSTI), Kittanning, PA, USA, a subsidiary of OmniTech Partners Inc., is specialized in manufacturing night sights and other small arm weapon sights. Currently, the manufacture techniques for electro optic sub-assemblies at OSTI are often labor intensive and require specialized handling to prevent the damage of components. Furthermore, frequent in-process cleaning is essential for the later functional efficiency of the products. Most of the assembly is done manually and many tasks must be performed by specialized personnel. This skilled personnel spends a significant amount of its time to these lower-skill tasks during product assembly and tests. Thus, OSTI makes approaches to automate the assembly of optical systems. Since the assembly procedures, processes and components are subjected to a continuous improvement and adaptation process, and new products with similar characteristics but different components are taken up regularly in the product portfolio, not only a flexible but also an agile assembly system is required for this assembly task.

Minifactory is predestinated to produce these optical devices in an automated assembly line. In comparison with traditional automated assembly systems, Minifactory can meet these demands. It is part of the AAA philosophy to adapt to new tasks fast and easily. This ability allows the assembly of different product variations within the same product line. Thereby, assembly cost can be reduced and customized solutions can be embedded in the standard production line. As OSTI has many different sub-assemblies that could be built on the same automated platform, AAA is specially qualified for these assembly tasks. Minifactory also has the ability to perform the same operations on similar sub-assemblies. Another benefit is the high precision of Minifactory, which is required for many assembly steps at OSTI and which is inherent to the system of AAA. The simple adaptation of Minifactory to a clean room system is a reason for using AAA for OSTI applications.

Because of the combination of versatility and precision AAA offers, OSTI chose Minifactory to make an attempt to automate the assembly of optical systems.

3.1 Task Analysis

For a first assembly task, OSTI proposed the automated assembly of a collimator sub-assembly. This sub-assembly consists of a collimator housing, four lens elements, three lens retainer rings and a lens spacer (*Figure 9*).

The following three basic operation types are required for the assembly:

- Pick and place operation to grasp, transport and position the component parts
- Adhesive application to fix lens element #1
- Screwing processes to tighten the lens retainers

Since the glue dispensing agent was still in development and thus not applicable, only a dry assembly was attempted.

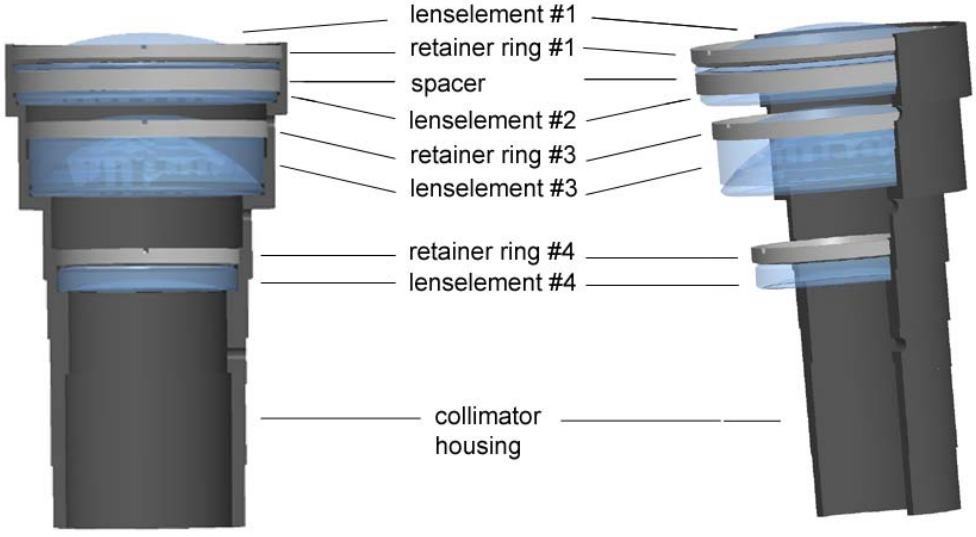


Figure 9: OSTI collimator

The OSTI project was partitioned into smaller work packages and handled by a team. The work packages are listed below:

1. Designing of an end effector and tools to pick and place parts and to screw the retainer rings.
2. Generating a virtual Minifactory simulation and visualizing the assembly process.
3. Programming the actual pick, place and screwing operation to be performed by the factory agents.
4. Joining the separate results back together and creating a working assembly.

Though the separate steps (1-4) depend on each other for the most part, the procedure is not strictly chronological. Thus, work packages could be partially settled parallel. The author of this thesis dealt predominantly with task two, whose work report is presented in the following sections.

3.2 Proceeding

The output of this work package is an operable virtual Minifactory that assembles collimator sub-assemblies. Pro/E (Pro/Engineer) files of the end effector, courier mountings, the tools and the collimator components as well as assembly information from OSTI were available as input information.

To generate the simulated assembly system, the task was handled in accordance with the following schedule:

1. Modifying the Pro/E files to adapt them to the requirements of the Interface Tool.
2. Transferring the part-files to the Interface Tool and transforming them to virtual objects.
3. Designing a virtual Minifactory.
4. Programming the virtual Minifactory.

3.2.1 Modification and Adaptation of the Pro/E Files

The team member who designed the parts provided the Pro/E files. The delivered folders include files containing the single components of the new end effector and of the two courier mountings as well as files of the tools and collimator parts. As all workshop drawings base on these 3D files, they show every detail. For the simulated factory, many of these details are unnecessary or even unwished. Some details are needless, because they are located inside the parts, as for example the lens system of the camera inside the new end effector, and are not displayed in the simulation – only the surface of the parts is shown. Other details, like screw holes in factory components, have no function in the virtual environment and are unnecessary. In principle, screws and other fixing devices are only displayed in the virtual Minifactory, if they are very characteristic or if they take part in a joining proceeding for the product assembly. Very small details – especially with micro assembly – cannot be displayed or only by excessive zooming. But the fact that the user cannot view them does not mean that they do not exist. To reduce the required computing power and to avoid a slowing of the simulation, it is advisable to keep only details that are essential for assembly proceeding or for understanding of the assembly system, because every additional edge means more rendering effort.

Thus the first step contains the abstraction of all parts. For the file editing, Pro/E Wildfire 2.0 was used. Each part was reviewed and checked for dispensable details. Depending on the nature of the details, there are two ways to remove them. If these items are ordinary elements of the part, as drill holes or rounded edges, they can be undone by deleting them in the tree diagram of the part. But if the details are plaited in geometrically dependencies or are elements of the inner life of the part, it is often less labor-intensive to redesign the parts. In this manner, parts with a complex surface structure or internal spaces were replaced by simpler geometrical bodies. Sometimes it was expedient to replace a sub-assembly, consisting of several complex component parts, by a single solid part.

In some cases the design of the parts had to be reengineered in the tool shop. To keep an accurate match between the physical and virtual assembly system,

these changes have to be adapted to the Pro/E files. Also, material and color of the virtual components have to be adjusted to the appearance of the physical parts.

After all reasonable abstractions have been made, the end effector and the two courier mountings, which contain the collimator components and the tools, are assembled and saved as Pro/E assembly files.

Before transferring the files to the Interface Tool, it should be checked if the point of origin of the parts and assemblies is located favorable to the later use in the Interface Tool. By choosing the point of origin cleverly, the later effort for transforming the parts' matrixes can be reduced considerably.

All Pro/E files have to be converted into iv-files, so that they can be displayed with the Open Inventor. The choice of the iv-files resolution must be well considered. The resolution defines the number of triangles the surface of the part's image is constructed of. A balance has to be found between too many triangles, which requires a higher computing power, and too few, which would cause a very square-edged image.

3.2.2 Transformation of the Parts Files to Virtual Objects

Since Pro/E does not work on a Linux system, the iv-files have to be transferred via network from a Windows machine to the computer the Interface Tool is installed on.

Once the iv-files have been sent to the Interface Tool, they were saved in a folder which has to be integrated at the right place in the path of Interface Tool, so that the program running the simulated factory can access them.

Before the iv-files can be used in the virtual environment, they have to be modified. Since Minifactory bases on the metric system, the unit of the iv-file has to be changed from inch to mm (*Source code 2*). Now the files can be opened with the Ivviewer, a program that can display the content of iv-files, and be checked for bugs and appearance. An iv-file offers under the category "Material" the option to change the colors and transparency of the part components. In this way, changes of color nuances can be made to optimize contrasts and consort tones without going back to Pro/E (*Source code 2*).

For instance, the transparency of the lens elements was increased to obtain a glass like appearance.

```
Separator {
    ShapeHints {
        hints    (SOLID | ORDERED | CONVEX)
    }
    Units {
            units MILLIMETERS
    }
    Transform {
         rotation 1 0 0 1.57079
         translation 0 0 6.20557
    }
    Material {
           diffuseColor    0.548947 0.744058 1
           transparency 0.5
    }
```

Source code 2: code extract from iv-file of lens element #1

The iv-files cannot be integrated directly into a virtual factory but have to be embedded into Python files, which are called AAA-files (.aaa) (*Figure 10*).

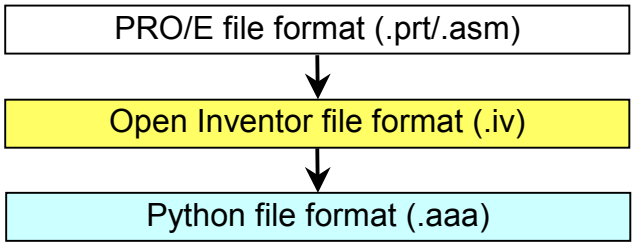


Figure 10: Conversion of part files

Depending on the part kind, there are three different objects to build an aaa-file from the iv-data.

- Iv-files of component parts of products (except courier fixtures) and demountable tools
- Iv-files of courier attachment
- Iv-files of end effector parts

In this project the collimator, the lens elements, retainers, spacer and tools are among the first category. The iv-files of these parts are built in aaa-files as follows:

```
Description {
    view InventorView { body { File {name element_1.iv } } }
}
```

Source code 3: aaa-file of the lens element #1

The integration of the other two categories is done in the course of creation of the new factory components and presented in the following section.

3.2.3 Designing a Virtual Minifactory

In this project, the nontypical case is present that the physical factory has already been set up. Thus, the virtual factory is modeled on the real factory. However, this initial position does not forbid categorically any changes in the real factory's setup.

The physical Minifactory works with a single unit consisting of:

- 1x base frame + base unit
- 1x platen tile
- 1x bridge
- 6x curb elements
- 1x standard overhead manipulator with customized end effector
- 2x standard couriers modules with customized attachments

Except for the robot agent, the virtual Minifactory can be built from standard Minifactory modules which can be chosen from the component palette. Standard manipulator and standard courier can provide a base for the agents in the OSTI factory.

3.2.3.1 Creating the OSTI Manipulator

In most instances, standard manipulators have to be adapted to the applications. The already existing manipulators can only be used for assembly tasks which can be managed with standard tools, as standard vacuum grippers. Otherwise, the manipulator's end effector or the assembly tool equipped to the end effector has to be replaced - just as in the physical world. In the GUI of the Interface Tool manipulator, end effector and assembly tool are treated like one object and for every newly designed end effector/tool a new manipulator file must be added to the component palette. The files are interlaced in the following way:

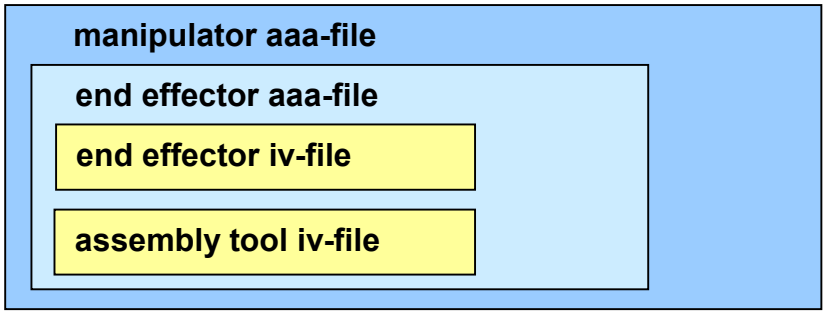


Figure 11: Nesting levels of manipulator files

The aaa-file of a manipulator contains the iv-files for its brain box and its arm as well as an aaa-file of the end effector, which is equipped to its arm. Furthermore it contains a gif (Graphic Interchange Format) file that is responsible for the thumbnail picture of the manipulator in the component palette, and the link to the ProgManipulatorInterface determining its behavior (see *Figure 7*).
The end effector aaa-file, which is included in the manipulator aaa-file, contains iv-files for its appearance and iv-files for the tool attached to it.
This structure allows an easy combination of different manipulators, end effectors and assembly tools and simplifies the creation proceeding of appropriate manipulators. At best, only the tool iv-files and the manipulator gif-file have to be replaced to obtain a new agent.
For the OSTI project, not only a new assembly tool but also a new end effector type (*Figure 12*) has to be attached to a manipulator (see *Figure 11*).

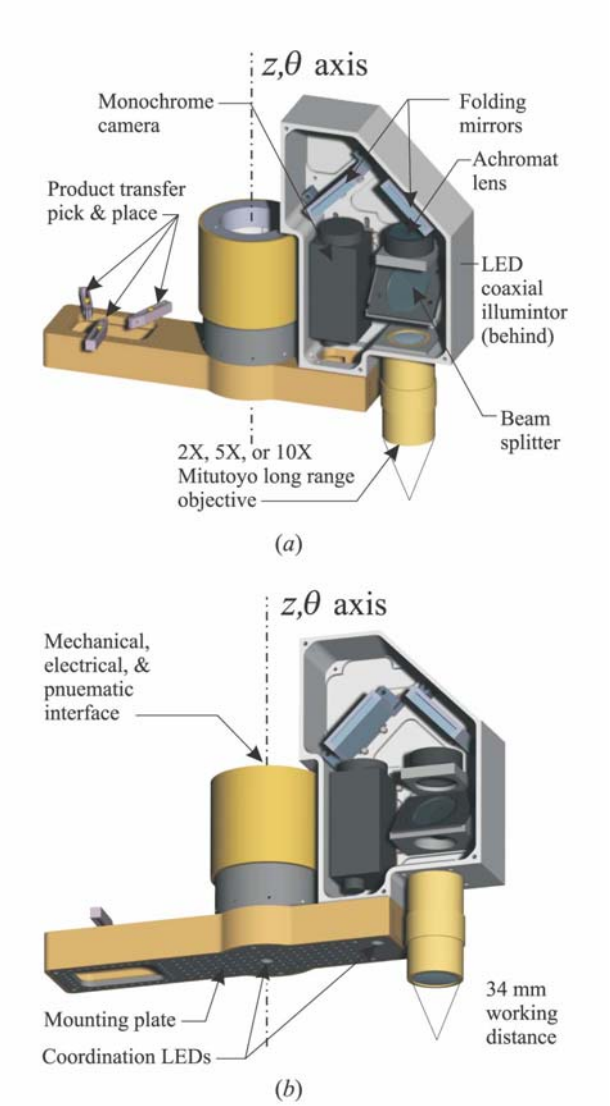


Figure 12: New end effector with attached camera

The new end effector was not developed solely for the OSTI project, but presents the new standard end effector for all future applications, and the newly designed tools bank on it.

In the first step a new end effector aaa-file, which contains the iv-file of the new end effector, is coded (*Source code 4*).

Then, this newly created aaa-file is included in the source code of a manipulator (*Source code 5*). The standard manipulator with a common brain box and arm is chosen for this purpose. This new manipulator file is also added to the component palette so that the result can be viewed in the virtual factory environment (aaa-files cannot be displayed by the Ivviewer).

```
Effector {
    view InventorView {
        body {
            File {
                name neweffector.iv
            }
        }
    }
    grippers (
              Link {
                  num <Int> 2
                  matrix [1, 0, 0, 0,
                          0, 1, 0, 0,
                          0, 0, 1, 0,
                          0, 0, -71.801, 1]

                  mount { { 0, 0, -1 }, { 0, 0, 0} }
              }
              )
}
```

Source code 4: OSTI end effector aaa-file

```
file common_manip.aaa {

    image file Osti_manip.gif

    effector file Osti_endeffector.aaa {}

    interface ProgManipulatorInterface {
    }
}
```

Source code 5: OSTI manipulator aaa-file

In the physical Minifactory, the position of the end effector relative to the end of the manipulator's arm bar is defined by the connection interface (*Figure 3*). In the virtual version, the points of origin of the two components determine the location of the end effector. By integrating the end effector iv-file into the manipulator aaa-file, the two points of origin are congruent by default. To check the positions of the two components, a test factory file, only consisting of one of the new manipulators, can be started. Almost in every case the position of the end effector has to be adjusted. The two problems that occur most frequently are on the one hand a twisted and on the other hand a displaced end effector. Often, there is a combination of both problems. The first problem can be explained by the fact that only the points of origin but not the corresponding planes (X-plane and X-plane etc.) are congruent. This can be solved by rotating the end effector's coordinate system in the iv-file (like in *Source code 2*).
The displacement is caused by a disadvantageous location of the end effector's point of origin. A sphere with the center coordinates (0,0,0) can be added

to the iv-files to uncover the points of origin of the parts. To view the spheres, the visualization settings of the GUI can be change from solid mode to line mode, which displays solely the edges of the triangles and allows a look inside the bodies (*Figure 13*).

By adding a transformation in the iv-file of the end effector, the current origin can be translated to the desired point of origin. Since there is no such thing like a connector that defines the clearance between manipulator bar and the end effector flange, the clearance has to be coded manually. To get the numerical number of the offset between both parts, measuring at the physical factory and metering in the accordant Pro/E files is required. After attaching the new end effector to a standard manipulator agent, the assembly tool for the OSTI task is linked to the newly created manipulator. The orientation and position of the tool is adjusted to the end effector in the same way the end effector's orientation and position is adapted to the position to the manipulator's arm. After that, the direction of the tool (downwards= 0,0,-1, sidewards=+/-1,0,0 or 0,+/-1,0) and the coordinates of the tool tip relative to its origin have to be specified. In a final step, a picture of the new manipulator agent is taken in the environment of the GUI with the built-in picture function. This picture is then converted into gif-format and integrated in the manipulator aaa-file.

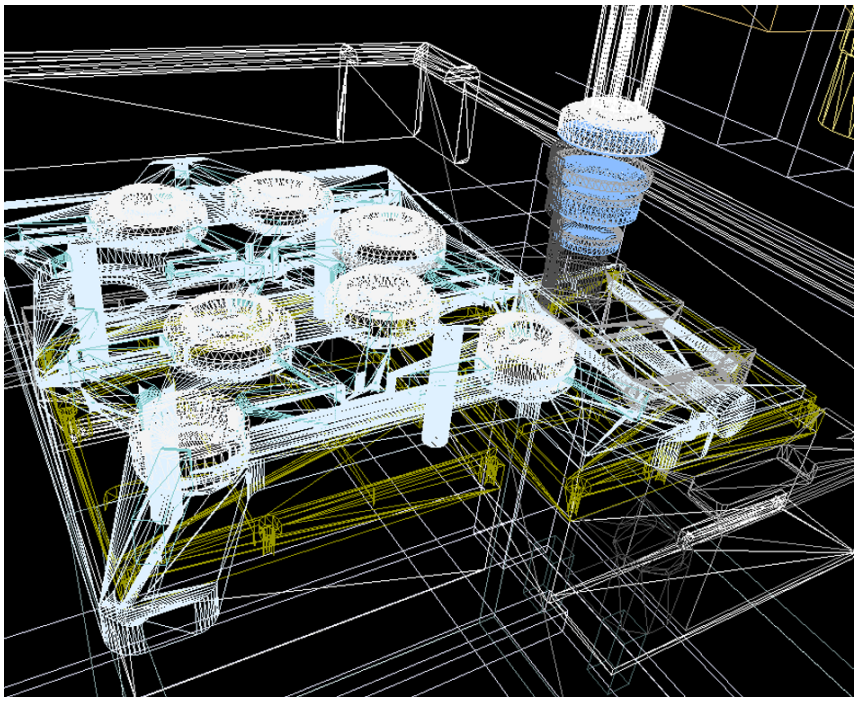


Figure 13: Line mode of the GUI

3.2.3.2 Creating the OSTI Couriers

The virtual courier agents provided by the Interface Tool consist of a brain box, the courier cube and the connecting tether. The courier cubes are bare and can be equipped with mountings for retaining product parts as well as changeable tools. Similar to the manipulator-end effector-relation, a new courier has to be created whenever a new mounting is to be attached. Similar to the manipulator aaa-files, a courier file contains in addition to the gif-file and iv-files for its components, as brain box and cube, iv-files with information about the attachments.

The afore edited iv-files of the retainers are included each in one new courier aaa-file. These retainers also have to be aligned by including rotation and translation operations into their iv-files. To guarantee a high accordance between virtual and physical Minifactory, the accurate values of mounting positions have to be measured at the real factory. At assembly, tasks with low tolerances, even a few mm difference, can cause problems as soon as the program of the virtual factory is uploaded to the real factory.

The simulation schedule of the OSTI project stipulates that collimator parts do not appear on the couriers until the simulation is started. In doing so, the manual filling of the retainers is simulated. Thus, the iv-files of the collimator components are not element of the courier aaa-files but are integrated in the main program file. The spaces where the product parts are fixed, however, have to be marked in the courier aaa-files. These marks, named ‘fixtures’, contain the coordinates (X,Y) of the empty spaces and an arbitrary name. All create, pick and place commands refer later to these names. The fixture’s coordinates define the place where the component’s point of origin is located and are relative to the courier’s point of origin. The simplest way to obtain the coordinate values is measuring the Pro/E models of the courier mountings.

As in the case of the OSTI manipulator, a pictures was taken of each new courier to receive a gif-file for the courier aaa-files.

3.2.3.3 Setting Up the Virtual Minifactory

After creating the OSTI manipulator and the two OSTI courier agents, the factory can be assembled. The other required factory modules are listed at the beginning of section 3 and were already available as standard parts.
In the GUI, the base frame and the platen are connected and a bridge is attached in the center of the base frame. The two OSTI-courier brain boxes are clamped on the left and on the right side of the base frame and the OSTI-manipulator is attached in the center of the bridge (*Figure 14*).

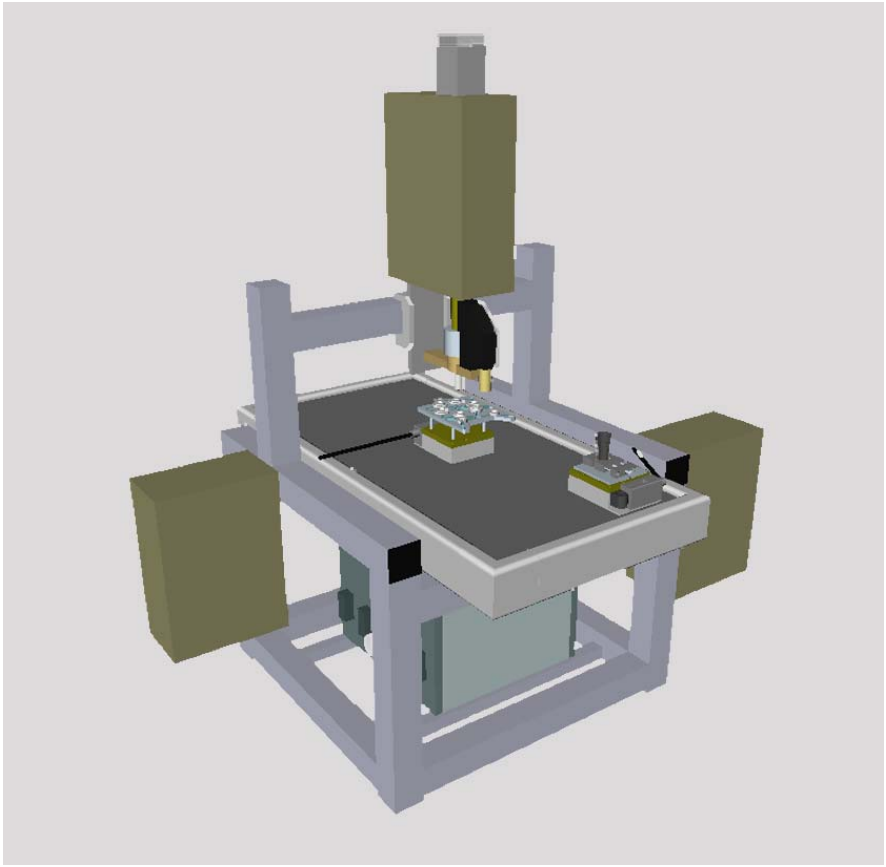


Figure 14: Set up of the OSTI Minifactory

After the assembly, the factory components can be moved in the environment of the GUI in the same DOFs like the components of the real factory can be moved for adjustment purpose. In this way, a rough alignment is possible. An accurate positioning can only be achieved by entering the numerical values, which were measured at the real factory, in the component's position matrix in the factory-file. To change the matrix that defines the 3D position of the factory components in the virtual environment, the OSTI factory has to be saved and

the GUI closed. Then, the OSTI-file has to be opened with the compiler program to view and modify the matrixes in the source code.

3.2.4 Programming the Virtual Factory

The fac-file created in step 3 is the basis for the further procedure. This fac-file can be generated with the GUI of both Interface Tool environments (*sim* and *tool*). However, the use of the *sim*-version is recommended, because it offers a more convenient way to make the thumbnails required for the component palette. As long as the factory file contains no program lines for the assembly processing, it can be opened and compiled in both environments. In this project, it was decided to start with the creation of a simulation in the *sim* environment.

3.2.4.1 Factory Concept

The factory set up in this project is a test system and is supposed to prove the qualification of AAA for OSTI applications. This factory was not designed for serial production, but to demonstrate the high precision and velocity of the Minifactory. For this reason and for a fast realization of the demands, a factory with only three robotic agents is conceived. Here is the assignments of tasks: The manipulator is equipped with a vacuum tool, which can grasp different pick up tools, and performs all assembly operations (*Figure 15*).

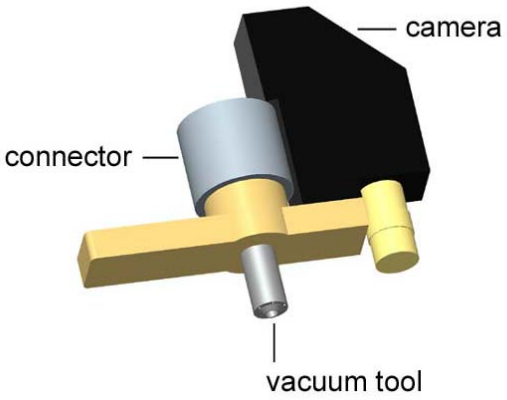
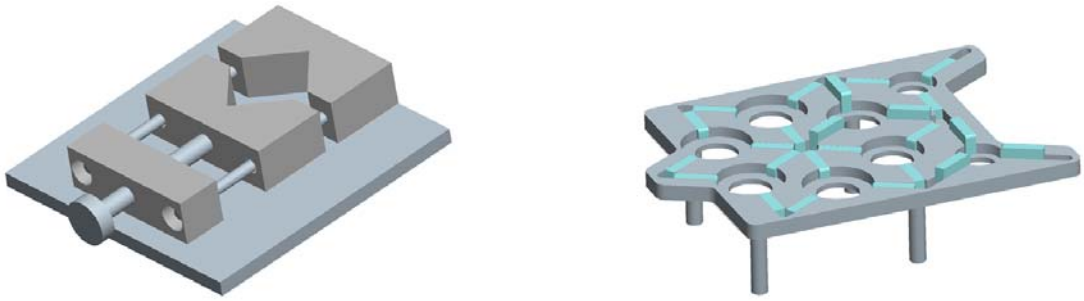


Figure 15: New end effector with attached OSTI vacuum tool

The first courier agent carries the collimator housing. *Figure 16* a) shows the retainer for the collimator housing. To the second courier, a tray is attached which contains the other product parts and the tools (*Figure 16* b).



a) b)

Figure 16: Courier mountings

The tray provides an own fixture for each part. Since this concept does not provide a separate manipulator agent for each product element but a single manipulator with a tool change system, there must also be a depository for the different tools to be used with the different parts. Thus, the second courier also carries the pick up tools, each on top of the corresponding part.

In the physical Minifactory, the end effector grasps the pick up tools via vacuum. Beside the vacuum channel for holding the pick up tools, the vacuum tool equipped to the end effector features another vacuum channel that supplies the attached pick up tools with vacuum. The vacuum tool (*Figure 17* a) equipped to the end effector features beside the vacuum channel for holding the pick up tools another vacuum cannel that supplies the attached tools with a vacuum. Since the two vacuum channels can be controlled independently, it is possible to lift tool and part at once and to place the part alone.

All tools used for the OSTI project handle the product parts with low-pressure. The pick up tools for the retainer rings additionally comprise salient angles on two opposite sides that fit into the recess of the retainers. With these devices, the torque from the manipulator can be transmitted to the rings, so that they can be screwed into the threads of the collimator housing.

The described factory concept is appropriate to the assembly of a single collimator and has to be reloaded manually to assemble another collimator.

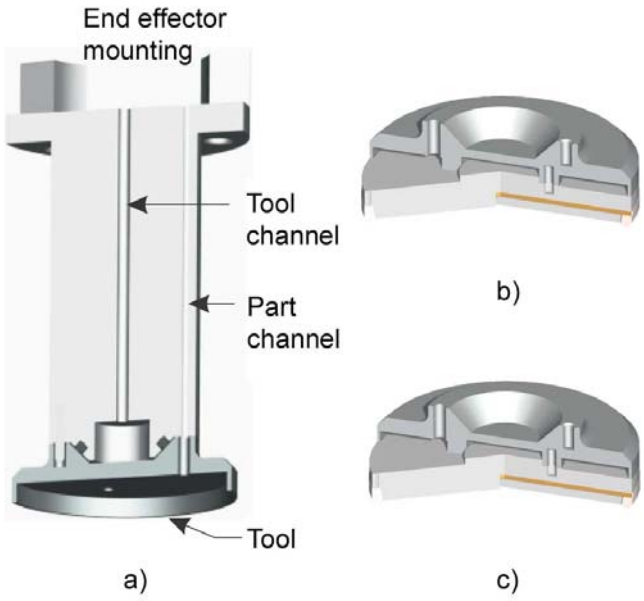


Figure 17: a) OSTI vacuum tool, b)+c) retainer ring tools (sections)

3.2.4.2 The Assembly Sequence

Currently, products like the collimator sub-assembly are assembled completely manually at OSTI. Since these kinds of optical products are very sensitive to pollution, many intermediate steps for cleaning are required. Assembled with an automated system, the component parts have no contact to potential pollution sources, as persons. Minifactory also was developed under the criterion of clean room application. Since the cleaning steps cease to apply, solely the actual assembly operations have to be considered.

In the first assembly step, lens element #4 (*Figure 9*) is inserted into the collimator housing. Thereon, the lens is fixed with retainer ring #4 which is screwed in a thread. After that, lens element #3 is placed and fixed with retainer ring #3 similarly – the pickup operation of this ring is displayed in *Figure 18*. In the next step, lens element #2 is inserted, followed by the spacer and lens element #1. In a final step the retainer ring #1 is screwed in.

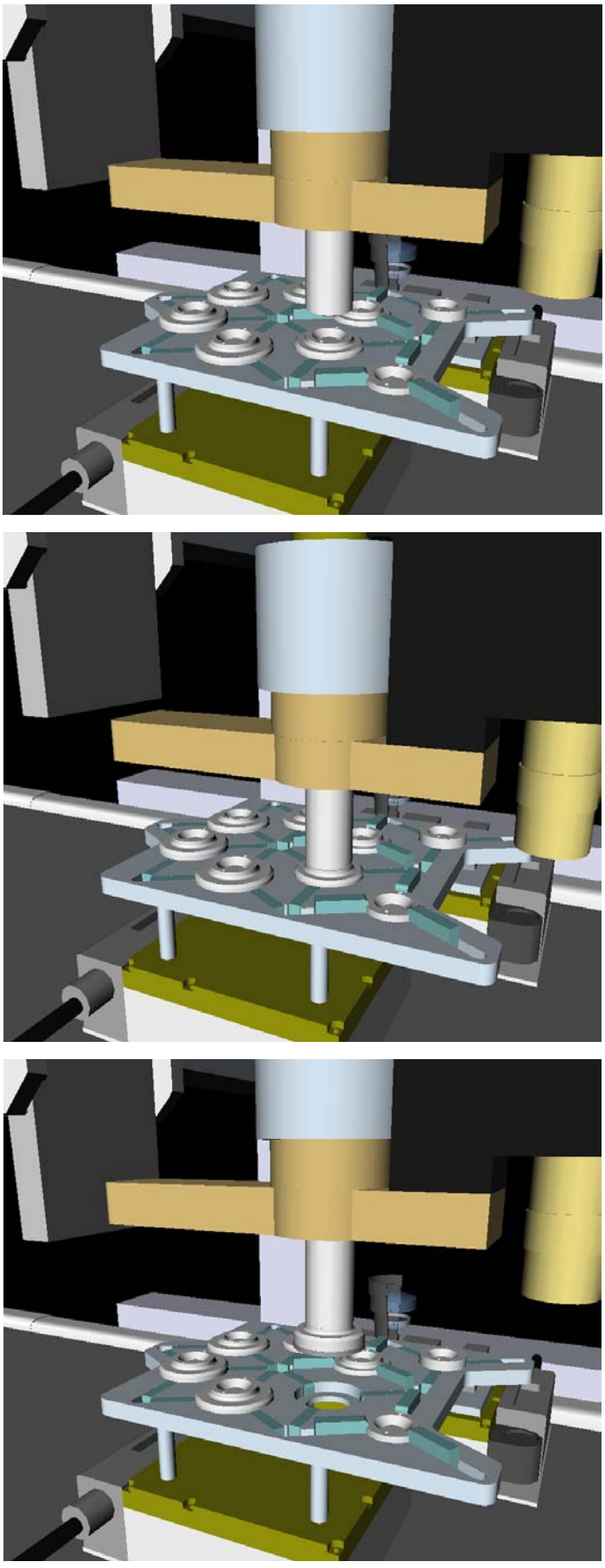


Figure 18: Pickup operation of the retainer ring #3

3.2.4.3 Writing the Program

Now, the assembly steps described in paragraph 3.2.4.2, have to be transformed into a program. The program lines are inserted in the OSTI fac-file under the matrixes of the robot agents (*Source code 1*). Program commands affecting the manipulator are added to the manipulator code, commands affecting the first courier added to the code of the first courier and so on. As already mentioned in chapter 2.1, the courier and the manipulator agents have to cooperate to accomplish operations requiring more than two DOFs. To simplify the programming of these co-operations, manipulators take over the control of the couriers they are cooperating with. Thus, every command referring to a cooperative operation is added to the manipulator code. By including the following commands in the courier code, the courier transfer the control to a manipulator agent:

`initiateRendevous(self, "MX", "tag")`: This command blocks the courier until the manipulator named “MX” is ready to accept a rendezvous named “tag”.
`reserve(self, "MX")`: This command appoints the manipulator named “MX” the control is transferred to.
`performRendevous(self)`: This command turns over the control to the manipulator.
`endRendevous(self)`: This command orders the courier to leave its clearance/working space and finishes the cooperation.
`unreserve(self, "MX")`: This command releases the courier from the control of the manipulator “MX”.

On the part of the involved manipulator, two commands have to be inserted. Between those two command lines the other move commands for the courier-manipulator cooperation are written:

`acceptRendevous(self, "tag")`: This command checks for pending rendezvous tags and unblocks the script, if a tag is found.
`finishRendevous(self)`: This command finishes the rendezvous and blocks the script until the courier has left the manipulator's working space.

In the present case, it is not possible to integrate all assembly steps in a single rendezvous as there are two couriers in alternate cooperation. Thus there are three kinds of rendezvous:

1. The rendezvous between the manipulator and the part and tool handling courier for picking up the component parts.
2. The rendezvous between the manipulator and the courier with the collimator housing for placing the component parts.
3. The rendezvous between the manipulator and the part and tool handling courier for placing the tool and - unless it is not the final assembly step - picking up the next component part.

Beside the presented commands, Interface Tool provides other commands that manage rendezvous. As these other commands are only necessary for the programming of factories with more than one manipulator, they are not mentioned here.
Before generating the program, the assembly operations (3.2.4.2) have to be divided into motion sequences. Below, this division is show by means of the assembly operation for fixing the retainer ring #4:

1. Moving second courier from home position to manipulator.
2. Picking up retainer tool #4 and retainer ring #4 at once and pulling back end effector.
3. Moving second courier aside.
4. Moving first courier from home position to manipulator.
5. Placing retainer ring #4 in collimator housing.
6. Screwing retainer ring #4.
7. Pulling back end effector.
8. Moving first courier aside.

9. Moving second courier to Manipulator.
10. Placing retainer tool #4 back into its fixture and pulling back end effector.

The other product components are assembled in a similar way, except that the lens elements and the spacer do not require a screwing sequence.

The first step of the OSTI simulation is filling the fixtures of both couriers with the collimator components. For this task the Interface Tool *sim* environment provides an own command. With `createPart(self, "part_name.aaa", "part_fixture_name", 0)`, iv-files which are embedded in aaa-files, can be displayed at any place in the virtual environment, which is marked by a "fixture". In this case, the "fixtures" are defined in the aaa-files of the couriers. After appearance of the product parts, the assembly operations are performed. The most important action command in this simulation program is `coordMoveTo(self, depth, x, y, z, th, speed, blooking)`. This command works only within a rendezvous cooperation and can control both the courier and the manipulator at the same time. The arguments X and Y determine the courier position, whereas Z defines the distance between end effector and platen, and theta (th) the angularity. Since the *sim* simulation does not display the behavior of the physical OSTI factory, the screwing operation was reduced to a simple rotation of the end effector without considering the change of the Z axis affected by the thread lead.
In the final step of the OSTI simulation, the two couriers move back to their home position, the first one carrying a completely assembled collimator, the second one carrying the pick-up tools. To make the simulation more demonstrative, it was decided to replace the original iv-file of the collimator housing with an iv-file containing the image of a cut in a halve collimator housing. The osti.fac file is displayed in appendix: A OSTI Files.

3.3 Further Development of the OSTI Project

In the next project step, the actual pick, place and screwing operations which are performed by the physical factory agents, are programmed. When pro-

gramming the screwing process of the real manipulator, the dependency between rotation of the end effector and the position of the Z axis caused by the thread lead has to be considered. For the picking and placing operations, the vacuum and compressed air supply of the tools have to be controlled precisely so that the collimator components can be grasped and released at the right moment in the program. The acceleration and retardation as well as the position detection of the robot agents have to be integrated into control instructions for the real factory, too. This project step was handled by the team members who were responsible for the physical factory agents.

In the final step, the results of the two previous steps were combined in a program that can be uploaded to the physical factory agents and displays a synchronously running virtual version of the real OSTI Minifactory. This program is written in the *tool* environment, since *sim* environment does not provide the ability to communicate with the physical machines. Like the *sim* program, it bases on the virtual factory as created in section 3.2.3.3. Instead of *sim* specific commands, *tool* commands are included in the agents' code. The *tool* and *sim* commands differ in many respects. There are also co-operations between courier and a manipulator agents in the *tool* environment, but here, the assignment of tasks is interchanged. The courier agent takes over the manipulator's control. In general, *tool* does not offer such a large number of ready-made commands, and the arguments of the provided commands bear more reference to the mechanical functionality of the physical agents. Thus in some cases, commands have to be developed for special tasks.

The complete *tool* OSTI program is displayed in appendix: A.2 Tool: osti.fac.

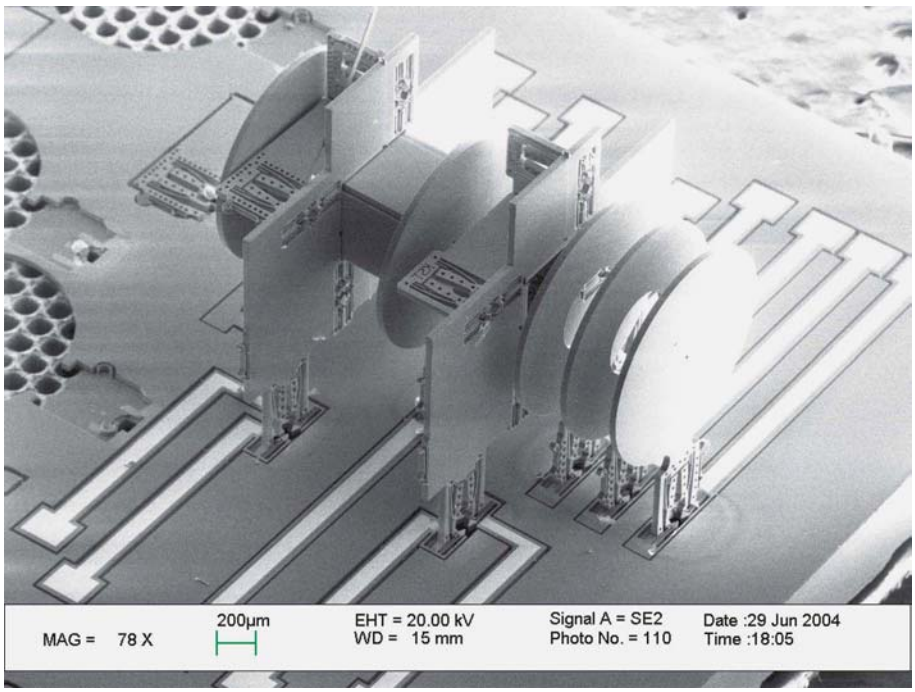
The last project step, described above, was managed by all team members.

At the end of the OSTI project, a virtual version of the physical OSTI factory that can perform the collimator assembly in a simulation, as well as the control program for the physical Minifactory were obtained.

4 ZYVEX Project: Precision Assembly of MEMS Parts

ZYVEX Corporation, Richardson, Texas, USA is a molecular nanotechnology company that supplies tools, products and services which enable adaptable and molecularly precise manufacturing (ZYVEX 2007). The customer base of ZYVEX includes branches of industry like Aerospace/Defense, Electronic/semiconductor and Medical/HealthCare industries.

A main focus of ZYVEX is microscale assembly of MEMS parts. ZYVEX develops software for commercial MEMS technologies, processes for manufacturing MEMS parts and assembly techniques for MEMS structures (*Figure 19*). The project MLS handled in cooperation with ZYVEX deals with the assembly of these MEMS structures.



ZYVEX Corporation

Figure 19: Example of a 3D MEMS structure

4.1 Definition of MEMS

Micro-Electro-Mechanical System (MEMS) stand for the integration of mechanical elements, sensors, actuators and electronics on a common silicon substrate through microfabrication technology (Memscap 2007). The electronic

elements are fabricated using common integrated circuits (IC) process sequences. To create the micromechanical components, compatible “micromachinig” processes are used that selectively etch away parts of the silicon wafer or add new structural layers to form the mechanical and electromechanical devices (Memscap 2007). Thus, a complete system can be merged on a single chip. While the integrated microelectronic circuits connect and control the system elements, the sensors, mechanical elements and actuators enable a MEMS to interact with its environment. Information about thermal, mechanical, chemical, biological, magnetic and optical phenomena are registered by the sensors and afterwards processed by the electronic elements. Then, the output signals of the “mini controller” direct the actuators to respond by moving and controlling the environment. For instance, the actuator response can comprise positioning, pumping and regulating. The benefits of this approach are obvious. On smallest area an entire system can be placed that combines the computational ability of microelectronics with the precision and control capabilities of microsensors and microactuators. Thus, the possible designs and applications are numerous. Furthermore, the manufacturing costs are relatively low, because of the use of batch fabrication techniques (Memscap 2007). Application areas of MEMEs are, for example, Biotechnology (Polymerase Chain Reaction Microsystems), Communications (RF-MEMS technologies), Automotives (accelerometers for airbags) or Office Technologies (micronozzles in inkjet printers) (MEMS and Nanotechnologie Exchange 2007).

4.2 Three-dimensional Microassembled Systems

The ability to built 3D microassembled systems creates unique capabilities and potential opportunities within the already broad range of applications based on MEMS devices and may develop new application areas (Tsui 2004). The fabrication of true 3D structures cannot be accomplished by common micromachining techniques. For many commercial devices, monolithic fabrication may be the easiest and most profitable method of manufacturing 3D systems, but as

soon as dissimilar materials within the structure's elements are required, other manufacturing methods have to be used. Thus, some level of assembly is inevitable. There have been several approaches to assemble 3D microstructures in the past. For instance, hinged structures, plastic deformation assembly techniques, assembly using surface tension forces and fluidic self-assembly are employed to assemble these kinds of structures (Tsui 2004). The first three methods have the disadvantage that the parts to be assembled have to be in close proximity of where they were fabricated. The fluidic self assembly, which does not have this constraint, cannot guarantee a controllable orientation of the structure. To avoid these problems, ZYVEX developed a more general directed pick and place microassembly that allows exact positioning of structure elements that can be fabricated in several different processes.
Currently, the assembly of micro scale objects requires a high degree of time consuming manual labor. For that reason, the packaging and assembly of MEMS components often account for a significant part of the costs and reduce the overall yield (Ellis 2002). ZYVEX tries to tackle this cost problem by automating as many process steps of their microassembly method as possible.

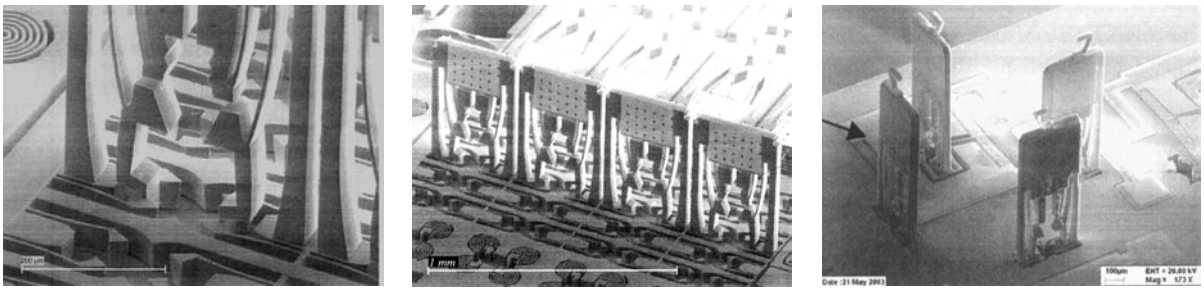
4.3 ZYVEX Assembly Method

The approach ZYVEX developed for microassembly of 3D MEMS is a combination of two ideas. On the one hand, the components of the MEMS have a special design that eases the join proceeding, on the other hand, ZYVEX developed a robot to perform the assembly.

4.3.1 Microcomponent Design

The ZYVEX microassembly principle bases on 3D MEMS structures to be constructed of microconectors (*Figure 20* a). That means that one of the join partners owns connectors, the other one a kind of a socket, and both together build a force/form-locked join. The part's connectors are inserted into the sockets and then fixed. Usually, several microconnectors are inserted into a

base that provides a socket for each element. The arrangement of the sockets define the layout of the MEMS structure (*Figure 20* b+c).



a) b) c) ZYVEX Corporation

Figure 20: a) Magnification of the connection, b)+c) 3D MEMS structures

In some cases, the microconnectors themselves feature sockets to hold other microconnectors. An important component of the ZYVEX assembly concept is the “zero insertion force” strategy (Ellis 2002). This strategy implies, that there is no force needed to insert the microconnector into the socket. Instead of snapping in place, the microconnectors are first put on the socket and then fixed in the next step. Thus, no force is required for the insertion and the risk of destroying components by applying too high forces or by incorrect insertion can be avoided. After putting the microconnector in the socket, the end effector that handled the part before spreads the legs of the microconnector which then snap into the locking notch of the socket. The design of the socket-connector couple provides self-centering and self-alignment capabilities. For this reason, the placing needs not be hundred per cent precise, neither needs the assembly system.

The assembly is described in all its particulars with an example in section 4.4.1.2.

4.3.2 Micro Assembly Robotic System

The current robotic assembly system used for the pick and place operation consist of a five DOF robotic system (Ellis 2002). The system is composed primarily of five Newport stages and is schematically shown in *Figure 21*. The assembly substrate is moved by three stages in X, Y and around the Z axis (Θ). The motion towards and away from the substrate is performed by a Z

stage. Linked to that Z stage is the ϕ stage that rotates about an axis around parallel to the XY plane of motion. Different micro grippers can be attached to the end effector arm that is placed to the ϕ stage. A precision of one micro meter is provided by close loop control systems for all five stages.

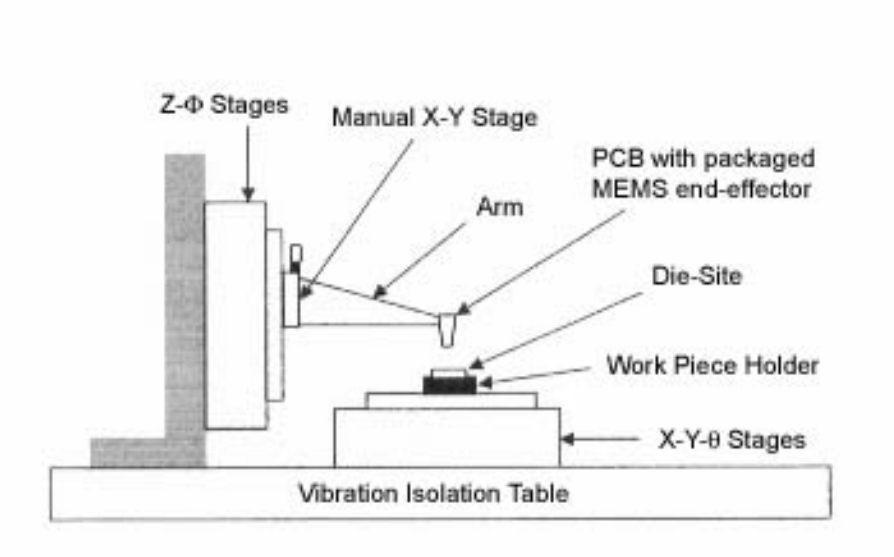
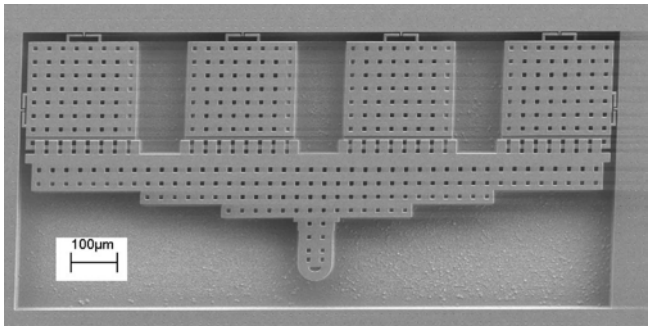


Figure 21: ZYVEX 5 DOF robotic system

For the grasping operation, ZYVEX embarks on two different strategies. On the one hand, ZYVEX developed active end effectors. These micro grippers allow to grasp micro components in a controlled manner. Different actuation principles including electrothermal, electrostatic and piezoelectric techniques are used to open, respectively close the opposing arms (Tsui 2004). The parts to be handled with micro grippers have to show points of contact where the gripper can take hold and where it can spread the legs to fix the parts. On the other hand, passive end effectors (*Figure 22*) were developed that do not feature moving parts and do not require separate control. In this case the matching of micro parts and end effectors is even more important than when using micro grippers. Since the passive end effectors are reduced to a simple inflexible tip, compliant elements have to be integrated into the micro parts. There are several advantages over the micro gripper approach. One benefit is the robustness of the passive end effectors. They have a low structural fragility

and can withstand higher forces than active tools. Micromanipulation using a passive end effector also eliminates dependence on actuator displacement and thus reduces control complexity. As their design is comparatively simple, there is enough space for force sensing systems. Embedded piezoelectric elements can be used to built force feedback loops. For these reasons a passive end effector was chosen for this project.



CMU

Figure 22: Passive end effector

4.4 ZYVEX-Minifactory Cooperation

Though ZYVEX has already developed a fully functional production system to fabricate 3D MEMS structures, there is an interest in alternative solutions that feature advantages over the existing system. Apart from the already mentioned benefits of AAA of chapter 2.2, Minifactory provides the required accuracy and the possibility of serial production. The existing system with the five DOF robots only allows serial production by paralleling many of these robot units. An assembly line cannot be set up with the ZYVEX system. Because of the very small range of the ZYVEX robots, sockets and the storage of the micro parts always have to be on the same wafer. With Minifactory, the wafers that contain the micro parts can be retained on one courier agent, the wafer on which the MEMS structures are built on another. This arrangement permits the flow-through manufacturing approach of Minifactory, as opposed to the more traditional work cell methods of microassembly (Hollis 2006). Since the ZYVEX robot cannot perform the assembly in an open loop system, calibration of this system is critical for automated assembly and necessary to overcome several cumulative sources of misalignment. Before the assembly can start, a manual

calibration of the workpiece, as well as a manual calibration of the end effector is required (Tsui 2003). The calibration process is time-consuming and causes additional costs. Minifactory, however, can perform the assembly, due to cleverly devised sensor system (including the camera mounted on the new end effector), in closed loop and can adapt automatically to small inaccuracies.

For these reasons, ZYVEX decided to examine to what extent Minifactory can meet the demands the production of 3D MEMS structures require. Accordingly, a virtual Minifactory which is able to process these 3D MEMS structures is created in the first step of this project.

4.4.1 Micro Parts for the ZYVEX Project

To demonstrate Minifactory's ability to build 3D MEMS structures, an assembly task was chosen that contains the general demands. The demonstration consists of the pick up operation of a typical microconnector that is then placed in a socket. A micro mirror that is an element of micro interferometers and of other optical micro devices serves as demonstration sample. In the following paragraph, all product parts are described.

4.4.1.1 End effector/Jammer

As already mentioned in section 4.3.2. a passive end effector is used for the Minifactory demonstration. The used end effector, also referred to as jammer, belongs to the new generation of passive end effectors. Contrary to the single crystal silicon jammers, this implementation (*Figure 22*) has built-in force sensors. It can detect out-of-plane forces with an integrated piezoresistive half-bridge in areas of high stress low stress concentrations.

4.4.1.2 Micro Mirror/Socket couple

The micro mirror (*Figure 23* a) belongs to the part family that is designed for the handling with passive edeffectors. Batch fabrication methods, which are

common in the production of MEMS parts, are used to manufacture the micro mirrors. With these techniques, a large number of components can be produced simultaneously. Like all wafer scale fabrication techniques, well ordered arrays of components with known positions can be produced. The last point is especially important for automated assemblies. The mirrors and sockets are fabricated in a 50 µm thick single crystal silicon (SCS) deep reactive ion etched (DRIE) process on a six inch silicon-on-isolator (SOI) wafer (Tsui 2004). The production process includes a backside DRIE through the 600 µm handle. Nitride filled trenches in the single 50 µm thick structural layer provide in-plane electrical isolation (Tsui 2004). For electrical contacts, evaporated gold layer depositions are used. Since the components are assembled after fabrication, fully released micro mirrors are required. To keep the controlled orientation, which results from the fabrication process, as well as the easy availability for the jammer the parts are tethered to the substrate by a thin compliant flexure (*Figure 23* a). This flexure prevents the mirrors from floating away and maintain accurate component palletization (Tsui 2004). Before picking up a micro mirror, the tether that anchors it to the device layer has to be broken. Unlike with the use of active end effectors, where the tethers have to be broken manually and the parts then placed in the work piece holder, the robust jammer can also be used for a breaking operation as part in an automated assembly sequence. For breaking the compliant tether structure, the tip of the end effector is lowered into a void in the structural layer and translated in-plane till it contacts, deflects and breaks the tether.

The design of the micro mirrors was matched to the used passive end effector. The design of micro mirror can be roughly sectioned into two parts (*Figure 23* a). One part consists of a plain that is coated with a mirror surface and provides the functionality demanded from the 3D MEMS structure. The second part has the role to connect the micro part with the substrate. The connector part consists of three interleaving pairs of beams. The innermost pair, also referred to as compliant handles, is very flexible and allows the rounded end effector tip to move between. Like springs, the compliant handles press against the tip and thus the friction between handles and tip is high enough that the

micro mirror can be lifted by the end effector. The middle beam pair consists of two legs with two attached feeds. During the assembly, the feeds are inserted in the socket, and snap in its locking notches. After putting the micro mirror's feet into the socket the passive end effector keeps on pushing down until it leaves the compliant handle zone and enters the wedge-shaped part of the legs. By further lowering of the end effector, its tip has to pass the chamfers and spreads the legs at the same time. Only by spreading the legs, the feet can snap into the socket. The outer beam pair, also named anchor arms, aid in realizing the part from the end effector and ensures that the micro mirror is level upon insertion into the socket.

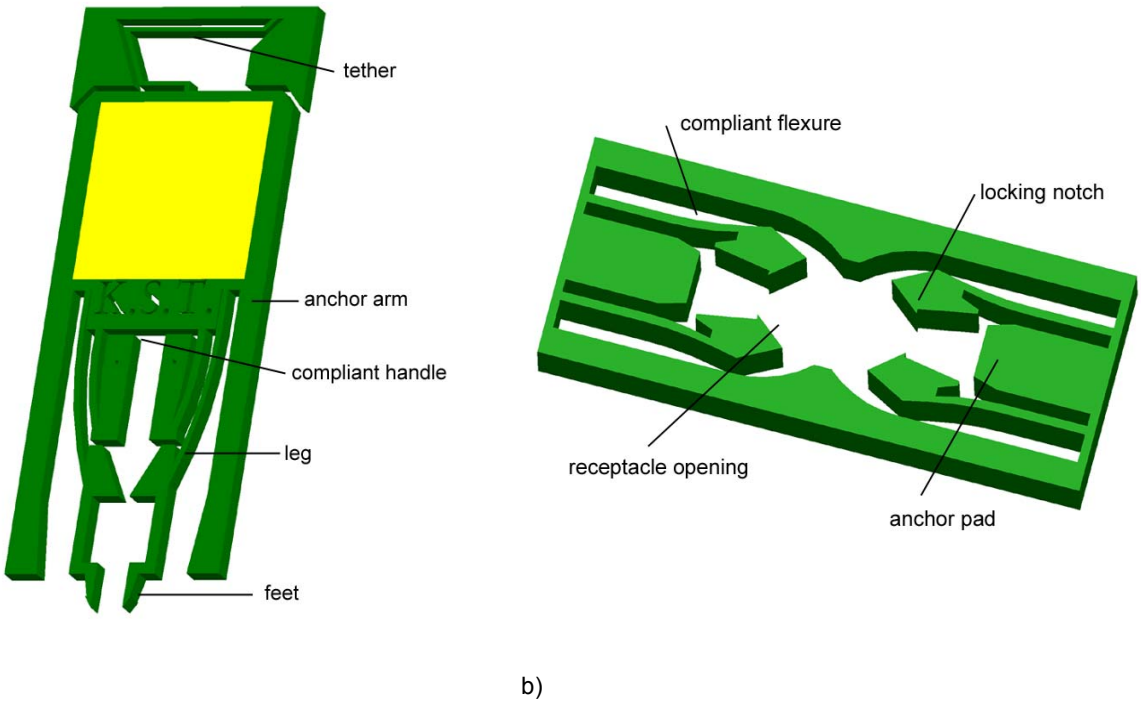


Figure 23: a) Tethered micro mirror, b) Socket

The socket (*Figure 23* b) is fabricated the same way like the micro mirror – since the ZYVEX micro assembly method only works if microconnectors and sockets are part of the same wafer. In the middle of the socket there is the receptacle opening, where the connector feet can be placed without resistance. On both sides of the opening there are each a pair of compliant flexures with locking notches. When the connectors' feet are spread, they pass the locking notches and snap in after the connectors' legs unbend.

The cooperation of the jammer, micro mirror and socket is displayed in *Figure 24*.

Figure 24: Assembly principle of the micro mirror

This connecting principle works for all micro components, handled with a jammer. For this reason, it is enough to use only one microconnection model to demonstrate the qualification of Minifactory for assembling MEMS structures.

4.4.2 Task Analysis

At the end of this project, an operating virtual Minifactory that assembles micro mirrors has to be set up. Since this project is confined to a virtual factory version to the time being, it is not accomplished by a team. The author of this thesis is in charge of project progression.

The ZYVEX project can be partitioned into the following work packages:

1. Designing of the MEMS parts and the jammer.
2. Modifying the Pro/E files to adapt them to the requirements of the Interface Tool.
3. Transferring the part files to the Interface Tool and transforming them to virtual objects.
4. Creating the ZYVEX manipulator.
5. Programming an additional axis for the manipulator.
6. Designing a virtual Minifactory
7. Programming the virtual Minifactory.

4.4.3 Designing of the MEMS Parts and the Jammer

Unfortunately, ZYVEX could not deliver the CAD files for the micro components in time. To avoid a delay in the project schedule, the CAD files were re-

designed in Pro/E. The Pro/E files are modeled on drawings in ZYVEX documents. By measuring the drawings, which had an attached scale, and using information about the components' size, the parts could be reconstructed. Of course, it had to be considered that the jammer has to fit to the micro mirror and the mirror to the socket. Thus, the parts' dimensions and proportions had to be adapted until the three parts matched perfectly up.

Since the dimensions of all used parts are within the µm/mm scale, a close observation of the assembly site during the simulation was unavoidable. By zooming so close, all details of the components will be visible. Thus, it was decided to take over all details, even the letters on the micro mirror.

After designing the single components, their wafer environment had to be created.

For the wafer no prototypes exist, because ZYVEX always places all different microconnectors, sockets and calibration elements on a single wafer. By studying several images of wafers that contain among other parts the micro components used in this project, the usual grouping of these parts were found. In the ZYVEX Minifactory, sockets and mirrors are stored on two different courier agents. This means that two different wafers have to be designed. The first wafer contains 32 sockets arranged in two 4x4 patterns, which are shifted by 90 degrees. This arrangement was chosen, because on the already existing ZYVEX wafers the sockets are often grouped in several lines of each four pieces. With the 90 degree shift, Minifactory can demonstrate that like with the ZYVEX assembly system the micro components can be placed in any angle. The second wafer contains the micro mirrors. In the center of the wafer, 100 mirrors are stored in a 10x10 pattern. Though the existing ZYVEX wafer features not so many micro mirrors, their arrangement was adapted to the new wafer. The mirrors are always placed multi- or at least double-row and in each second row the mirrors are turned 180 degrees. Like on the original wafers, the micro mirrors are tethered to the substrate.

To allow the grasping and placing operation in the simulation, the micro components have to be put into separate files. That means, the wafers cannot be designed as a whole but as templates with gaps for the micro mirrors and al-

ternatively for the sockets. Since after breaking the tether is not any longer a part of the mirror, the tether flexures are implemented as parts of the wafer frame.

To obtain a better contrast when the micro mirrors are plugged to the sockets, the two wafers were colored in two slightly different hues.

4.4.4 Modification and Adaptation of the Pro/E Files

Like in the OSTI project, the Pro/E files have to be modified so that they can be transferred to the Interface Tool. In this case, the component parts do not have to be abstracted to eliminate needless details, because during the design process decisions about details have already been made. The same applies to the choice of the place for the point of origin of the parts' coordinate systems.

Beside the micro components, there are other new Pro/E files. For the ZYVEX project, a new effector mechanism has been developed by MSL that is supposed to bear the passive end effector (*Figure 25*).

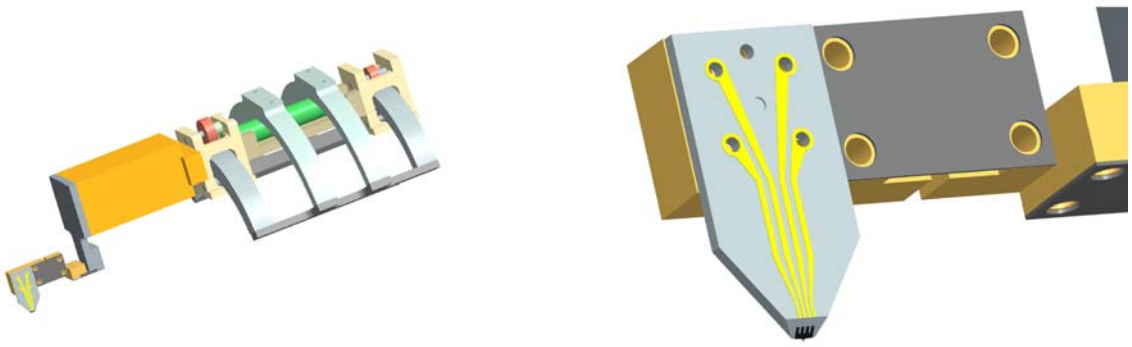


Figure 25: ZYVEX effector mechanism with attached jammer

It has already been designed and manufactured before Interface Tool was introduced into the project. As the physical effector mechanism differs from its Pro/E version, the Pro/E file has to be adapted to guarantee a hundred per cent matching. The differences occurred because of modifications of the real effector mechanism. Furthermore, many unnecessary details have to be eliminated to reduce the computing power for the rendering process during the

simulation. Since the components of the effector mechanism are relative complex and rangy, some elements are deleted completely, others were replaced by redesigned and simplified versions.

Also, the coordinate system's origin of the moving component of the effector mechanism has to be replaced. The new origin is placed where the tip of the passive end effector will be located, so that programming of the new rotating axis is simplified. In the last step of the effector mechanism modification process, the newly designed passive end effector is assembled to the effector mechanism.

After preparing the Pro/E files for the requirement of the Interface Tool, they are converted into iv-files. Also in this project, the choice of the iv-files resolution must be well considered to avoid the requirement of too high computing power on the one hand, and to obtain a nice appearance of the parts in the simulation on the other hand.

4.4.5 Transformation of the Parts Files to Virtual Objects

The first step after the file transfer is to change the unit in all iv-files from inch to mm, as Minifactory bases on the metric system. As already mentioned in section 4.4.3, the micro mirrors on the wafer can be orientated in two ways. The two versions only differ in a 90 degree rotation about the Z axis. To simplify the integration in the aaa-file of the courier, two different mirror iv-files are created by changing the rotation matrix of the original iv-file.

Like in the OSTI project, new courier aaa-files have to be created whenever new mountings are attached to standard couriers. Due to the lack of information about the design of the die holders, the wafers are placed directly on the courier cube surface. In this project, not only the iv-files of the wafers but also the socket and micro mirror iv-files are integrated in the courier aaa-file. Since the sockets and micro mirrors are inherent parts of the wafers, they should always be visible and not appear after the simulation starts. All iv-files have to be added under the fixture line in the courier aaa-file. By doing this, the iv-files are automatically attached as a mounting to the virtual courier in the simulation. The iv-file name, an arbitrary fixture name and the coordinates of its posi-

tion relative to the couriers' origin of each part have to be itemized. Unfortunately, the structure of the courier aaa-files does not allow to integrate loop functions in their fixture section. Thus, the information of each micro mirror respective to each socket has to be added manually to the files.

After generating two new couriers, they are added to the component palette so that their data can be retrieved in the GUI of Interface Tool. In the next step, the new couriers are included into the *sim* simulation environment and checked for orientation and position failures. These failures can be fixed by changing the position matrix of the iv-files of the used components. Finally pictures from the couriers are taken in the *sim* environment to obtain gif-files for the thumbnails in the component palette pop-up window.

The new effector mechansim, the passive end effector is equipped with, cannot be attached to a standard manipulator the same way like the vacuum tool of the OSTI project. The ZYVEX effector mechansim features a movable part that should be controllable by the Interface Tool. How the effector mechanism is integrated into the simulation is described in the following section.

4.4.6 Creating the ZYVEX Manipulator

To assemble the micro mirrors, they are removed from the substrate using the passive end effector, rotated 90°(*Figure 26*) and inserted into the proper receptacle. The result is a micro mirror attached perpendicular to the substrate.

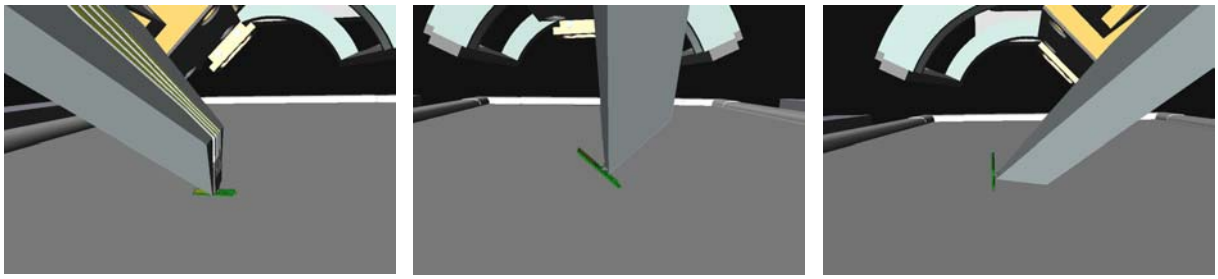


Figure 26: Rotation of the micro mirror with the ZYVEX effector mechanism

The existing Minifactory, both the physical and the virtual version, does not offer a way to rotate the mirror horizontally. While the existing DOFs include linear movements in the X-Y-plane (courier), up and down movements along the Z axis (manipulator) and rotatory movements about the Z axis (manipula-

tor), a horizontal rotation is not practicable. To handle the new assembly task, another axis has to be added. The most logical and also the least laborious way is to develop an attachment for the standard manipulator end effector that includes this additional axis.

4.4.6.1 Physical Version of the ZYVEX Manipulator

Since the connection interface between manipulator and end effector offers many controllable electrical links, it is no problem to equip a driven axis The physical version of the effector mechanism is pictured in *Figure 27*.

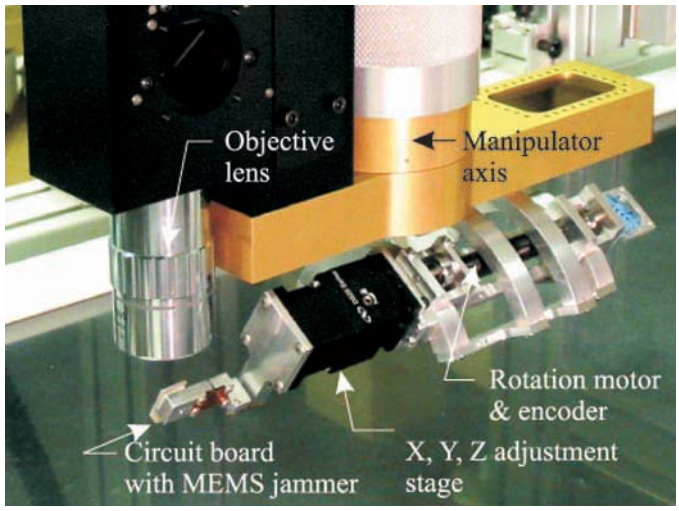


Figure 27: New end effector with attached effector mechanism

A friction capstan that is driven by a small gear motor with incremental decoder and is rolling on a precision arc-shaped guideway effects a controlled rotation of +/-47.5 degrees (Hollis 2006). The rotation resolution is approximately 0.09 degrees and the speed approximately 15°/s. To place the tip of the MEMS jammer near the center of the microscope's field of view, the rotating member part, also referred to as carriage, is equipped with a small manual three axis translation stage and small angular adjustment features (Hollis 2006).

4.4.6.2 Virtual Version of the ZYVEX Manipulator

For the planned ZYVEX simulation and as basis for future programming of the physical ZYVEX Minifactories, the effector mechanism has to be implemented

into Interface Tool. At first, a new manipulator has to be generated with an end effector that contains the ZYVEX effector mechanism.

Before going about the generation of the new manipulator, a lot of investigation had to be made. The data files of all already created virtual Minifactories and of their robot agents were inspected. Also the Python and C++ data files of all existing agents were examined and compared with each other. The result of the investigation can be summarized as follows. Since Interface Tool exists, no other movement axis had been added to a manipulator. Furthermore, the structure of Interface Tool offers no special options for the extension to additional axes/DOFs.

Basically, there are five different types of virtual robot agents. There is the group of overhead manipulators whose members are modified versions of the standard manipulator. The modifications usually consist of the attachment of different end effectors and tools. The same principle applies for the couriers. They only differ in their project specific mountings and sometimes in the courier's cube design. The other three agent types are purpose-built items and not as flexibly applicable as the standard agents. On the one hand, there is a laser welding agent with two laser cannons welding objects in their laser beam intersection points and a UV hardener, which hardens adhesives by radiation. Both agents do not feature an additional DOF. On the other hand, there is an elevator-courier-magazine combination. With this combination, courier can be refilled automatically with larger product components. The elevator courier moves under a magazine column and lifts its platform to the column whereon a product part is placed. This elevator courier is the only agent with three controllable DOFs. Its additional Z axis can act as a guide for the development of the horizontal axis of the ZYVEX effector mechanism. Since the layout of the couriers' data files is quite different from that concerning the manipulator agents and since in the case of the courier a linear and not an additional rotatory axis was implemented, the guidance is limited. At least this example could present a rough guideline for the implementation of the new axis.

The development of the new manipulator was an iterative process. According to the TOTE (Test Operate Test Exit) procedure, the written data is tested in the GUI of the Interface Tool, modified and tested again until the result meets the expectation.

In the first attempt, the arc-shaped guideway of the effector mechanism has already been assembled in the Pro/E environment to the new standard end effector with the built-in camera system. This means that after the conversion of the Pro/E file to the iv-format, the guideway is an inherent part of the end effector iv-file. Thereafter, the iv-file was embedded in an end effector aaa-file. The iv-file of the carriage of the effector mechanism was added to the gripper section in the identical aaa-file. This solution allows to pick up and place the micro mirrors, but after some testing it turned out that it offers no potential for creating a moving command for the horizontal axis. Although this attempt failed, the created ZYVEX manipulator can be used to obtain a thumbnail picture for the component palette pop-up window.

To understand how to create a virtual manipulator with a operating horizontal axis, the structure of the agents' data files has to be examined. All robot agents, regardless of which type, are implemented as described below.
Each agent is defined in a C++ file called Fo*Agent*Desc.cc. Together with its header file (*FoAgent*Desc.h) it determines of which parts the agent consists and how the agent is built up. It also defines the relation between the agent components. The Desc.h-file is included in the FoProg*Agent*Interface.cc file that defines the interface to a simulated agent which can be directed by a Python script.
Thus, it appears that for the ZYVEX effctor mechanism, a new agent type has to be created. This means that among other files, a new Desc-file has to be written. Since the new agent bases on the standard manipulator agent and only differs in an additional motional-axis, the new file consists of a subclass of the standard manipulator Desc-file that inherit the components of the standard manipulator and the effector mechanism's parts in addition. By subclassing,

the amount of work can be reduced and risk of failures can be limited. The relation between standard courier and elevator courier can be used as a source of information.

First, the header file of the Desc-file has to be written. It contains the class FoZYVEXManipulatorDesc, which is a subclass of FoManipulatorDesc from the standard manipulator header file (which again is a subclasses of the general agent's Desc-class). In this class, Descriptionfields of the effector mechanism's components and fields for the value margining of the carriage are defined. In the next step, the actual Desc-file is started. Here, content is assigned to the fields and member relations are determined, e.g. the guideway is a member of the end effector and the carriage a member of the guideway. Since this agent is destined for a particular project, a feature for changing the end effector or the tool is not necessary. That is why the iv-files of the effector mechanism are added directly to the manipulator aaa-file and not as usual to the end effector aaa-file. Accordingly, apart from the aaa-file of the new standard end effector, the ZYVEX manipulator aaa-file contains the iv-files of the end effector mechanism's components. To create the ZYVEX manipulator aaa-file, the aaa-file of the standard manipulator can serve as template. Besides an extension with the iv-files, only the interface link has to be changed from ProgManipulatorInterface to ProgZYVEXManipulatorInterface. The generation of the ZYVEX Interface-files are described in the following section. Even though the components of the effector mechanism are solely integrated in the manipulator aaa-file, information about the location of the tool tip (in this case the tip of the jammer) has to be specified in the end effector aaa-file, because the grasp and drop commands access the information about the tip location in this file. After coding the ZYVEX manipulator, the new aaa-file is added to the component palette. The FoZYVEXManipulatorDesc.h and FoZYVEXManipulatorDesc.cc files can be found in appendix: B.3 FoZYVEXManipulatorDesc.h, B.4 FoZYVEXManipulatorDesc.cc.

4.4.7 Programming an Additional Axis for the Manipulator

Writing the FoZYVEXManipulatorDesc files was the first step for implementing the fifth axis of freedom in the virtual Minifactory. So far, there is only a new virtual manipulator that exists but cannot operate in a simulation. Below, requirements are listed the manipulator's functionality has to fulfill:

- Up- and down-movements of the end effector and the tool along the Z axis
- Rotatory movements of end effector and tool about the Z axis
- Rotatory movements of the carriage of the effector mechanism about a horizontal axis
- Combination of the three movements
- Controllable velocity of all axes
- Grasping of parts
- Rotation of the grasped parts about the horizontal axis
- Dropping parts in every angle

The following diagram shows the files that have to be created to obtain an operating virtual ZYVEX manipulator. Furthermore, the connection between the files is displayed.

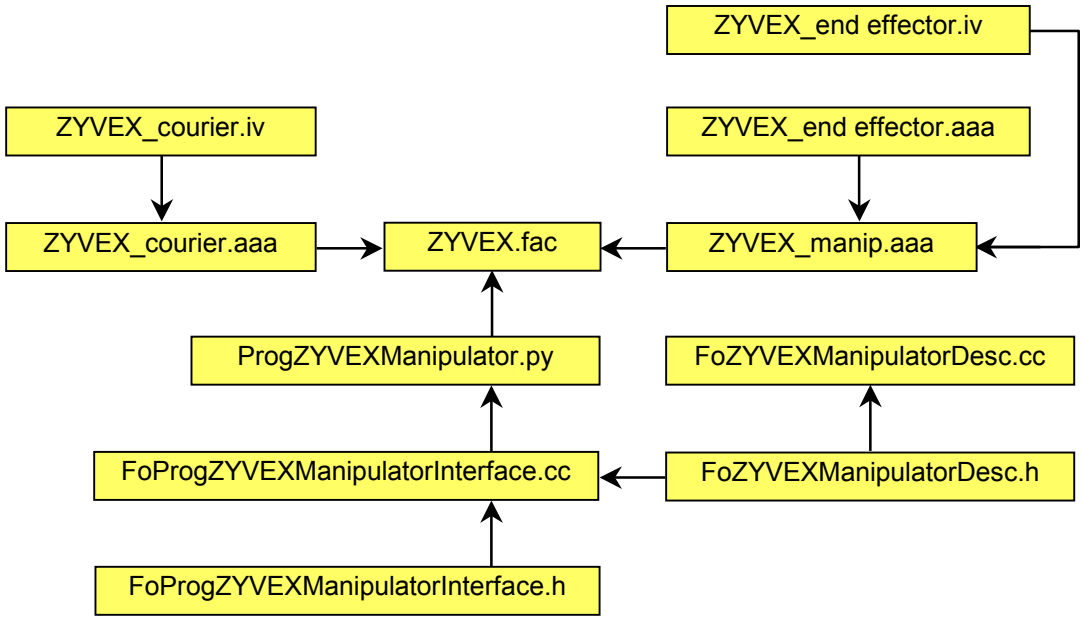


Figure 28: ZYVEX files schema

After creating the iv-, aaa- and Desc-files that determine the appearance of the new agent, the files have to be written that describe its behavior.
The all defining program is the ZYVEX Interface-file (e.g. FoProgZYVEXManipulatorInterface in *Figure 28*). In this file the methods are defined that realize the movement of the manipulator.
Since the ZYVEX manipulator differs from the standard manipulator only in the additional DOF, the Interface-file of the standard manipulator can provide the basis for the ZYVEX Interface-file. Although it would be possible to integrate the new functionality in the existing standard manipulator program, it was decided to create an extra file. By this it is easier to keep a well structured program layout. Beside the already existing abilities like moving, grasping and making rendezvous with a courier, the capability to move the carriage has to be part of the new program.
The best way to integrate the features of the standard manipulator into the new program is to form a subclass of the master class (FoProgManipulatorInterface) of the existing manipulator program. This subclass (FoProgZYVEXManipulatorInterface) automatically contains all variables and methods needed for the features the standard manipulator offers. The derived class presents the master class of the ZYVEX Interface program and has to be completed with member variables and member functions that are necessary to control the additional axis. In the following, the most important functions of the new class are described:

- `carriageGet`: This function recalls the current position of the simulated carriage.
- `moving`: This function checks if the simulated carriage is currently moving.
- `carriageSet`: This function sets the carriage to the next position.
- `start_carriage`: This function starts the trajectory.
- `moveCarriage`: This function moves the carriage to the arrival position at chosen velocity.

- `move_carriage`: This function provides the command for the Python programming environment.
- `update`: This function updates the ZYVEX manipulator interface and delivers/gets information to/from the carriageGet/carriageSet function.

Beside the main class, there is the class CarriagePointList that creates a FIFO (First In First Out) queue of points being tracked by the simulated carriage. This point queue is the basis for the master class's functions. *Figure 29* shows the components of the FoProgManipulatorInterface class and the element required for the added functionality.

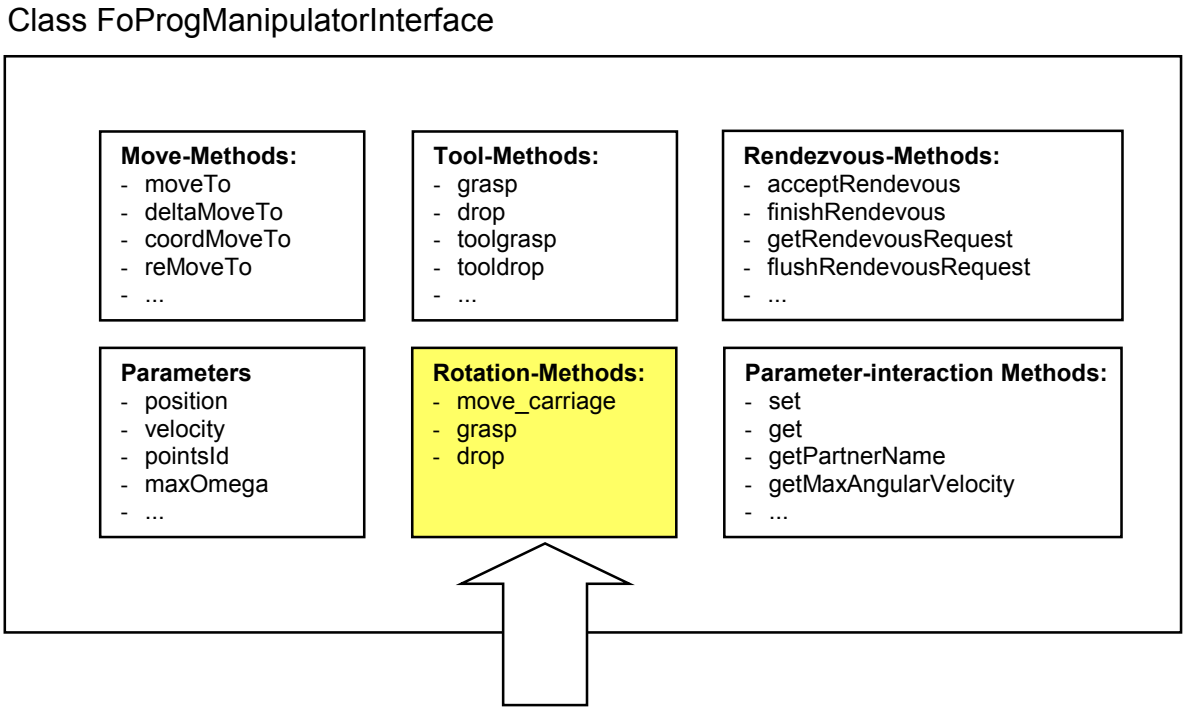


Figure 29: Class FoProgManipulatorInterface and additional Methods for ZYVEX project

In the next step, a py-file has to be written that contains the new command for controlling the carriage. This file is the connection between the Interface-file which executes the actions and the fac-file that regulates the simulation course. Since the newly created ZYVEX manipulator also requires the already existing commands of the standard manipulator, the py-file of the standard manipulator has to be imported.

All discussed ZYVEX files can be found in the appendix section B.
In the final step, the just created files have to be placed in several folders in the path of the Interface Tool. As long as all files are not debugged or/and are not correctly located in the Interface Tool, the ZYVEX manipulator does not work, or in the worst case does not appear in the simulation at all.

4.4.8 Designing a virtual Minifactory

After creating the ZYVEX manipulator, all components that are necessary for the factory set up are available. Similar to the OSTI factory, the ZYVEX factory consists of one unit composed of:

1x base frame + base unit
1x platen tile
1x bridge
6x curb elements
1x ZYVEX overhead manipulator
2x standard courier modules with customized attachments

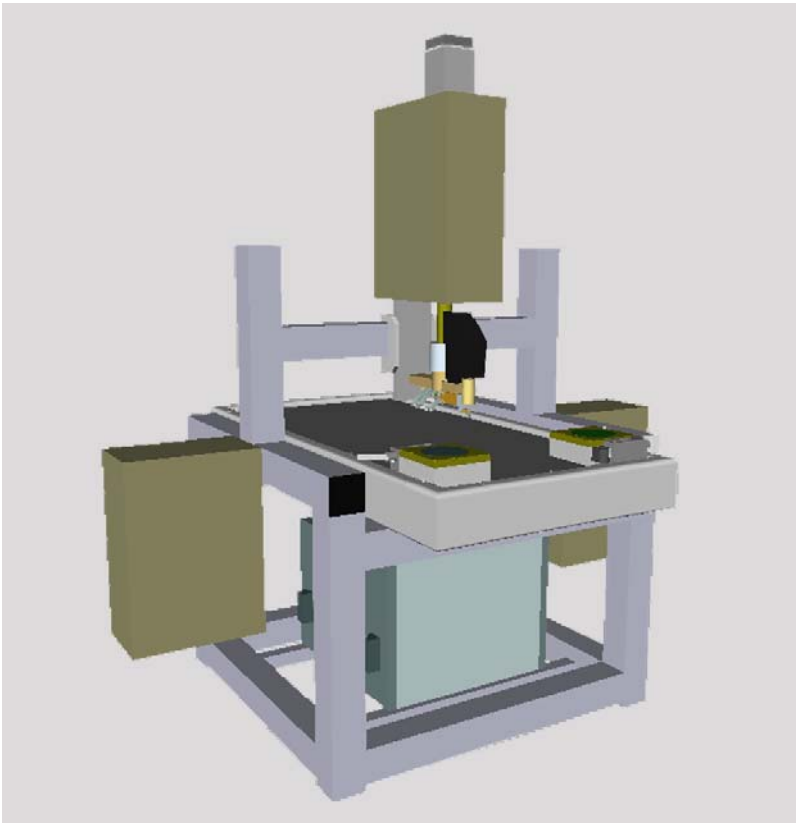


Figure 30: Set up of the virtual ZYVEX Minifactory

The Minifactory components are assembled in the GUI of the Interface Tool. Since this factory is not modeled on an existing real factory, the components do not have to be placed at predefined locations and the laborious manual input of position coordinates is omitted. The layout of this Minifactory (*Figure 30*) resembles the OSTI factory.
On both sides of the base frame, a courier brain box is attached. The bridge with the ZYVEX manipulator clamped in its center is assembled in the middle of the base frame.

4.4.9 Programming the Virtual Factory

The job of this Minifactory is to pick up micro mirrors from one wafer, rotate them 90° and place them perpendicular on the other wafer. Before generating the program, the assembly progress has to be planned. Below the sequence - divided into work steps -for the assembly of one mirror is displayed:

1. move first courier to manipulator
2. lower jammer to the tether until it breaks and pulling back end effector
3. grasp micro mirror and pulling back jammer
4. rotate jammer +90°
5. move first courier aside
6. move second courier to manipulator
7. place micro mirror in socket
8. lower jammer until microconnector legs snap into socket and jammer tip stops touching the micro mirror
9. back the jammer tip out of the micro mirror opening
10. pull back end effector
11. move second courier aside
12. rotate jammer –90°

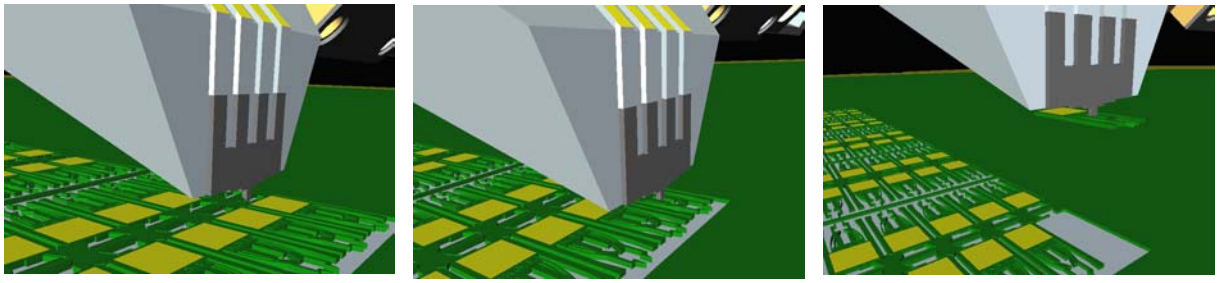


Figure 31: Detach and pick up of a micro mirror

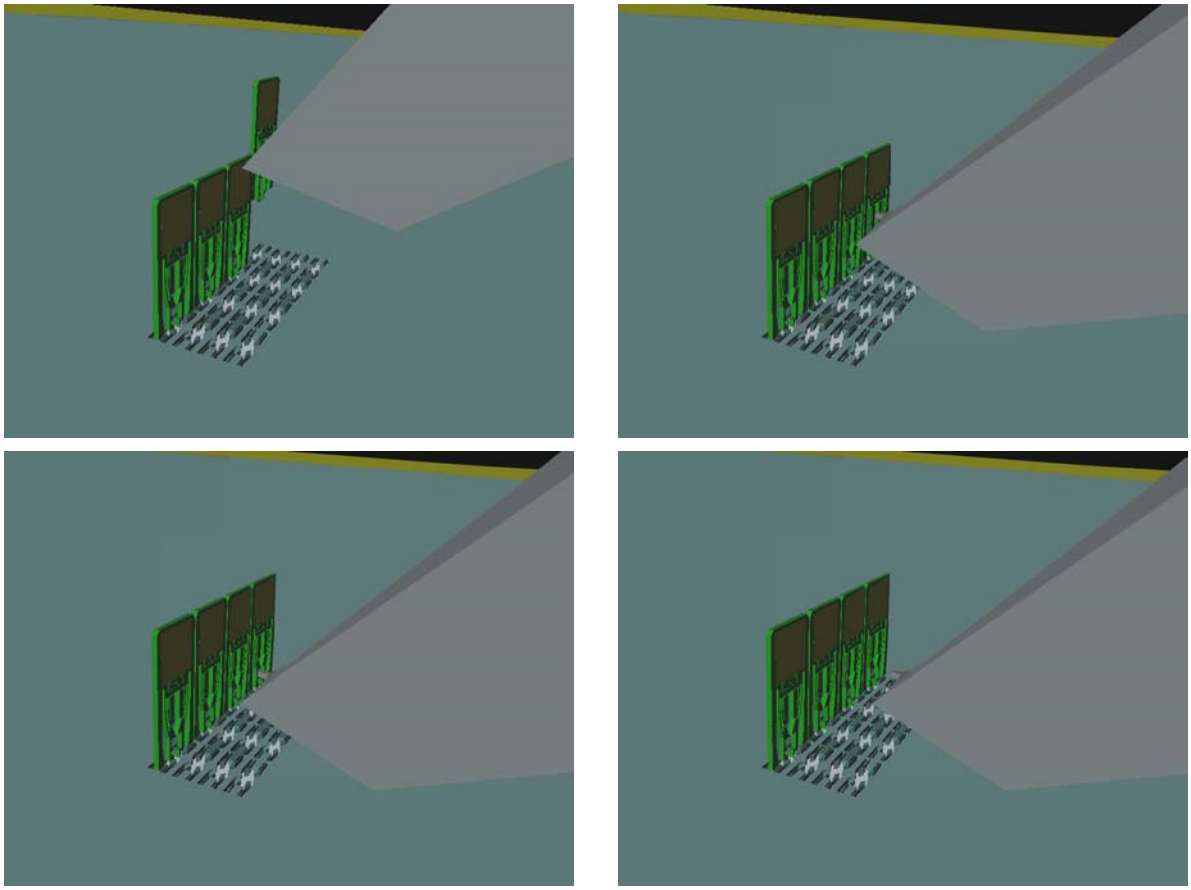


Figure 32: Placing and fixing of a micro mirror

The described assembly sequence shows the basic proceeding. Depending on the position of the micro mirrors and sockets on the wafers, additional rotatory movements of the end effector might be necessary. In the simulation, 32 micro mirrors are assembled, as this is the number of available sockets on the second courier.

After saving the designed ZYVEX factory, the GUI is closed and the ZYVEX fac-file is opened in a compiler environment. In the next step, the action commands have to be integrated into the agents' code. The complete zyvex.fac file

can be found in appendix: B.8 zyvex.fac. Like in the OSTI program, the structure of this program is composed of rendezvous co-operations between the manipulator and the couriers. In these rendezvous, the manipulator takes control over one of the couriers and forms a five DOF robot. Beside the standard commands, like `coordMoveTo`, the newly created command `moveCar` to control the carriage is used. Since the simulation consists of 32 almost identical assembly sequences, loops are suited for structuring the program. By using loops, the number of program lines can be reduced and the program is more clearly arranged.

Due to the task division being inherent in the rendezvous concept of the Interface Tool, all program lines concerning the actual assembly operation are part of the ZYVEX manipulator code. The command lines of the courier consist only of a few commands that initiate and finish the rendezvous. To be sure that the couriers are always ready to form a co-operation with the manipulator, their part of the rendezvous commands is framed in a while-loop. By setting “1” for the while argument, the condition is always met and the couriers start after every finished rendezvous inquiring for the next one. The commands for the manipulator are framed in a for-loop which is run through 32 times. As already mentioned, the assemble sequence is not exactly the same for every micro mirror. In some cases, extra moves of the end effector are required. To consider these variations of the standard sequence, if-else-conditions are added within the for-loop. With these conditions, the additionally required commands are integrated into the assembly sequence whenever it is necessary.

Currently, only a virtual version of the ZYVEX Minifactory exists. That means unlike the OSTI project, the simulation is stand-alone and does not synchronously display the operations a physical factory performs. Thus the time triggering is not affected by physical machines but is hard-coded. To obtain a realistic simulation, one’s attention has to be turned to a proper choice of the velocities of the trajectories. For instance, feed motions are realized in rapid traverse while sensible motions are accordingly realized more slowly.

The result of this project is an operating virtual Minifactory that assembles 32 micro mirrors.

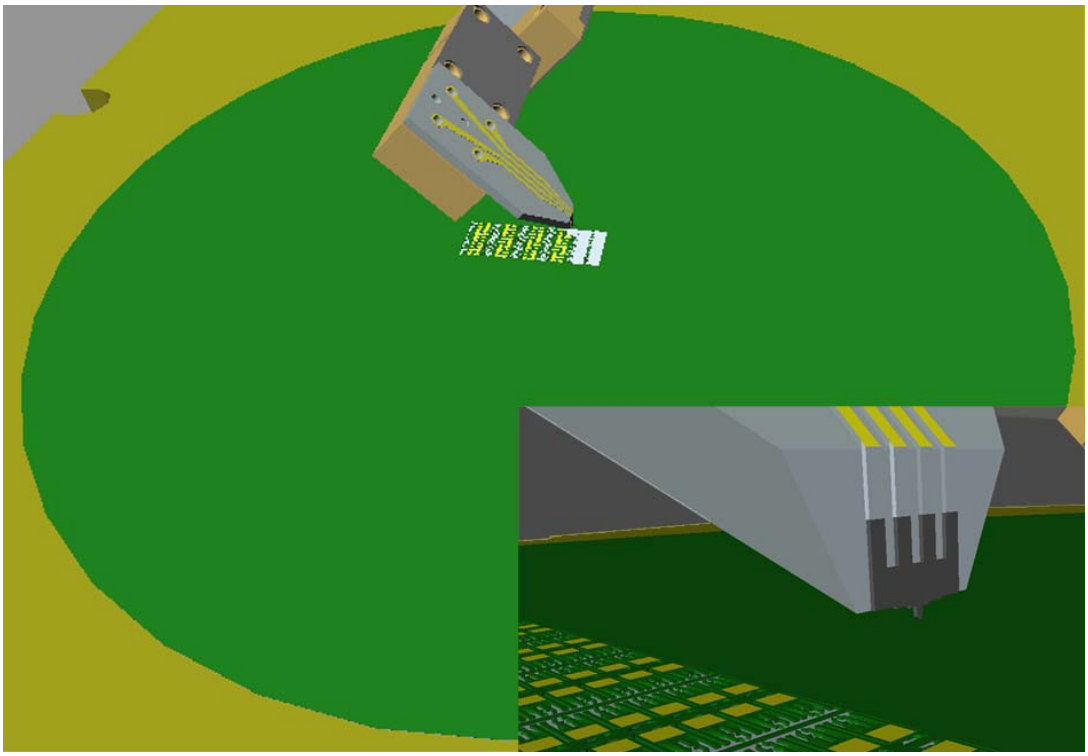


Figure 33: Mirror wafer and jammer tip

4.4.10 Outlook on the ZYVEX Project

After the ZYVEX project has been finished successfully, there are several possibilities for the future of ZYVEX-AAA cooperation. Since a physical version of the effector mechanism already exists, a real factory modeled on the just created virtual version could be set up. To realize the physical factory, real retainers for the wafers would have to be designed and mounted on the couriers. Furthermore, test series with the built-in sensors of the jammer would have to be accomplished to control the tether breaking operation. Similar to the OSTI project, a program in the *tool* environment would have to be written and uploaded to the physical factory agents. Another possibility would be to upgrade the existing virtual factory. ZYVEX developed a jammer that consists of several parallel tips and can pick up and place many parts at once. Thus, the Minifactory could operate more efficiently. To use the newly developed jammer, the effector mechanism and the wafer layout would have to be modified. As the current virtual ZYVEX Minifactory is a test set up to demonstrate the quali-

fication of AAA's Minifactory for handling MEMS, a more realistic version could be programmed. A virtual Minifactory that assembles a complete MEMS structure could be created. The microconnector system, ZYVEX developed, applies for all micro components which are designed for a handling with a passiveend effector. For that reason, the newly created ZYVEX manipulator could be used for the assembly of other micro components, too.

A virtual Minifactory could be set up which consists of several units and agents and assembles 3D MEMS structures that consist of many components.

In the end, a direct comparison of the physical Minifactory and the ZYVEX five DOF robot system will decide on the future of AAA's Minifactory with the ZYVEX Corporation. Both systems have to assemble a similar item and the assembly quality and quantity have to be checked.

In any case, Minifactory is predestined for assembly task like the one presented in this work and has the potential to revolutionize the segment of micro assembly.

5. Summary and Conclusion

Before attending to the actual projects, Interface Tool had to be reconditioned and upgraded. The transfer to a new computer system could not be performed without damage. Thus, in the beginning bugs had to be corrected and missing files and libraries had to be provided. After refitting the Interface Tool, the first project could be started. A virtual Minifactory for the assembly of telescopic sights was designed and a simulation was created. The programming of the physical factory is based on this virtual factory. During this project, problems that were not foreseeable before the project start had to be solved. In general, this project presented a good preparation for the ZYVEX project. By using already developed Minifactory components in the first assembly project, this part of the work could be used to acquire more knowledge about the Interface Tool. Thus, in the second project, already familiar with all prerequisites, the main attention could be turned to the expansion of the Interface Tool's functionality. To assemble the MEMS with Minifactory, a fifth DOF had to be implemented in the virtual environment of Interface Tool. To do this, the C++ source code of the Interface Tool had to be modified and supplemented. Before proceeding with the next step, the entire structure of the Interface Tool had to be analyzed and understood. After adding the extra movement axis, a virtual Minifactory was generated that picks up micro components, rotates them via the new DOF and places them perpendicular to the original orientation. That way, 3D MEMS structures can be built. The result of this project is not only an operating virtual Minifactory that assembles MEMSs, but also an reassessment of the entire Minifactory system. The fifth degree of freedom grants access to the MEMS industry, representing also an opportunity for the Minifactory to address problems not approachable before.

5.1 Future Work and Outlook

Since Interface Tool, especially the functionality of the GUI, still has not achieved its original condition on the Silicon Graphics machine, there is still

some repair work left. Once Interface Tool is in complete working order again, virtual factories and simulations can be created in shorter time and in a more convenient way. In general, it is always favorable to develop the component palette of Interface Tool. Every additional factory module, especially robot agents, extend the field of application. With every further project, new end effectors and assembly tools or even new robot agents are added to the component library of the Interface Tool.

By the time Interface Tool can provide all planned functionalities and the test stage has been finished, the *sim* and *tool* environments can be replaced by a single programming environment. Which would offer the capability to design and simulate factories and generate programs for the physical factories at the same time.

Efficient physical Minifactory modules in combination with a user-friendly software system and the agility concept will ensure a wealth of users.

Once Minifactory is established, several different manufacturers can design Minifactory modules with standardized hardware and software interfaces and a variety of robot agents will be available. Manufacturers, specialized in certain assembly techniques can combine there know-how with the concept of AAA and Minifactory and offer virtual and physical robot agents. Due to shorter product live cycles, Minifactories will be in a constant alteration. Thus modules which fall into disuse can be stocked or sold while required modules can be purchased newly or second-hand.

In the near future Minifactory could be a novel assembly system that changes the current way of fabrication in many branches of industry.

6 References

6.1 List of Literature

BROWN et al. 2001

Brown H. B.; Muir P.; Rizzi A.: Proceeding of the 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS ’10), volume 2, pages 1030 – 1035, A precision manipulator module for assembly in a minifactory environment. 2001.

ELLIS et al. 2002

Ellis M.; Skidmore G.; Geisberger A.: Microfabricated Silicon Mechanical Connectors and Assembly. Richardson, Texas USA: 2002.

GOWDY 1999

Gowdy J.; Butler Z. J.: An integrated Interface Tool for the Architecture for Agile Assembly, Carnegie Mellon University (1999).

HOLLIS 1995

Hollis R. L.; Quaid A.: Proc. Am. Soc. of Precision Engineering, 10th Annual Mtg. .An Architecture for Agile Assembly. Austin USA, October 15-19 1995.

HOLLIS 2003

Hollis R. L.; Rizzi A. A.: Proceeding Int’l Advanced Robotics Program. Towards a second-generation minifactory for precision assembly. Moscow Russia, April 2003.

HOLLIS et al. 2006

Hollis R. L.; Sarkar N.; Jones J.: 5th Int’l Symp. On Minifactories, Visio Guided Pick and Place in a Minifactory Environment. Beascon France, October 25-27 2003.

MA et al. 2000

Ma W. C.; Rizzi A.; Hollis R.: IEEE International Conference on Robotics and Automation 2000. Optical coordination sensor for precision cooperating robots. April 2000.

MEMS AND NANOTECHNOLOGY EXCHANGE

Mems and nanotechnology exchange.: <http://www.memsnet.org> - 10.02.2007.

MEMSCAP

MEMSCAP Inc.: <http://www.allaboutmems.com> - 10.02.2007.

QUAID 1998

Quaid A.; Hollis R. L.: IEEE International Conference on Robotics and Automation. Cooperative 3-dof closed-loop control for planar linear motors. May 1998.

RIZZI 1997

Rizzi A. A.; Gowdy J.: International Conference of Robotics and Automation. Agile Assembly Architecture: An Agent Based Approach to Modular Precision Assembly Systems. 1997.

TSUI et al. 2003

Tsui, k.; Geisberg, A. A.; Ellis M.: International Electronic Packaging Technical Conferences and Exhibition. Calibration systems and techniques for automated microassembly. Maui, Hawaii USA, July 6-11 2003.

TSUI et al. 2004

Tsui, K.; Geisberg, A. A.; Ellis M.: Micromachined end-effector and techniques for directed MEMS assembly. Journal of Micromechanics and Microengineering (2004) 14, S. 542-549.

ZYVEX

Zyvex Corporation: <http://www.zyvex.com> - 10.02.2007.

6.2 List of Figures

A OSTI Files

A.1 Sim: osti.fac

```
file base_frame.aaa {
    children {
        file lg_platen.aaa {

# naming platen so that later it can be defined on which platen the couriers move:
            name P1

            matrix [ 1 0 0 0 0 1 0 0 0 0 1 0 0 0 570 1 ]
        }

################### courier C1 ######################################

        file Osti_courier_1.aaa {

# naming the courier:
            name CS1

matrix [ 1 0 0 0 0 1 0 0 0 0 1 0 450 -330 655 1 ]

# reference to py-file containing the command definitions:
            program {from progCourier import *

while 1:

# adding the collimator housing to the simulation:
               createPart(self,"collimator.aaa","screwFixture-0",0)

# courier C1 transfers its control to manipulator M1 for the period of the ren-
#dezvous operation "Collimaator4a":
               initiateRendevous(self, "M1", "Collimator4a")
               reserve(self, "M1")
               performRendevous(self)
               endRendevous(self)
               unreserve(self, "M1")

# move C1 to home position:
               goTo(self,"P1",200,-400)

               initiateRendevous(self, "M1", "Collimator4b")
               reserve(self, "M1")
               performRendevous(self)
               endRendevous(self)
               unreserve(self, "M1")

               goTo(self,"P1",200,-400)


               initiateRendevous(self, "M1", "Collimator3a")
               reserve(self, "M1")
               performRendevous(self)
               endRendevous(self)
               unreserve(self, "M1")

               goTo(self,"P1",200,-400)

                       initiateRendevous(self, "M1", "Collimator3b")
               reserve(self, "M1")
               performRendevous(self)
               endRendevous(self)
               unreserve(self, "M1")
```

```
                goTo(self,"P1",200,-400)


                initiateRendevous(self, "M1", "Collimator2a")
                reserve(self, "M1")
                performRendevous(self)
                endRendevous(self)
                unreserve(self, "M1")

                goTo(self,"P1",200,-400)

                        initiateRendevous(self, "M1", "Collimator2b")
                reserve(self, "M1")
                performRendevous(self)
                endRendevous(self)
                unreserve(self, "M1")

                goTo(self,"P1",200,-400)


                initiateRendevous(self, "M1", "Collimator1a")
                reserve(self, "M1")
                performRendevous(self)
                endRendevous(self)
                unreserve(self, "M1")

                goTo(self,"P1",200,-400)

                        initiateRendevous(self, "M1", "Collimator1b")
                reserve(self, "M1")
                performRendevous(self)
                endRendevous(self)
                unreserve(self, "M1")

                goTo(self,"P1",200,-400)


}
            member home {
                matrix [ 1 0 0 0 0 1 0 0 0 0 1 0 -450 330 15 1 ]
            }
            member motor {
                matrix [ 1 0 0 0 0 1 0 0 0 0 1 0 158.518 -447.676 0 1 ]
            }
        }
        file Osti_courier_2.aaa {


#################### courier C2 ########################################

# naming the courier:
            name CS2

            matrix [ -0.999989 -1.50994e-07 -1.16675e-22 0 1.50994e-07 -0.999989
6.84564e-08 0 -1.40427e-13 6.8457e-08 1 0 -450 163.955 655 1 ]

# reference to py-file containing the command definitions:
            program {from progCourier import *

while 1:

              sleep(self,0.5)
# adding the collimator components to the simulation:
              createPart(self,"element_1.aaa","element_1_fixture",0)
              createPart(self,"element_2.aaa","element_2_fixture",0 )
              createPart(self,"element_3_4_doublet.aaa","element_34_fixture",0)
#
              createPart(self,"element_5.aaa","element_5_fixture",0)
              createPart(self,"lock_ring_el_1.aaa","ring_1_fixture",0)
```

```
              createPart(self,"lock_ring_el_3.aaa","ring_34_fixture",0)
              createPart(self,"lock_ring_el_5.aaa","ring_5_fixture",0)
              createPart(self,"spacer_el_1_2_ver2.aaa","spacer_fixture",0)
              sleep(self,1)

# add the pick-up tools to the simulation:
              creat-
Part(self,"vacuum_pickup_tool_lens_el1.aaa","element_1_fixture",1)
              create-
Part(self,"vacuum_pickup_tool_lens_el2.aaa","element_2_fixture",1)
              create-
Part(self,"vacuum_pickup_tool_lens_el3_4.aaa","element_34_fixture",1)
              create-
Part(self,"vacuum_pickup_tool_lens_el5.aaa","element_5_fixture",1)    #
              createPart(self,"ringtool_1.aaa","ring_1_fixture",1)
              createPart(self,"ringtool_2.aaa","spacer_fixture",1)
              createPart(self,"ringtool_3.aaa","ring_34_fixture",1)
              createPart(self,"ringtool_5.aaa","ring_5_fixture",1)

# courier C2 transfering its controll to manipulator M1 for the period of the
ren#dezvous operation "Lens_4":
              initiateRendevous(self, "M1", "Lens_4")
              reserve(self,"M1")
              performRendevous(self)
              endRendevous(self)
              unreserve(self, "M1")

# moving C1 to home position
              goTo(self,"P1",-200,-400)

              initiateRendevous(self, "M1", "tool4a")
              reserve(self,"M1")
              performRendevous(self)
              endRendevous(self)
              unreserve(self, "M1")

              goTo(self,"P1",-200,-400)

              initiateRendevous(self, "M1", "Retainer_4")
              reserve(self,"M1")
              performRendevous(self)
              endRendevous(self)
              unreserve(self, "M1")

              goTo(self,"P1",-200,-400)

              initiateRendevous(self, "M1", "tool4b")
              reserve(self,"M1")
              performRendevous(self)
              endRendevous(self)
              unreserve(self, "M1")

              goTo(self,"P1",-200,-400)

              initiateRendevous(self, "M1", "Lens_3")
              reserve(self,"M1")
              performRendevous(self)
              endRendevous(self)
              unreserve(self, "M1")

              goTo(self,"P1",-200,-400)

              initiateRendevous(self, "M1", "tool3a")
              reserve(self,"M1")
              performRendevous(self)
              endRendevous(self)
              unreserve(self, "M1")

              goTo(self,"P1",-200,-400)
```

```
initiateRendevous(self, "M1", "Retainer_3")
reserve(self,"M1")
performRendevous(self)
endRendevous(self)
unreserve(self, "M1")

goTo(self,"P1",-200,-400)

initiateRendevous(self, "M1", "tool3b")
reserve(self,"M1")
performRendevous(self)
endRendevous(self)
unreserve(self, "M1")

goTo(self,"P1",-200,-400)

initiateRendevous(self, "M1", "Lens_2")
reserve(self,"M1")
performRendevous(self)
endRendevous(self)
unreserve(self, "M1")

goTo(self,"P1",-200,-400)

initiateRendevous(self, "M1", "tool2b")
reserve(self,"M1")
performRendevous(self)
endRendevous(self)
unreserve(self, "M1")

goTo(self,"P1",-200,-400)

initiateRendevous(self, "M1", "Spacer")
reserve(self,"M1")
performRendevous(self)
endRendevous(self)
unreserve(self, "M1")

goTo(self,"P1",-200,-400)

initiateRendevous(self, "M1", "tool2b")
reserve(self,"M1")
performRendevous(self)
endRendevous(self)
unreserve(self, "M1")

goTo(self,"P1",-200,-400)

initiateRendevous(self, "M1", "Lens_1")
reserve(self,"M1")
performRendevous(self)
endRendevous(self)
unreserve(self, "M1")

goTo(self,"P1",-200,-400)

initiateRendevous(self, "M1", "tool1a")
reserve(self,"M1")
performRendevous(self)
endRendevous(self)
unreserve(self, "M1")

goTo(self,"P1",-200,-400)

initiateRendevous(self, "M1", "Retainer_1")
reserve(self,"M1")
performRendevous(self)
endRendevous(self)
unreserve(self, "M1")
```

```
              goTo(self,"P1",-200,-400)

              initiateRendevous(self, "M1", "tool1b")
              reserve(self,"M1")
              performRendevous(self)
              endRendevous(self)
              unreserve(self, "M1")

              goTo(self,"P1",-200,-400)


}
            member home {
                matrix [ 1 0 0 0 0 1 0 0 0 0 1 0 -450.005 163.957 14.9999 1 ]
            }
            member motor {
                matrix [ 1 0 0 0 0 1 0 0 0 0 1 0 156.115 441.65 -1.90735e-05 1 ]
            }
        }
        file bridge.aaa {
            matrix [ 1 0 0 0 0 1 0 0 0 0 1 0 0 147.233 920 1 ]
            member crossbar {
                matrix [ 1 0 0 0 0 1 1.46125e-15 0 0 -1.46099e-15 1 0 -3.57628e-05
1.89761e-06 96.0405 1 ]
                children {


#################### manipulator ######################################

                    file Osti_manip.aaa {

# naming the courier:
                        name M1

                        matrix [ -0.999994 2.16463e-09 1.54238e-16 0 4.23855e-08 -
0.999993 6.98763e-22 0 5.03578e-14 -1.82624e-13 1 0 4.10435 -39.9999 9.53675e-07 1
]

# reference to py-file containing the command definitions:
                        program {from progManipulator import *

while 1:

############## lens element #4 ######################################

#cooperation between C2 and M:
              acceptRendevous(self,"Lens_4")

# pick up part and tool at once:
              coordMoveTo(self, 0, 50.8, 87.9882, 70, 0, 0.5)
              coordMoveTo(self, 0, 50.8, 87.9882, 40, 0, 0.1)
              grasp(self,2,'element_5_fixture')
              grasp(self,1,'element_5_fixture')
              coordMoveTo(self, 0, 50.8, 87.9882, 40, 0, 0.1)
              coordMoveTo(self, 0, 50.8, 87.9882, 70, 0, 0.5)

              finishRendevous(self)

# cooperation between C1 and M:
              acceptRendevous(self,"Collimator4a")
# placing part:
              coordMoveTo(self, 0, 0, 25.399, 85, 0, 0.5)
              coordMoveTo(self, 0, 0, 25.399, 43.4848, 0, 0.1)
              drop(self, 0,'screwFixture-0')
              coordMoveTo(self, 0, 0, 25.399, 50, 0, 0.1)
              coordMoveTo(self, 0, 0, 25.399, 85, 0, 0.5)

              finishRendevous(self)

# cooperation between C2 and M:
```

```
                acceptRendevous(self,"tool4a")

# placing tool:
                coordMoveTo(self, 0, 50.8, 87.9882, 70, 0, 0.5)
                coordMoveTo(self, 0, 50.8, 87.9882, 40, 0, 0.1)
                drop(self, 0,'element_5_fixture')
                coordMoveTo(self, 0, 50.8, 87.9882, 40, 0, 0.1)
                coordMoveTo(self, 0, 50.8, 87.9882, 70, 0, 0.5)
                finishRendevous(self)

############## retainer ring #4 ######################################

                acceptRendevous(self,"Retainer_4")
                coordMoveTo(self, 0, -50.8, 87.9882, 70, 0, 0.5)
                coordMoveTo(self, 0, -50.8, 87.9882, 40, 0, 0.1)
                grasp(self,2,'ring_5_fixture')
                grasp(self,1,'ring_5_fixture')
                coordMoveTo(self, 0, -50.8, 87.9882, 40, 0, 0.1)
                coordMoveTo(self, 0, -50.8, 87.9882, 70, 0, 0.5)

                finishRendevous(self)

                acceptRendevous(self,"Collimator4b")
                coordMoveTo(self, 0, 0, 25.399, 85, 0, 0.5)
                coordMoveTo(self, 0, 0, 25.399, 47.3267, 0, 0.1)
                drop(self, 0,'screwFixture-0')
                coordMoveTo(self, 0, 0, 25.399, 50, 0, 0.1)
                coordMoveTo(self, 0, 0, 25.399, 85, 0, 0.5)

                finishRendevous(self)

                acceptRendevous(self,"tool4b")
                coordMoveTo(self, 0, -50.8, 87.9882, 70, 0, 0.5)
                coordMoveTo(self, 0, -50.8, 87.9882, 40, 0, 0.1)
                drop(self, 0,'ring_5_fixture')
                coordMoveTo(self, 0, -50.8, 87.9882, 40, 0, 0.1)
                coordMoveTo(self, 0, -50.8, 87.9882, 70, 0, 0.5)

                finishRendevous(self)

############## lens element #3 ######################################

                        acceptRendevous(self,"Lens_3")
                coordMoveTo(self, 0, -25.4, 43.9941, 70, 0, 0.5)
                coordMoveTo(self, 0, -25.4, 43.9941, 40, 0, 0.1)
                grasp(self,2,'element_34_fixture')
                grasp(self,1,'element_34_fixture')
                coordMoveTo(self, 0, -25.4, 43.9941, 40, 0, 0.1)
                coordMoveTo(self, 0, -25.4, 43.9941, 70, 0, 0.5)

                finishRendevous(self)

                acceptRendevous(self,"Collimator3a")
                coordMoveTo(self, 0, 0, 25.399, 85, 0, 0.5)
                coordMoveTo(self, 0, 0, 25.399, 58.4708, 0, 0.1)
                drop(self, 0,'screwFixture-0')
                coordMoveTo(self, 0, 0, 25.399, 60, 0, 0.1)
                coordMoveTo(self, 0, 0, 25.399, 85, 0, 0.5)

                finishRendevous(self)

                acceptRendevous(self,"tool3a")
                coordMoveTo(self, 0, -25.4, 43.9941, 70, 0, 0.5)
                coordMoveTo(self, 0, -25.4, 43.9941, 40, 0, 0.1)
                drop(self, 0,'element_34_fixture')
                coordMoveTo(self, 0, -25.4, 43.9941, 40, 0, 0.1)
                coordMoveTo(self, 0, -25.4, 43.9941, 70, 0, 0.5)

                finishRendevous(self)
```

```
############## retainer ring #3 ######################################

              acceptRendevous(self,"Retainer_3")
              coordMoveTo(self, 0, 25.4, 43.9941, 70, 0, 0.5)
              coordMoveTo(self, 0, 25.4, 43.9941, 40, 0, 0.1)
              grasp(self,2,'ring_34_fixture')
              grasp(self,1,'ring_34_fixture')
              coordMoveTo(self, 0, 25.4, 43.9941, 40, 0, 0.1)
              coordMoveTo(self, 0, 25.4, 43.9941, 70, 0, 0.5)

              finishRendevous(self)

              acceptRendevous(self,"Collimator3b")
              coordMoveTo(self, 0, 0, 25.399, 85, 0, 0.5)
              coordMoveTo(self, 0, 0, 25.399, 67.5202, 0, 0.1)
              drop(self, 0,'screwFixture-0')
              coordMoveTo(self, 0, 0, 25.399, 70, 0, 0.1)
              coordMoveTo(self, 0, 0, 25.399, 85, 0, 0.5)

              finishRendevous(self)

              acceptRendevous(self,"tool3b")
              coordMoveTo(self, 0, 25.4, 43.9941, 70, 0, 0.5)
              coordMoveTo(self, 0, 25.4, 43.9941, 40, 0, 0.1)
              drop(self, 0,'ring_34_fixture')
              coordMoveTo(self, 0, 25.4, 43.9941, 40, 0, 0.1)
              coordMoveTo(self, 0, 25.4, 43.9941, 70, 0, 0.5)

              finishRendevous(self)

############## lens element #2 ######################################

              acceptRendevous(self,"Lens_2")
              coordMoveTo(self, 0, 25.4, -43.9941, 70, 0, 0.5)
              coordMoveTo(self, 0, 25.4, -43.9941, 40, 0, 0.1)
              grasp(self,2,'element_2_fixture')
              grasp(self,1,'element_2_fixture')
              coordMoveTo(self, 0, 25.4, -43.9941, 40, 0, 0.1)
              coordMoveTo(self, 0, 25.4, -43.9941, 70, 0, 0.5)

              finishRendevous(self)

              acceptRendevous(self,"Collimator2a")
              coordMoveTo(self, 0, 0, 25.399, 85, 0, 0.5)
              coordMoveTo(self, 0, 0, 25.399, 72.9486, 0, 0.1)
              drop(self, 0,'screwFixture-0')
              coordMoveTo(self, 0, 0, 25.399, 75, 0, 0.1)
              coordMoveTo(self, 0, 0, 25.399, 85, 0, 0.5)

              finishRendevous(self)

              acceptRendevous(self,"tool2a")
              coordMoveTo(self, 0, 25.4, -43.9941, 70, 0, 0.5)
              coordMoveTo(self, 0, 25.4, -43.9941, 40, 0, 0.1)
              drop(self, 0,'element_2_fixture')
              coordMoveTo(self, 0, 25.4, -43.9941, 40, 0, 0.1)
              coordMoveTo(self, 0, 25.4, -43.9941, 70, 0, 0.5)

              finishRendevous(self)

############## spacer ################################################

              acceptRendevous(self,"Spacer")
              coordMoveTo(self, 0, -25.4,-43.9941, 70, 0, 0.5)
              coordMoveTo(self, 0, -25.4,-43.9941, 40, 0, 0.1)
              grasp(self,2,'spacer_fixture')
              grasp(self,1,'spacer_fixture')
              coordMoveTo(self, 0, 25.4, 43.9941, 40, 0, 0.1)
              coordMoveTo(self, 0, 25.4, 43.9941, 70, 0, 0.5)
```

```
            finishRendevous(self)

            acceptRendevous(self,"Collimator2b")
            coordMoveTo(self, 0, 0, 25.399, 85, 0, 0.5)
            coordMoveTo(self, 0, 0, 25.399, 74.8337, 0, 0.1)
            drop(self, 0,'screwFixture-0')
            coordMoveTo(self, 0, 0, 25.399, 78, 0, 0.1)
            coordMoveTo(self, 0, 0, 25.399, 85, 0, 0.5)

            finishRendevous(self)

            acceptRendevous(self,"tool2b")
            coordMoveTo(self, 0, -25.4,-43.9941, 70, 0, 0.5)
            coordMoveTo(self, 0, -25.4,-43.9941, 40, 0, 0.1)
            drop(self, 0,'spacer_fixture')
            coordMoveTo(self, 0, -25.4,-43.9941, 40, 0, 0.1)
            coordMoveTo(self, 0, -25.4,-43.9941, 70, 0, 0.5)

            finishRendevous(self)

############## lens element #1 ######################################

            acceptRendevous(self,"Lens_1")
            coordMoveTo(self, 0, 50.8,0, 70, 0, 0.5)
            coordMoveTo(self, 0, 50.8,0, 40, 0, 0.1)
            grasp(self,2,'element_1_fixture')
            grasp(self,1,'element_1_fixture')
            coordMoveTo(self, 0, 50.8,0, 40, 0, 0.1)
            coordMoveTo(self, 0, 50.8,0, 70, 0, 0.5)

            finishRendevous(self)

            acceptRendevous(self,"Collimator1a")
            coordMoveTo(self, 0, 0, 25.399, 85, 0, 0.5)
            coordMoveTo(self, 0, 0, 25.399, 78.0517, 0, 0.1)
            drop(self, 0,'screwFixture-0')
            coordMoveTo(self, 0, 0, 25.399, 80, 0, 0.1)
            coordMoveTo(self, 0, 0, 25.399, 85, 0, 0.5)

            finishRendevous(self)

            acceptRendevous(self,"tool1a")
            coordMoveTo(self, 0, 50.8,0, 70, 0, 0.5)
            coordMoveTo(self, 0, 50.8,0, 40, 0, 0.1)
            drop(self, 0,'element_1_fixture')
            coordMoveTo(self, 0, 50.8,0, 40, 0, 0.1)
            coordMoveTo(self, 0, 50.8,0, 70, 0, 0.5)

            finishRendevous(self)

############## retainer ring #1 ######################################

            acceptRendevous(self,"Retainer_1")
            coordMoveTo(self, 0, -50.8, 0, 70, 0, 0.5)
            coordMoveTo(self, 0, -50.8, 0, 40, 0, 0.1)
            grasp(self,2,'ring_1_fixture')
            grasp(self,1,'ring_1_fixture')
            coordMoveTo(self, 0, -50.8, 0, 40, 0, 0.1)
            coordMoveTo(self, 0, -50.8, 0, 70, 0, 0.5)

            finishRendevous(self)

            acceptRendevous(self,"Collimator1b")
            coordMoveTo(self, 0, 0, 25.399, 85, 0, 0.5)
            coordMoveTo(self, 0, 0, 25.399, 79.1747, 0, 0.1)
            drop(self, 0,'screwFixture-0')
            coordMoveTo(self, 0, 0, 25.399, 80, 0, 0.1)
            coordMoveTo(self, 0, 0, 25.399, 85, 0, 0.5)

            finishRendevous(self)
```

```
            acceptRendevous(self,"tool1b")
            coordMoveTo(self, 0, -50.8, 0, 70, 0, 0.5)
            coordMoveTo(self, 0, -50.8, 0, 40, 0, 0.1)
            drop(self, 0,'ring_1_fixture')
            coordMoveTo(self, 0, -50.8, 0, 40, 0, 0.1)
            coordMoveTo(self, 0, -50.8, 0, 70, 0, 0.5)

            finishRendevous(self)

}

                    member effectorLink {
                        children {
                        }
                    }
                    member base {
                        matrix [ 1 0 0 0 0 1 0 0 0 0 1 0 0 0 100 1 ]
                    }
                }
            }
        }
    }
}
```

A.2 Tool: osti.fac

```
file base_frame.aaa {
    children {
        file lg_platen.aaa {
            name P1
            matrix [ 1 0 0 0 0 1 0 0 0 0 1 0 0 0 570 1 ]
        }
        cached rat ipt:englishhorn.msl.ri.cmu.edu|1390,interface {
            name CS1
            matrix [ 1 0 0 0 0 1 0 0 0 0 1 0 450 -330 655 1 ]
            program {
from OSTICourierProgram import OSTICourierProgram
from FoDescription import FoDescriptionPtr
import linear

class Program(OSTICourierProgram):
    def bind(self):
        OSTICourierProgram.bind(self)
#        self.collimator_proto = self.bindPrototype("Collimator")
        self.collimatorhalf_proto = self.bindPrototype("Collimatorhalf")

    def run(self):
#        self.collimator = self.attachPart(self.collimator_proto,
#                                          "screwFixture_0")
        self.collimator = self.attachPart(self.collimatorhalf_proto,
                                          "screwFixture_0")

        # reset the courier's position to be offset (0,0) from the lower right
        # platen corner
        self.setHome(1, -1, 0.0, 0.0)

        manip_intf = self.manip.getInterface()
        self.manip_object = self.getProgramObject(manip_intf)

        self.do_operation("Element5Place")

        self.do_operation("Ring5Place")
```

```
        self.do_operation("Element34Place")

        self.do_operation("Ring34Place")

        self.do_operation("Element2Place")

        self.do_operation("SpacerPlace")

        self.do_operation("Element1Place")

        self.do_operation("Ring1Place")

        while 1:
            self.sleep(0.2)

    def do_operation(self, place_rendezvous):
        print "Initiating"
        self.initiateRendezvous(self.manip, place_rendezvous)

        print "Coordinating"
        fixture = FoDescriptionPtr(self.description.screwFixture_0)
        mat = fixture.getLocalMatrix()
        self.coordinateTo(self.manip_object, mat[3][0], mat[3][1])
        print "Finishing"

        self.coordinateTo(self.manip_object, -200, 0)
        print "Cleared"
        self.finishRendezvous(place_rendezvous)


program = Program()
}
            member home {
                matrix [ 1 0 0 0 0 1 0 0 0 0 1 0 -450 330 15 1 ]
                children {
                }
            }
            member motor {
                matrix [ 1 0 0 0 0 1 0 0 0 0 1 0 158.518 -447.676 0 1 ]
                children {
                }
            }
        }
        cached vole ipt:englishhorn.msl.ri.cmu.edu|1391,interface {
            name CS2
            matrix [ -0.999989 -1.50994e-07 -1.16675e-22 0 1.50994e-07 -0.999989
6.84564e-08 0 -1.40427e-13 6.8457e-08 1 0 -450 163.955 655 1 ]
            program {
from OSTICourierProgram import OSTICourierProgram
from FoDescription import FoDescriptionPtr
import linear

class Program(OSTICourierProgram):
    def bind(self):
        OSTICourierProgram.bind(self)
        self.element_1_proto = self.bindPrototype("Element1")
        self.element_2_proto = self.bindPrototype("Element2")
        self.element_3_4_doublet_proto = self.bindPrototype("Element34Doublet")
        self.element_5_proto = self.bindPrototype("Element5")
        self.lock_ring_el_1_proto = self.bindPrototype("LockRingElement1")
        self.lock_ring_el_3_proto = self.bindPrototype("LockRingElement3")
        self.lock_ring_el_5_proto = self.bindPrototype("LockRingElement5")
        self.spacer_el_1_2_proto = self.bindPrototype("SpacerElements12")
        self.vacuum_pickup_tool_lens_el1_proto =
self.bindPrototype("VacuumPickupToolLensElement1")
        self.vacuum_pickup_tool_lens_el2_proto =
self.bindPrototype("VacuumPickupToolLensElement2")
        self.vacuum_pickup_tool_lens_el3_4_proto =
self.bindPrototype("VacuumPickupToolLensElement34")
```

```
        self.vacuum_pickup_tool_lens_el5_proto =
self.bindPrototype("VacuumPickupToolLensElement5")
        self.ringtool_1_proto = self.bindPrototype("RingTool1")
        self.ringtool_2_proto = self.bindPrototype("RingTool2")
        self.ringtool_3_proto = self.bindPrototype("RingTool3")
        self.ringtool_5_proto = self.bindPrototype("RingTool5")

    def run(self):
        self.element_1 = self.attachPart(self.element_1_proto, "element_1_fixture")
        self.element_2 = self.attachPart(self.element_2_proto, "element_2_fixture")
        self.element_3_4_doublet = self.attachPart(self.element_3_4_doublet_proto,
"element_34_fixture")
        self.element_5 = self.attachPart(self.element_5_proto, "element_5_fixture")
        self.lock_ring_el_1 = self.attachPart(self.lock_ring_el_1_proto,
"ring_1_fixture")
        self.lock_ring_el_3 = self.attachPart(self.lock_ring_el_3_proto,
"ring_34_fixture")
        self.lock_ring_el_5 = self.attachPart(self.lock_ring_el_5_proto,
"ring_5_fixture")
        self.spacer_el_1_2 = self.attachPart(self.spacer_el_1_2_proto,
"spacer_fixture")
        # note: pick better z-offsets here
        self.vacuum_pickup_tool_lens_el1 =
self.attachPart(self.vacuum_pickup_tool_lens_el1_proto, "element_1_fixture")
        self.vacuum_pickup_tool_lens_el2 =
self.attachPart(self.vacuum_pickup_tool_lens_el2_proto, "element_2_fixture")
        self.vacuum_pickup_tool_lens_el3_4 =
self.attachPart(self.vacuum_pickup_tool_lens_el3_4_proto,
                        "element_34_fixture")
        self.vacuum_pickup_tool_lens_el5 =
self.attachPart(self.vacuum_pickup_tool_lens_el5_proto, "element_5_fixture")
        self.ringtool_1 = self.attachPart(self.ringtool_1_proto, "ring_1_fixture")
        self.ringtool_2 = self.attachPart(self.ringtool_2_proto, "spacer_fixture")
        self.ringtool_3 = self.attachPart(self.ringtool_3_proto, "ring_34_fixture")
        self.ringtool_5 = self.attachPart(self.ringtool_5_proto, "ring_5_fixture")

        # reset the courier's position to be offset (0,0) from the lower right
        # platen corner.  Remember "cornerness" is from point of view of the
        # courier, which is upside-down in x-y relative to the other courier
        # since it is mounted on the opposite side
        self.setHome(1, 1, 0.0, 0.0)

        manip_intf = self.manip.getInterface()
        self.manip_object = self.getProgramObject(manip_intf)

        self.do_operation("Element5Grab", [  self.vacuum_pickup_tool_lens_el5,
                                             self.element_5 ],
                          "element_5_fixture", "Element5Return")

        self.do_operation("Ring5Grab", [ self.ringtool_5,
                                         self.lock_ring_el_5 ],
                          "ring_5_fixture", "Ring5Return")

        self.do_operation("Element34Grab", [  self.vacuum_pickup_tool_lens_el3_4,
                                              self.element_3_4_doublet ],
                          "element_34_fixture", "Element34Return")

        self.do_operation("Ring34Grab", [ self.ringtool_3,
                                          self.lock_ring_el_3 ],
                          "ring_34_fixture", "Ring34Return")

        self.do_operation("Element2Grab", [  self.vacuum_pickup_tool_lens_el2,
                                             self.element_2 ],
                          "element_2_fixture", "Element2Return")

        self.do_operation("SpacerGrab", [ self.ringtool_2,
                                          self.spacer_el_1_2 ],
                          "spacer_fixture", "SpacerReturn")

        self.do_operation("Element1Grab", [  self.vacuum_pickup_tool_lens_el1,
```

```
                                            self.element_1 ],
                          "element_1_fixture", "Element1Return")

        self.do_operation("Ring1Grab", [ self.ringtool_1,
                                         self.lock_ring_el_1 ],
                          "ring_1_fixture", "Ring1Return")

        print "Sleeping"
        while 1:
            self.sleep(0.2)

    def do_operation(self, grab_rendezvous, products, fixture_name,
                     return_rendezvous):
        print "Initiating"
        self.initiateRendezvous(self.manip, grab_rendezvous)
        self.presentedProducts = products
        print "Coordinating"
        fixture = FoDescriptionPtr(self.description.get(fixture_name))
        mat = fixture.getLocalMatrix()
        self.coordinateTo(self.manip_object, mat[3][0], mat[3][1])
        print "Finishing"
        self.coordinateTo(self.manip_object, -150, 0)
        print "Cleared"

        self.finishRendezvous(grab_rendezvous)

        self.initiateRendezvous(self.manip, return_rendezvous)

        print "Coordinating"
        fixture = FoDescriptionPtr(self.description.get(fixture_name))
        mat = fixture.getLocalMatrix()
        self.coordinateTo(self.manip_object, mat[3][0], mat[3][1])

        self.finishRendezvous(return_rendezvous)



program = Program()
}
            member home {
                matrix [ 1 0 0 0 0 1 0 0 0 0 1 0 -450.005 163.957 14.9999 1 ]
                children {
                }
            }
            member motor {
                matrix [ 1 0 0 0 0 1 0 0 0 0 1 0 156.115 441.65 -1.90735e-05 1 ]
                children {
                }
            }
        }
        file bridge.aaa {
            matrix [ 1 0 0 0 0 1 0 0 0 0 1 0 0 147.233 920 1 ]
            member crossbar {
                matrix [ 1 0 0 0 0 1 1.46125e-15 0 0 -1.46099e-15 1 0 -3.57628e-05
1.89761e-06 96.0405 1 ]
                children {
                    cached puma ipt:englishhorn.msl.ri.cmu.edu|1389,interface {
                        name M1
                        matrix [ -0.999994 2.16463e-09 1.54238e-16 0 4.23855e-08 -
0.999993 6.98763e-22 0 5.03578e-14 -1.82624e-13 1 0 4.10435 -39.9999 9.53675e-07 1
]
                        program {
from OSTIManipProgram import OSTIManipProgram

program = OSTIManipProgram()
}
                        member effectorLink {
                            children {
                            }
```

```
                        }
                        member base {
                            matrix [ 1 0 0 0 0 1 0 0 0 0 1 0 0 0 100 1 ]
                            children {
                            }
                        }
                    }
                }
            }
        }
    }
}
file outercornercurb.aaa {
    matrix [ -8.84336e-08 1 2.82723e-15 0 -1 -8.83553e-08 3.40823e-08 0 3.40823e-08
-4.51566e-12 1 0 -297 -600 577.25 1 ]
}
file outercornercurb.aaa {
    matrix [ -1 1.5353e-09 4.59664e-17 0 5.53562e-10 -1 7.18114e-15 0 1.56768e-16 -
2.12542e-14 1 0 300 -596.5 577.25 1 ]
}
file outercornercurb.aaa {
    matrix [ 1 0 0 0 0 1 0 0 0 0 1 0 -300 596.5 577.25 1 ]
}
file outercornercurb.aaa {
    matrix [ 9.05718e-08 -1 -1.236e-15 0 1 9.57624e-08 3.41116e-08 0 -3.42113e-08
3.87845e-09 1 0 297 600 577.25 1 ]
}
file shortcurb.aaa {
    matrix [ 1 0 0 0 0 1 0 0 0 0 1 0 0 -603.5 577.25 1 ]
}
file shortcurb.aaa {
    matrix [ -1 -2.92803e-07 -6.61493e-09 0 2.14959e-10 -0.999999 -1.14458e-15 0 -
1.86728e-08 -2.04577e-08 0.999999 0 -1.34161e-05 603.5 577.25 1 ]
}
product element_5_proto.aaa
product ringtool_3_proto.aaa
product vacuum_pickup_tool_lens_el3_4_proto.aaa
product lock_ring_el_5_proto.aaa
product collimatorhalf_proto.aaa
product ringtool_2_proto.aaa
product spacer_el_1_2_ver2_proto.aaa
product lock_ring_el_1_proto.aaa
product element_1_proto.aaa
product lock_ring_el_3_proto.aaa
product vacuum_pickup_tool_lens_el1_proto.aaa
product ringtool_5_proto.aaa
product element_3_4_doublet_proto.aaa
product element_2_proto.aaa
product vacuum_pickup_tool_lens_el2_proto.aaa
product vacuum_pickup_tool_lens_el5_proto.aaa
product ringtool_1_proto.aaa
```

B ZYVEX Files

B.1 zyvec_endeffector.aaa

```
Effector {
   view InventorView {
             body {
               File {
                 name pureendeffector.iv
               }
             }
   }

      grippers (
                Link {
                     num <Int> 2
                     matrix [1, 0, 0, 0,
                             0, 1, 0, 0,
                             0, 0, 1, 0,
                             0, 100.5961, -79.072, 1]

                     mount {{0,0,-1}, {0,0,1}}
      }
      )
}
```

B.2 zyvex_manipulator.aaa

```
file common_zyvex_manip.aaa {

     image file Zyvex_manip.gif

     effector file test_endeffector.aaa {}

     arc Component {
        view InventorView {
           body {
            File {
               name arcm.iv
             }
           }
      }
     }
     carriage Component {
        view InventorView {
          body {
           File {
            name simpel_carriage_5.iv
           }
          }
        }
       matrix [1, 0, 0, 0,
               0, 1, 0, 0,
               0, 0, 1, 0,
               0, 100.5815, -78.9458, 1]
     }

     interface ProgZYVEXManipulatorInterface {
     }
}
```

B.3 FoZYVEXManipulatorDesc.h

```
//////////////////////////////////////////////////////////////////////
//
//                      FoZYVEXManipulatorDesc.h
//
// Define the FOZYVEXManipulator agent
//
// Classes define for export:
//   FoZYVEXManipulatorDesc
//
//////////////////////////////////////////////////////////////////////

#ifndef fo_zyvex_manipulator_h
#define fo_zyvex_manipulator_h

#include <AAA/descriptions/FoComponents.h>
#include <AAA/descriptions/FoManipulatorDesc.h>
#include <AAA/descriptions/FoCourierDesc.h>


//class FoReservedDesc;

class FoZYVEXManipulatorDesc : public FoManipulatorDesc {
    FO_DESCRIPTION_HEADER(FoZYVEXManipulatorDesc);

  public:
    FoZYVEXManipulatorDesc();

    FoFloatField range;             // range of movement of the carriage
    FoCargoMountPointField arcMount; // where to mount the arc
    FoDescriptionField arc;          // runner of the carriage
    FoDescriptionField carriage;     // the moving carriage

    //    virtual FbBool moveMember(FoComponentDesc*);

    static void initClass();

  private:
    void setInterface(FoInterfaceBase*, FoInterfaceBase*);
};

#endif
```

B.4 FoZYVEXManipulatorDesc.cc

```
////////////////////////////////////////////////////////////////////////
//
//                          FoZYVEXManipulatorDesc.cc
//
// Implement the FoZYVEXManipulatorDesc
//
// Classes implemented for export:
//   FoZYVEXManipulatorDesc
//
////////////////////////////////////////////////////////////////////////

#include <AAA/descriptions/FoZYVEXManipulatorDesc.h>

FO_DESCRIPTION_SOURCE(FoZYVEXManipulatorDesc);

void FoZYVEXManipulatorDesc::initClass()
{
    FO_DESCRIPTION_INIT_CLASS(FoZYVEXManipulatorDesc, "ZYVEXManipulator",
"Manipulator");
}

FoZYVEXManipulatorDesc::FoZYVEXManipulatorDesc()
{
   FO_DESCRIPTION_CONSTRUCTOR(FoZYVEXManipulatorDesc);
   FO_ADD_FIELD(range, 1.5707);
   FO_ADD_FIELD(arcMount, FbCargoMountPoint());
   FO_ADD_SUBMEMBER_FIELD(arc, effector);
   arc.setAttribute(FO_FIELD_NO_WRITING);
   FO_ADD_SUBMEMBER_FIELD(carriage, arc);
}
```

B.5 FoProgZYVEXManipulatorInterface.h

```
////////////////////////////////////////////////////////////////////////////
//
//                          FoProgZYVEXManipulatorInterface3.h
//
// Defines the interface to a simulated 3-axis manipulator agent which can be
// directed by a python script
//
// Classes defined for export:
//  FoProgZYVEXManipulatorInterface - the simulated, programmable manipulator in-
//  terface
//
// Classes defined for internal use:
//  FoComponentDesc
//  CarriagePointList
//
////////////////////////////////////////////////////////////////////////////

#ifndef fo_prog_zyvex_manipulator_interface_h
#define fo_prog_zyvex_manipulator_interface_h

#include "FoProgManipulatorInterface.h"

#include <AAA/FbTime.h>

class FoComponentDesc;
class CarriagePointList;
```

```
class FoProgZYVEXManipulatorInterface : public FoProgManipulatorInterface {
         FO_BASE_HEADER(FoProgZYVEXManipulatorInterface);

  public:
    FoProgZYVEXManipulatorInterface();
    virtual ~FoProgZYVEXManipulatorInterface();

    FoIntField isMoving;       // 1 if moving
    FoFloatField carrMaxSpeed;  //maximum travel speed of the carriage

    virtual void update();

    void moveCarriage(float r, float speed, FoAction* = NULL);

    virtual FbBool moving() const;

    void carriageGet(float&);
    void carriageSet(float);

    static void initClass();

  private:
    void setCompletionAction(FoAction*);
    FbBool startCarriage(float);
    void move_carriage(const char*);
    void grasp(const char* params);
    void drop(const char* params);

    CarriagePointList* _carr_traj; // moving position list of carriage
    FbBool _carr_moving;           // true if carriage is moving
    float _carr_v;                 // current moving velocity
    float _carr_r;                 // last "fixed" position
    FbTime _carr_basetime;         // time that begin tracking
    float _carr_steptime;          // forwardtime from current position to next
                                   // point
    int _max_capacity;
};
#endif
```

B.6 FoProgZYVEXManipulatorInterface.cc

```
////////////////////////////////////////////////////////////////////////////////
////////
//
//                     FoProgZYVEXManipulatorInterface3.cc
//
// Implements the interface to a simulates xouroer agent whoch can be
// directed by a physon script
//
// Classes implemented for export:
//  FoProgZYVEXManipulatorInterface - the simulated, programmable manip. interface
//
// Classes implemented for internal use:
//  CarriagePoint - a point tracked by the simulated ZYVEX Manipulator's carriage
//  CarriagepointList - A class for the points being tracked by a simulated
//                       ZYVEX Manipulator's carriage
//
////////////////////////////////////////////////////////////////////////////////

#include "FoProgZYVEXManipulatorInterface.h"

#include <AAA/AAA.h>
#include <AAA/descriptions/FoComponents.h>
#include <AAA/descriptions/FoZYVEXManipulatorDesc.h>
```

```
#include <math.h>
#include <AAA/FbLinear.h>


struct CarriagePoint {
    FbVec2f point;        // the point to track (r,v)
    FoAction* action;     // the action to take when we get to this point
    CarriagePoint* next;  // the next point in the list
};

// Aclass for a FIFO queue of the points for carriage tracking to update
class CarriagePointList {
   public:
     CarriagePointList() { _head = _tail = NULL; }
     ~CarriagePointList();

     CarriagePoint* pop();                          // pop a point off the head
     void append(FbVec2f pt, FoAction* act);        // add a point to the tail
     CarriagePoint* head() const { return _head; } // return whats at the head

  private:
     CarriagePoint* _head; // head of the point list
     CarriagePoint* _tail; // tail of the point list
};

//destroy a point list
CarriagePointList::~CarriagePointList()
{
    CarriagePoint* elem = _head;
    CarriagePoint* doomed;
    while (elem) {
          doomed = elem;
          if (elem->action)
              elem->action->unref();
          elem = elem->next;
          delete doomed;
    }
}

//pop a point of the front of the list. If the list is empty, return NULL
CarriagePoint* CarriagePointList::pop()
{
    if (!_head)
        return NULL;

    CarriagePoint* res = _head;
    _head = _head->next;
    if (!_head)
        _tail = NULL;

    return res;
}

// append (r,v) and corresponding action to the end of the point list
void CarriagePointList::append(FbVec2f pt, FoAction* act)
{
     CarriagePoint* elem = new CarriagePoint;
     elem->point = pt;
     elem->action = act;
     if (act)
         elem->action->ref();
     elem->next = NULL;
     if (_tail)
         _tail->next = elem;
     else
         _head = elem;
     _tail = elem;
}
```

```
FO_BASE_SOURCE(FoProgZYVEXManipulatorInterface);

void FoProgZYVEXManipulatorInterface::initClass()
{
   FO_BASE_INIT_CLASS(FoProgZYVEXManipulatorInterface,
                      "ProgZYVEXManipulatorInterface", "ProgManipulatorInterface");
}


////////////////////////////////////////////////////////////////////////////////


FoProgZYVEXManipulatorInterface::FoProgZYVEXManipulatorInterface()
{
   FO_BASE_CONSTRUCTOR(FoProgZYVEXManipulatorInterface);

   // set up carriage variables
   _carr_traj = new CarriagePointList;
   _carr_moving = FALSE;
   _carr_r = 0; //11.26

//   _max_capacity = 0;

     FoFieldRestrictions* res = exportField("isMoving"); // 30.11

     FO_ADD_FIELD(carrMaxSpeed, 2*M_PI);
     res = exportField("carrMaxSpeed");
     res->writable.setValue(TRUE);

    registerAction("grasp",
                   new FoClassProgAction<FoProgZYVEXManipulatorInterface>
                   (this, &FoProgZYVEXManipulatorInterface::grasp));
    registerAction("drop",
                   new FoClassProgAction<FoProgZYVEXManipulatorInterface>
                   (this, &FoProgZYVEXManipulatorInterface::drop));
   registerAction("moveCarr",
                  new FoClassProgAction<FoProgZYVEXManipulatorInterface>
                  (this, &FoProgZYVEXManipulatorInterface::move_carriage));
}

FoProgZYVEXManipulatorInterface::~FoProgZYVEXManipulatorInterface()
{

    CarriagePoint* point;
    while (point=_carr_traj->pop()) {
        if (point->action)
            point->action->unref();
        delete point;
    }
    delete _carr_traj;
}

#define ABS(x) (((x) > 0) ? (x) : -(x))

// start the trajectory stored in the _points member variable, considering
// that we should have started if head_start seconds ago. returns TRUE
// if the trajectory is started successfully
FbBool FoProgZYVEXManipulatorInterface::startCarriage(float head_start)
{
     // if no point to track, return FALSE
     CarriagePoint* target = _carr_traj->head();
     if (!target)
         return FALSE;

     // set the time at which we should have started  tracking
     _carr_basetime = FbTime::getTimeOfDay() - FbTime(head_start);

    // figure how far we have to go to the next point
    float dest_r, speed_factor;
    _carr_traj->head()->point.getValue(dest_r, speed_factor);
```

```
    while (dest_r > M_PI)
        dest_r -= 2*M_PI;
    while (dest_r < -M_PI)
        dest_r += 2*M_PI;
    float dr = dest_r - _carr_r;
    _carr_traj->head()->point[1] =dest_r; ///eventuell [0]

    // set r velocitie based on the longer travel time to achieve
    // the desired r at the given speed
    float rot_time = ABS(dr)/(carrMaxSpeed.getValue()*speed_factor);
    if (rot_time < 0.00001) {
        _carr_steptime = 0;
        _carr_v = 0;
    } else { // get there
        _carr_v = dr/rot_time;
        _carr_steptime = rot_time;
    }

    return TRUE;
}

// move the carriage to r at speed percentage of maximum.
// invoke act when we are finished
void FoProgZYVEXManipulatorInterface::moveCarriage(float displacement,  // neces-
sary?
                                                   float speed, FoAction* act)
{
   FbVec2f pt;
   pt[0] = displacement;
   pt[1] = speed;
   _carr_traj->append(pt, act);

   if (!_carr_moving) {
       carriageGet(_carr_r);
       startCarriage(0);
       _carr_moving = TRUE;
   }
}

// set the carriage to displacement r

void FoProgZYVEXManipulatorInterface::carriageSet(float r)
{
    if (!getDescription()->isOfType(FoZYVEXManipulatorDesc::getClassTypeId()))
        return;
    FoZYVEXManipulatorDesc* manipulator = (FoZYVEXManipulatorDesc*) getDescrip-
tion();

    FoDescription* carriage = manipulator->carriage.getValue();
    if (!carriage)
       return;
    FoDescription* base = manipulator->arc.getValue();

    FbVec3f t;
    FbRotation o;
    carriage->matrix.getValue().getTransform(t, o);



    carriage->setPositiony(0,100.5815,-78.9458,r); // new function: setPositiony

        if (!_carr_moving) {
           _carr_r = r;
          return;
        }
}
// return TRUE if the effector is currently moving
FbBool FoProgZYVEXManipulatorInterface::moving() const
```

```
{
     return _carr_moving;
}



// get the current position of the simulated carriage
void FoProgZYVEXManipulatorInterface::carriageGet(float& r_out)
{
     if (!_carr_moving) {
         // if we are not moving, use the last position for where the effector is
         r_out = _carr_r;
         return;

     }
     // we are moving, so we have to figure out where the effector should be,
     // not just where it is

     // take first guess at where the effector should be, given the velocitie and
     // the elapsed time
     FbTime current = FbTime::getTimeOfDay();
     float elapsed = (current - _carr_basetime).getValue();

     r_out = _carr_r + _carr_v*elapsed;

     // that guess will be wrong if the elapsed time is longer than the time
     // we calculated it should take to get to the point we are aiming at

     FoAction* action = NULL;
     int reget = 0;
     if (elapsed > _carr_steptime) {
        // so, get the next point
        CarriagePoint* target = _carr_traj->pop();
        // set our position to be at that point
        _carr_r = target->point[0];  //evetuell [1]
        if (!_carr_traj->head()) {
           // if we are done with all points, stop the carriage
           r_out = _carr_r;
           _carr_moving = FALSE;
        } else {
           // else start the trajectory, taking into account we have
           // overshot by a little bit
           startCarriage(elapsed - _carr_steptime);
           reget = 1;
        }
        action = target->action;
        delete target;
     }

     // set the effector's position to the proper value
     while (r_out > M_PI)
         r_out -= 2*M_PI;
     while (r_out < -M_PI)
         r_out += 2*M_PI;
     carriageSet(r_out);

     if (action) { // if we need to execute an action
         action->execute();
         action->unref();
     }
     if (reget) // we do this here if we have just started a trajectory
        carriageGet(r_out);
}

//update manipulator
void FoProgZYVEXManipulatorInterface::update()
{
     // update the manipulator interfacce
     FoProgManipulatorInterface::update();
```

```
     // update the carriage position
     if (!_carr_moving)
        return;
     float r;
     carriageGet(r);
     carriageSet(r);
}


// Action for the "moveCarr" action tag
// moveCarr r speed blocking
// Move the carriage to r
// at speed percentage of maximum. unblock at completion if blocking is 1
void FoProgZYVEXManipulatorInterface::move_carriage(const char* params)
{
    float r, speed;
    int blocking;

    if (sscanf(params, "%f %f %d" , &r, &speed, &blocking) != 3) {
        printf("Programmed ZYVEX manipulator syntax error on '%s'\n", params);
        return;
    }

    FoAction* action;
    if (blocking)
       action = new FoUnblockAction(this);
    else
       action = NULL;
    moveCarriage(r, speed, action);
}

// action for the "grasp" action tag
//  grasp depth
// Transfer the reference of the object at depth from the motor of the
// partnered courier to the gripper
void FoProgZYVEXManipulatorInterface::grasp(const char* params)
{
    int depth;
    char fixture_name[100];

    printf("ZYVEX grasp %s\n", params);

    if (!getDescription()->isOfType(FoManipulatorDesc::getClassTypeId())) {
        printf("%s: Programmed manipulator not owned by a manipulator\n",
               getDescription()->getName().getString());
        return;
    }

    if (sscanf(params, "%d %s", &depth, &fixture_name[0]) != 2) {
        printf("%s: Progammed manipulator syntax error on '%s'\n",
               getDescription()->getName().getString(), params);
        return;
    }

    FoCourierDesc* courier;
    FoDescription* obj = getChild(getPartnerName(),
                                  depth, fixture_name, courier);
    if (!obj)
      return;

    FoZYVEXManipulatorDesc* manipulator = (FoZYVEXManipulatorDesc*) getDescrip-
tion();
    FoDescription* gripper=manipulator->carriage.getValue();
    if (!gripper) {
        printf("%s: EEK no carriage for ZYVEX manipulator!",
               getDescription()->getName().getString());
        return;
    }
```

```
    obj->transferReference(gripper);

    unblock();
}

// action for the "drop" action tag
// drop depth
// Transfer the reference of the object on the endeffector to the object
// at depth from the motor of the partnered courier
void FoProgZYVEXManipulatorInterface::drop(const char* params)
{
    int depth;

    printf("ZYVEX drop %s\n", params);

    if (!getDescription()->isOfType(FoManipulatorDesc::getClassTypeId())) {
        printf("%s: Programmed manipulator not owned by a manipulator\n",
               getDescription()->getName().getString());
        return;
    }

    char fixture_name[100];
    if (sscanf(params, "%d %s", &depth, &fixture_name[0]) != 2) {
        printf("%s: Progammed manipulator syntax error on '%s'\n",
               getDescription()->getName().getString(), params);
        return;
    }

    FoCourierDesc* courier;
    FoDescription* obj = getChild(getPartnerName(),
                                  depth, fixture_name, courier);
    if (!obj)
      return;

    FoZYVEXManipulatorDesc* manipulator = (FoZYVEXManipulatorDesc*) getDescrip-
tion();
    FoDescription* gripper=manipulator->carriage.getValue();
    if (!gripper) {
        printf("%s: EEK no carriage for ZYVEX manipulator!",
               getDescription()->getName().getString());
        return;
    }

    int index = gripper->numChildren()-1;
    if (gripper->getChild(index))
        gripper->getChild(index)->transferReference(obj);

    unblock();
}

////////////////////////////////////////////////////////////////////
static FoDescription* get_gripper(FoZYVEXManipulatorDesc* zmanip)
{
    FoDescription* carriage = zmanip->carriage.getValue();
    if (!carriage) {
        return zmanip->arc.getValue();
    }

    FoDescriptionsField* grippers_field =
        FO_GET_FIELD(carriage, "grippers", Descriptions);
    if (!grippers_field)
         return NULL;
    FbList<FoDescription*>* grippers = grippers_field->getValue();
    if (!grippers || !grippers->numElems())
         return NULL;
    return (*grippers)[0];
}

static FbBool get_gripper_point(FoZYVEXManipulatorDesc* zmanip, FbVec3f& point)
{
```

```
    FoDescription* carriage = zmanip->carriage.getValue();
    FoDescription* link = zmanip->arc.getValue();
    if (!link)
        return FALSE;
    FoCargoMountPointField* mount_field =
        FO_GET_FIELD(link, "mount", CargoMountPoint);
    FbVec3f link_mount = mount_field->getValue().point;
    if (!mount_field)
        return FALSE;
    if (!carriage) {
        point = mount_field->getValue().point;
        return TRUE;
    }

    FoDescriptionsField* grippers_field =
        FO_GET_FIELD(carriage, "grippers", Descriptions);
    if (!grippers_field)
        return FALSE;
    FbList<FoDescription*>* grippers = grippers_field->getValue();
        return FALSE;
    FoDescription* gripper = (*grippers)[0];
    if (!gripper)
        return FALSE;
    mount_field = FO_GET_FIELD(gripper, "mount", CargoMountPoint);
    if (!mount_field)
        return FALSE;
    point = mount_field->getValue().point;
    FbVec3f gripper_trans;
    FbRotation rot;
    gripper->getMatrix().setTransform(gripper_trans, rot);
    point += gripper_trans;
    point += link_mount;
    return TRUE;
}
```

B.7 ProgZYVEXManipulator.py

```
import progInterface
from progManipulator import*

def moveCarr(id, r, speed):
    progInterface.send_action(id, 'moveCarr %f %f 1' % (r, speed))
    progInterface.block(id)

def grasp(id, depth = 1, fixture = "None"):
    progInterface.send_action(id, 'grasp %d %s' % (depth, fixture))
    progInterface.block(id)

def drop(id, depth = 1, fixture = "None"):
    progInterface.send_action(id, 'drop %d %s' % (depth, fixture))
    progInterface.block(id)
```

B.8 zyvex.fac

```
file base_frame.aaa {
    children {
        file lg_platen.aaa {
            name P1
            matrix [ 1 0 0 0 0 1 -3.3304e-16 0 0 3.33081e-16 1 0 0 0 570 1 ]
```

```
        }
        file bridge.aaa {
            matrix [ 1 0 0 0 0 1 0 0 0 0 1 0 0 -0.0140262 920 1 ]
            member crossbar {
                children {

######################### Manipulator ########################

                    file test_manip.aaa {
                        name M1
                        matrix [ -0.99995 1.01418e-15 2.51476e-13 0 -2.36965e-16 -
0.99995 4.25452e-17 0 -2.45819e-13 -2.22741e-12 1 0 -24.9795 -39.999 -4.76879e-07 1
]
                        program {from progZYVEXManipulator import*
m=['mi_1', 'mi_2', 'mi_3', 'mi_4', 'mi_5', 'mi_6', 'mi_7', 'mi_8', 'mi_9', 'mi_10',
'mi_11', 'mi_12', 'mi_13', 'mi_14', 'mi_15', 'mi_16','mi_17', 'mi_18', 'mi_19',
'mi_20', 'mi_21', 'mi_22', 'mi_23', 'mi_24', 'mi_25', 'mi_26', 'mi_27', 'mi_28',
'mi_29', 'mi_30', 'mi_31', 'mi_32']
s=['so_1', 'so_2', 'so_3', 'so_4', 'so_5', 'so_6', 'so_7', 'so_8', 'so_9', 'so_10',
'so_11', 'so_12', 'so_13', 'so_14', 'so_15', 'so_16', 'so_17', 'so_18', 'so_19',
'so_20', 'so_21', 'so_22', 'so_23', 'so_24', 'so_25', 'so_26', 'so_27', 'so_28',
'so_29', 'so_30', 'so_31', 'so_32']
j=10
#while 1:

for i in range(32):

 if (i>=0 and i<10) or (i>19 and i<30) or (i>40 and i<50) or (i>60 and i<70) or
(i>80 and i<90):
  acceptRendevous(self, "C1_M1")
  moveCarr(self,0.78535,1)
  coordMoveTo(self,0, -0.780 ,0 , 2.5, 0, 0.5, m[i])
  coordMoveTo(self,0, -0.780, 0, 1.15, 0, 0.01, m[i])
  coordMoveTo(self,0, -0.780, 0, 1.5, 0, 0.01, m[i])
  coordMoveTo(self,0, 0, 0, 1.5, 0, 0.01, m[i])
  coordMoveTo(self,0, 0, 0, 1.15, 0, 0.01, m[i])
  grasp(self, 0, m[i])
  coordMoveTo(self,0, 0, 0, 1.15, 0, 0.01, m[i])
  coordMoveTo(self,0, 0, 0, 2.5, 0, 0.5, m[i])
  movePartnerTo(self,"C1",-250,-400,1)
  moveCarr(self, -0.78535, 0.5)
  finishRendevous(self)

 else:
  acceptRendevous(self, "C1_M1")
  moveCarr(self,0.78535,1)
  moveTo(self, 20, 3.14159)
  coordMoveTo(self,0, 0.780 ,0 , 2.5, 3.14159, 0.5, m[i])
  coordMoveTo(self,0, 0.780, 0, 1.15, 3.14159, 0.01, m[i])
  coordMoveTo(self,0, 0.780, 0, 1.5, 3.14159, 0.01, m[i])
  coordMoveTo(self,0, 0, 0, 1.5, 3.14159, 0.01, m[i])
  coordMoveTo(self,0, 0, 0, 1.15, 3.14159, 0.01, m[i])
  grasp(self, 0, m[i])
  coordMoveTo(self,0, 0, 0, 1.15, 3.14159, 0.01, m[i])
  coordMoveTo(self,0, 0, 0, 2.5, 3.14159, 0.5, m[i])
  movePartnerTo(self,"C1",-250,-400,1)
  moveCarr(self, -0.78535, 0.5)
  finishRendevous(self)

 if i>15:
  acceptRendevous(self, "C2_M1")
  moveTo(self, 20, -1.5707, 1)
  coordMoveTo(self,0, 0, 0, 2.5, -1.5707, 0.5, s[i])
  coordMoveTo(self,0, 0, 0.158, 1.52, -1.5707, 0.01, s[i])
  drop(self, 0, s[i])
  coordMoveTo(self,0, 0, 0.158, 1.52, -1.5707, 0.01, s[i])
  coordMoveTo(self,0, 0, 0.158, 1.2, -1.5707, 0.01, s[i])
  coordMoveTo(self,0, 0, 0.3, 1.2, -1.5707, 0.01, s[i])
  coordMoveTo(self,0, 0, 0.3, 2.5, -1.5707, 0.5, s[i])
  movePartnerTo(self,"C2",-250,400,1)
```

```
  moveTo(self, 20, 0, 1)
  finishRendevous(self)

 else:
  acceptRendevous(self, "C2_M1")
  #moveTo(self, 20, 0 1)
  coordMoveTo(self,0, 0, 0, 2.5, 0, 0.5, s[i])
  coordMoveTo(self,0, -0.158, 0, 1.52, 0, 0.01, s[i])
  drop(self, 0, s[i])
  coordMoveTo(self,0, -0.158, 0, 1.52, 0, 0.01, s[i])
  coordMoveTo(self,0, -0.158, 0, 1.2, 0, 0.01, s[i])
  coordMoveTo(self,0, -0.3, 0, 1.2, 0, 0.01, s[i])
  coordMoveTo(self,0, -0.3, 0, 2.5, 0, 0.5, s[i])
  movePartnerTo(self,"C2",-250,400,1)
  moveTo(self, 20, 0, 1)
  finishRendevous(self)


}
                        member effectorLink {
                            children {
                            }
                        }
                        member base {
                            matrix [ 1 0 0 0 0 1 0 0 0 0 1 0 0 0 100 1 ]
                        }
                        member carriage {
                            matrix [ 1 0 0 0 0 1 0 0 0 0 1 0 0 100.581 -78.9458 1 ]
                        }
                    }
                }
            }
        }

######################### Courier 1 ########################

        file zyvex_courier_1.aaa {
            name C1
            matrix [ 1 0 0 0 0 1 0 0 0 0 1 0 450 0.120276 655 1 ]
            program {from progCourier import*

while 1:
 initiateRendevous(self,"M1","C1_M1")
 reserve(self,"M1")
 performRendevous(self)
 endRendevous(self)
 unreserve(self,"M1")
}
            member home {
                matrix [ 1 0 0 0 0 1 0 0 0 0 1 0 -450 -0.120276 15 1 ]
            }
            member motor {
                matrix [ 1 0 0 0 0 1 0 0 0 0 1 0 202.016 -466.439 0 1 ]
            }
        }

######################### Courier 2 ########################

        file zyvex_courier_2.aaa {
            name C2
            matrix [ -0.999998 8.6625e-07 -1.99703e-15 0 -1.0664e-07 -0.999998
2.43866e-07 0 3.79626e-07 1.86594e-07 0.999998 0 -450 -2.99364 655 1 ]
            program {from progCourier import*
while 1:
 initiateRendevous(self,"M1","C2_M1")
 reserve(self,"M1")
 performRendevous(self)
 endRendevous(self)
 unreserve(self,"M1")
}
```

```
            member home {
                matrix [ 1 0 0 0 0 1 0 0 0 0 1 0 -450.001 -2.99405 14.9998 1 ]
            }
            member motor {
                matrix [ 1 0 0 0 0 1 0 0 0 0 1 0 203.876 468.695 0.000125885 1 ]
            }
        }
    }
}
file outercornercurb.aaa {
    matrix [ 1 0 0 0 0 1 0 0 0 0 1 0 -300 596.5 577.25 1 ]
}
file outercornercurb.aaa {
    matrix [ 1.84613e-11 -1 -3.1916e-15 0 1 -4.10281e-10 -1.16662e-07 0 1.16662e-07
-8.34593e-11 1 0 297 600 577.25 1 ]
}
file outercornercurb.aaa {
    matrix [ 1.47346e-07 1 2.13361e-15 0 -1 1.29278e-07 3.32102e-08 0 3.32102e-08 -
1.38687e-14 1 0 -297 -600 577.25 1 ]
}
file outercornercurb.aaa {
    matrix [ -1 -3.79424e-09 -5.32004e-16 0 -2.4597e-09 -1 -5.46555e-15 0 1.47104e-
16 1.52341e-15 1 0 300 -596.5 577.25 1 ]
}
file shortcurb.aaa {
    matrix [ -1 -9.83008e-09 1.57065e-10 0 -3.01328e-10 -1 3.89389e-15 0 -2.28295e-
10 -6.9526e-12 1 0 -4.78195e-06 603.5 577.25 1 ]
}
file shortcurb.aaa {
    matrix [ 1 0 0 0 0 1 0 0 0 0 1 0 0 -603.5 577.25 1 ]
}
```

Eidesstattliche Erklärung

Ich erkläre hiermit eidesstattlich, dass ich die vorliegende Arbeit selbständig angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.
Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt.

Garching/Augsburg, den 15.03.2007

(Christoph Bergler)