MD-SIR: a methodology for developing sensor guided industry robots

D.I. Kosmopoulos , T.A. Varvarigou , D.M. Emiris , A.A. Kostas

Abstract

In this paper, we present a generic methodology for the synthesis of industrial robot applications with sensory feedback at the end-effector level. The presented methodology assumes an open controller architecture and leads to the creation of a library of modular and reusable entities, which can be used to build new systems based on the proposed architectural framework. The library facilitates the integration of new algorithms and it evolves as new modular applications are built. The library components belong to the control objects layer of the open controller architecture and implement functionality for sensor interfacing, sensor modeling, pattern recognition, state estimation and state regulation. The validity of the approach is verified by composing real industrial applications. The experimental results indicate the high quality of the developed systems.

Keywords: Open robot controller architecture; Sensor integration; Control objects library; Application development

1. Introduction

The current tendencies in the global economy and in the consuming behavior place new challenges in industry, concerning production line flexibility, production volume, quality and cost. The robots that employ sensory feedback at the end-effector level, are capable of adapting themselves to the environment or to manipulator uncertainties and they lend themselves as a feasible option for the fulfillment of those requirements.

Despite the undeniable advantages of sensor-guided robots, such installations are still the exception in industry, due to technology constrains and market policies. For many years the sensor technology has posed many restrictions (e.g. high noise, low acquisition speed, lack of simple drivers); the robustness of robot guidance algorithms was in most cases inadequate; the hardware performance was low and the software development methods were immature. In addition to the above, the robot constructors adopted for many years closed proprietary controller architectures, which made prohibitive the integration of external sensory feedback.

Recently, the above restrictions begun to be raised. The hardware platform of personal computers offers now big processing and networking capabilities, ease of use and maintenance and high communication capabilities at low cost. The modeling languages (e.g. UML), the integrated programming environments, the standard libraries (e.g. STL) and the open integration platforms (e.g. CORBA [6]) facilitate the development of robotic applications. Moreover, the sensors technology has abandoned many restrictions of the past and sensor employment is now possible with a relatively simple environment configuration [15]. Apart from the above, lately many robot constructors adopted a more open architecture for their controllers. Thus, it becomes obvious that the present political and technical environment favors the development of a software methodology for sensor integration in robotic systems. The need for

such a methodology is dictated by the complexity of the domain, the complexity of the available technology and the difficulty in managing the development process.

The rest of the paper is structured as follows: in the following section we will define the context of this research; in Section 3 we will present an overview of the MD-SIR methodology for integrating sensors in robotic systems; in Section 4 we will demonstrate the validity of the approach by presenting the implementation of a pilot sunroof fitting application; in Section 5 the implementation of a gap-measuring system is presented following the same approach; the paper ends with Section 6, which presents the conclusions and the future work.

2. Research context and previous work

The literature regarding integration of sensory feedback in robots is quite rich. However, a holistic approach in developing software for applications with sensory feedback at the end-effector level is still missing. Before analyzing the methodology we present the context of our research and the related work. Fig. 1 illustrates the architecture of a typical robotic application. The left part is adapted from [26], where the levels of *organization*, *processing* and *execution* are presented according to the intelligence of the included components (intelligence increases from bottom to top). The organization level includes the procedure manager that defines the robotic tasks execution sequence (usually by using a robot programming language), the knowledge base that includes the system's knowledge about the environment and the user interface. The processing level includes the controller, which defines the robot motion by using the processed input from internal and external sensors. The execution level includes the actuators, which perform the requested tasks and the sensors that provide raw sensory data to the system. The right part in Fig. 1 presents the layered architecture of the open robot controllers as presented in OSACA [22]. The upper layers use the services provided by the lower ones. The *hardware* layer includes the processing units that are used for the software execution. The *operating system* controls and synchronizes the use of the resources and acts as the intermediate layer between the hardware and the application software. The *communication* implements the information exchange between the subsystems. The *configuration* allows the installation and parameterization of the desired subsystems. The *control objects* are the subsystems that provide the continuous processing functionality in the application. They are interfaced to the software and hardware system layers through the *software*- and *hardware-API* layer, which should provide plug and play capabilities.

This work focuses on the control objects layer, which offers its services independently of the organization, execution and its underlying layers with regard to the open controller architecture. The control objects are usually associated with a particular sensor or actuator type or they can just provide services based on input from other control objects. The sequence of the provided services is defined by the organization level (procedure manager, user interface) and is influenced by the environment.

The research at the control objects layer has been quite active. The OROCOS project [23] aims in the development of open robot controllers and until now it has issued many useful general guidelines. The GenoM [16] has presented a methodology for the definition of the parts of the control objects that perform event-based state transition. It has described the states for the standard control objects and has separated the state control from continuous processing.

For the development of the subsystems of the controller development, environments such as ORCCAD [4] have been proposed. For the implementation of control objects that are able to perform robotic tasks (elementary actions) the ESTEREL language [2] has been used, while the robotic procedures have been defined using the MAESTRO language [7]. Due to the

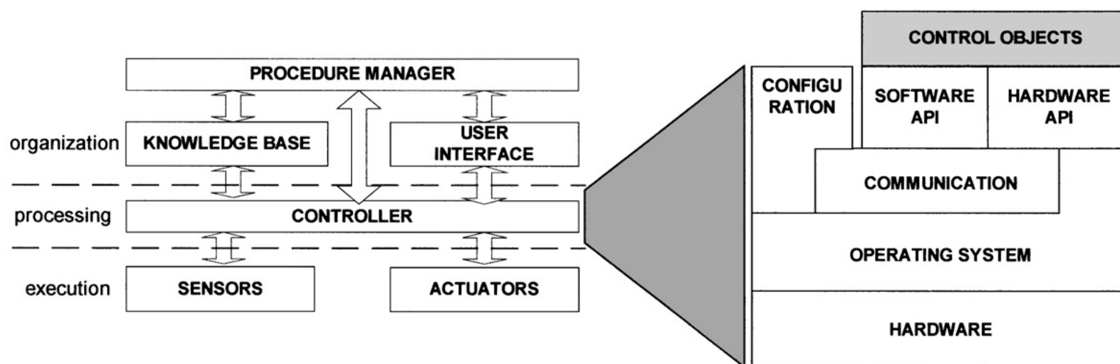

Fig. 1. Typical layered architecture of robotic applications. The organizational level includes the procedure manager, the knowledge base and the user interface; the execution level includes the sensors and the actuators; the processing level includes the robot controller, which may be analyzed to the open architecture layers (see [22]).

use of formal languages the testing at the organization level has become possible, while at the processing level only simple control laws have been able to be tested assuming that the state error is available. Other commercial environments that are based on similar principles are the ControlShell [21] and the GENERIS [19].

Many other attempts have been made to implement open control objects such as the SMART [1], the START [18] and the OSCAR [12], however, without much emphasis on sensor integration. The ViSP [17] has presented an object-oriented library of classes that are used for visual servoing using a single camera mounted on a robot arm. The systems XVision [8] and Servomatic [24] have been combined to guide robots using cameras. Servomatic has been able to use the measurements coming from systems using pattern recognition libraries like XVision in order to guide a robot by using elementary tasks such as fitting point-to-point, point-to-line, line-to-point and line-to-line. However, the approach is valid only for image-based endpoint-closed-loop systems [10]. As regards the control objects that perform pattern recognition in the sensory data, there is plenty of open source initiatives or commercial products, mainly focusing on vision, such as the TargetJr [27], the Open CV [11], the Halcon [20], etc.

Many of the aforementioned approaches have great scientific value but cannot be directly applied in industrial applications due to the fact that most robot controllers restrict the access at the joint level. Furthermore, many of the above approaches are limited to specific sensors or control methods at the end-effector level. Currently, many robot manufacturers endow their controllers with an increasing "openness" especially regarding feeding sensory data into the trajectory generation. Therefore, the need for a generic methodology for developing software for sensor-based robot guidance that considers the commercial controller restrictions becomes obvious. The presented work capitalizes on the increased "openness" of modern robot controllers, by extending the capabilities provided by the manufacturer, through intelligent handling and fusion of acquired information. The proposed approach constitutes a modular and systematic (algorithmic) way in order to encompass different types of sensory input; obviously, this approach can be fully profitable, when the control system architecture allows for this information exchange to take place. More specifically, in this research we present the MD-SIR methodology (Methodology for Developing Sensor-guided Industry Robots) which offers:

- An architecture that enables the easy sensor integration into the controllers of open (or semi-open) architecture. The architecture is generic enough to facilitate the use of the most commonly used sensors and the employment of various control methods. Furthermore, it is modular, extensible and minimizes the coupling between the components.
- A library of modular and reusable software components, at the subsystem or object level. By using the library units properly, the easy, reliable and cost-effective development of applications can be achieved.
- Possibility for a closer collaboration between research and industry. Through the extensibility of the library new algorithms can be easily integrated and applied in the production.
- Paradigms for the use of the library components along with sensors for the execution of the most typical robotic tasks.

The methodology is independent of the organizational and execution levels of the application and of the rest layers of the robot controller, which are assumed to be available. Furthermore, the methodology is not limited to specific control and pattern recognition algorithms.

3. Methodology overview

In the past many robot control schemes have been presented. However, the industrial robotic manipulators use invariably the PID controllers in their independent joint servo control systems. The PID controllers at the joint level are very simple and robust but are unable to deal with environment uncertainties and therefore external sensory feedback is required. By employing sensory feedback at the end-effector level the dynamics of the robot may often be decoupled (due to the high frequency joint control loops) and the manipulator may be treated as a nearly ideal positioning device.

Most open architecture robot controllers unlike typical closed architecture controllers, may permit the user or an external system to define precisely the interpolation points and oblige the end-effector to strictly pass through them. However, this restriction may produce oscillations at the end-effector level depending partially on the coarseness of the trajectory. These oscillations may be avoided if an additional regulator is used. The control scheme presented in Fig. 2 is then applied. The presented methodology proposes how to develop software that will undertake the robot regulation at the end-effector level by profiting of the sensory data.

The methodology leads to the creation of a *library* of modular *control objects* (subsystems that can be analyzed in reusable components), which are going to be applied in the framework of an *architecture* to compose robotic applications that employ sensory feedback. Vice versa, the composition of new reusable entities for the needs of new applications enriches the library. The

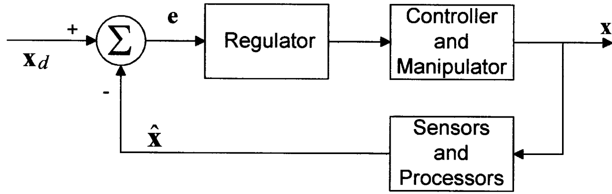

Fig. 2. Closed loop control scheme at the end-effector level. The end-effector's state x is measured by the sensing devices and the corresponding measurement $\hat{x}$ is given as feedback; then $\hat{x}$ is compared to the desired state x_d, the difference is fed to the regulator and the regulation output is sent to the robot controller to move the manipulator.

development is based on the open controller architecture and is driven by requirements in an iterative manner (each development cycle follows the requirements–analysis–design–implementation–testing pattern). The library entities are described by attributes, methods, technical characteristics, restrictions, rules of composition and interconnection, etc.

3.1. Requirements

The requirements presented here express the functionality of the typical robotic application. They are sets of robotic tasks (according to the terminology used in [4]) and include teaching, parameterization and execution.

- Teaching. Some sections of the robot trajectory can be executed without strict precision requirements. These trajectory sections must be pre-recorded during a procedure called "teaching". In the recorded trajectory program we can determine the interpolation nodes, at which tasks requiring sensory feedback will be executed.
- Parameterization. During parameterization the data needed for sensor-based control are defined and stored. These data are essential for the operation of system and they are retrieved each time a task with sensory feedback has to be executed. For each task the system must be parameterized separately. The parameterization may regard the end-effector, the sensor, the regulator and the processing. It may include training, during which system parameters are automatically calculated, e.g. inverse Jacobian matrices resulting from feature measurements while the robot executes movement at predetermined steps for each degree of freedom.
- Execution. The system executes a task in real time. The task includes movement with and without sensory feedback at the end-effector level. When the robot moves without sensory feedback it must read from a database the trajectory that has been stored during teaching. When sensory feedback is required the movement must be calculated online using the parameters that have been fixed during parameterization; the parameters are loaded from a database.

3.2. Architecture

The Controller presented in Fig. 1 may be further decomposed to the Data Processor, the Sensors' Controller and the Actuators' Controller at the control object layer (Fig. 3). The controllers of sensors and actuators include the vendor software that controls the sensors and the actuators by setting their desired state and receiving data from them (sensory data or current state). The Data Processor receives the sensory data from the sensors and communicates the desired state vector to the actuators. It may be decomposed into the subsystems of the *Actuator Manager* (*AM*), the *Sensor Manager* (*SM*), the *State Error Calculator* (*SEC*) and the *Sensor–Actuator Interface* (*SAI*).

The *AM* receives at each robot cycle the current end-effector state error and calculates the desired state using a control law (state regulation). The regulation takes place during execution, parameterization or teaching, in order to avoid the unwanted end-effector oscillations and the desired state is sent to the actuator. The robot interpolation cycle and the sensor cycle may differ significantly (multi-rate system); therefore, the robot needs to be coupled with the sensors by using the intermediate module *SAI*. *SAI* receives asynchronously the state error from the sensors' subsystem whenever a measurement is made available by *SEC*; *SAI* also forwards the state error synchronously to *AM* at each robot interpolation cycle.

The *SM* extracts the predefined features (patterns) from sensory data during execution or parameterization. It sends data requests to the sensors and receives the sensor data. It outputs to the *SEC* the feature measurements. The *SEC* receives the feature measurement from the *SM* along with the current actuator state from the *AM* through *SAI*. It outputs to *AM* through the *SAI* the current state error of the end-effector.

The attributes of the objects used by the subsystems may be stored in databases; the corresponding database instances will be referred as *AMDB*, *SAIDB*, *SMDB* and *SECDB* in the following (all abbreviations are presented in Appendix A). Some of the typical library classes that were used for implementing the subsystems of the sunroof fitting application (to be discussed in Section 4) are presented in Appendix B.

Each of the aforementioned subsystems operates within a loop in a system process. The scheme of each of those system processes is displayed in Table 1. During initialization the initial local parameters are loaded from the corresponding database and the appropriate memory blocks are initialized (2). The main loop (5–10) performs the processing work. At the beginning of the loop the data from other processes and the corresponding

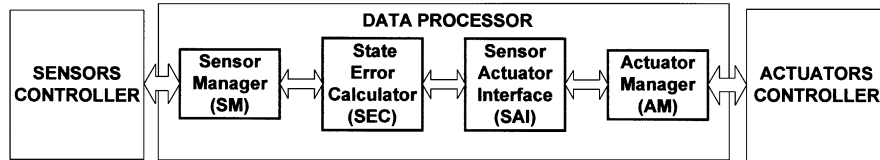

Fig. 3. Decomposition of the robot controller at the control object layer into the Sensors Controller, the Actuators Controller and the Data Processor; the latter is further decomposed into the SM, the SEC, the SAI and the AM.

Table 1
Execution scheme of a system process with the initialization phase, the main loop and the exit phase

```
 1  /* Initialization */
 2  {COMMUNICATION WITH DB AND INITIALIZATION OF LOCAL PARAMETERS}
 3
 4  /*Main loop*/
 5  while {CONDITION}
 6  {
 7     {RECEIVING DATA}
 8     {PROCESSING DATA}
 9     {SENDING DATA}
10  }
11
12  /* EXIT */
13  {STORING TO DB AND MEMORY DEALLOCATION}
```

database (when appropriate) are read (7); then the data become processed (8) and the results are sent to the other system processes (9). The integration platform may implement synchronous or asynchronous communication from either or both sides. The main loop commands may vary according to the procedure state, as will be explained in Sections 4.3 and 5.3. During the exit procedure, which is executed when the task is finished, the memory is released and the data may be stored into the database (13).

4. Sunroof placement application

4.1. Overview

In order to verify the applicability of the methodology we have developed industrial applications. One of them is the ‘‘sunroof placement’’ application, which aims to fit a sunroof onto a car-body on the automobile production line. The benefits of the employment of such a system in the production specialize the general benefits mentioned in Section 1.

We used the 6-DOF manipulator K-125 of KUKA with the KRC-1 controller, which allows integration of external software. Four synchronized cameras were used for recognizing the current image position of the four corners of the sunroof opening, which were compared to their reference position for proper fitting. The edges used for corner detection were enhanced by using LED illumination (Fig. 4). The required accuracy was better than 0.5 mm and the fitting task had to be executed within a few seconds.

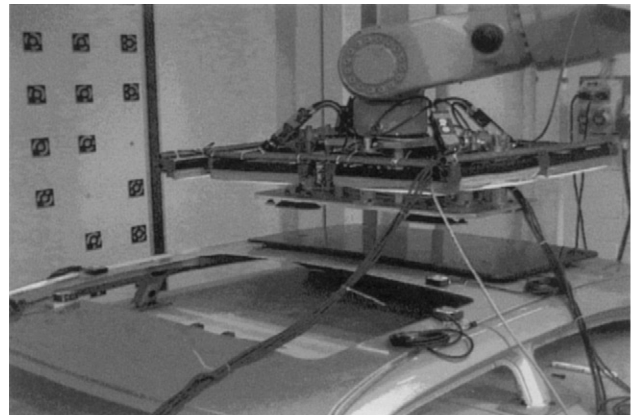

Fig. 4. The experimental installation mainly includes the manipulator, the gripper with four cameras and eight LED lamps mounted on it and the roof of a vehicle with a sunroof hole.

4.2. Software

The software structure at the subsystem level is presented in Fig. 5 and stems from the generic scheme presented in Section 3.2. The image processing requirements were covered using two parallel processes (P5-P6), each of them hosting a pair of instances of the *SM* subsystem and an *SMDB* database instance. Each of the

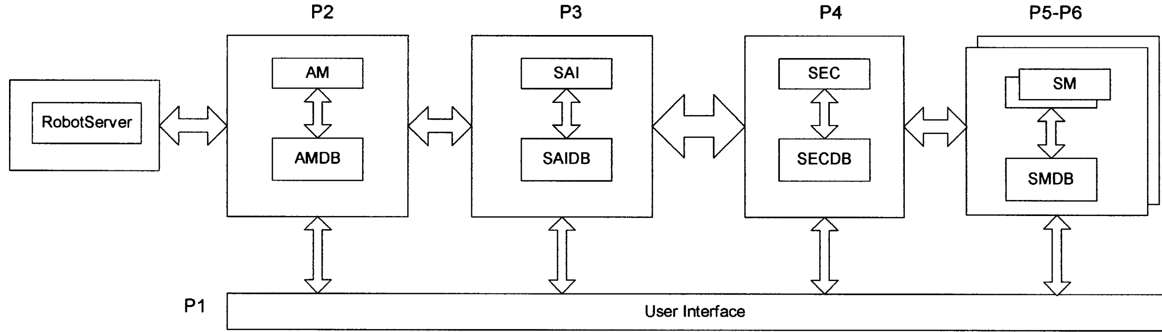

Fig. 5. The software architecture for the sunroof-placing system. One instance of AM, SAI, SEC and four instances of the SM subsystems are used.

SMI-SM4 processed the data acquired by the corresponding camera. The instances of *AM*, *SAI*, *SEC* are placed on the processes P2, P3, P4 correspondingly along with the databases *AMDB*, *SAIDB*, and *SECDB*, which maintain the subsystem parameters. The system communicated with the robot through a server process and with the user through the user interface UI. Appendix B presents some of the most typical library classes that were used for the subsystems' implementation.

The *AM* used a simple PID regulator and the *SAI* was programmed to send the end-effector pose (EEP) error e to *AM* in the immediate robot cycle after the e became available (see Appendix B). In case that no new error estimation was available, the forwarded error was set to zero.

The features objects used in *SM* for visual servoing were the four corners on the sunroof opening (each camera observed a corner). The corners were recognized as composite features (see Appendix B) resulting from the intersection of the sunroof opening edges. The `Track` method implemented a snake-like algorithm for edge recognition [13]. The desired states of the features were defined from the states of the recognized features when the end-effector had the nominal pose.

The approach that we followed in *SEC* for pose error calculation was image-based and endpoint open loop [10]. For the system state estimation we have used the extended Kalman filter [28] and Bierman factorization to reduce error propagation [3] (see Appendix B). The system state was given by the vector

$$W_k = [x, \dot{x}, y, \dot{y}, z, \dot{z}, a, \dot{a}, b, \dot{b}, c, \dot{c}]_k^{\mathrm{T}} \quad (1)$$

which includes the pose error of the end-effector with reference to a nominal pose and the corresponding velocities.[1] For the calculation of the EEP we have established an over-determined system through measuring the image coordinates x_i, y_i of the four corner points. The measurement vector $\mathbf{f}_k$ is given by the following equation:

$$\mathbf{f}_k = [x_1 \quad y_1 \quad x_2 \quad y_2 \quad x_3 \quad y_3 \quad x_4 \quad y_4]_k^{\mathrm{T}}. \quad (2)$$

We have assumed that the image Jacobian that linearizes the non-linear system is constant in the limited workspace region within which the end-effector is expected to move. We have calculated the Jacobian matrix through a training procedure. During this procedure the state and measurement noise covariance matrices were also estimated.

4.3. *Implementation of tasks*

For the realization of the required sets of tasks described in Section 3.1, the presented subsystems as well as the objects defined in the MD-SIR library have been used. The interaction of the subsystems as well as sample code for programming the subsystems are presented.

4.3.1. *Teaching*

The teaching procedure has been implemented as follows: while the user moves the robot along the desired trajectory the system receives synchronously from the robot controller each intermediate pose; then the system stores the pose vectors into the robot pose database.

The modules that participate in teaching (Fig. 6) are the *AM*, the robot server and the *RobPosDB*. The latter is used for storing the intermediate taught robot positions. Analytically:

1. The *AM* initializes the database instance *RobPosDB* sending data that indicate the recording of a new trajectory.
2. From RobotServer the current robot pose is read.
3. The *AM* sets to *RobPosDB* the current EEP.

   Steps 2–3 are repeated at the robot interpolation rate, while the robot moves along the desired trajectory. Then:
4. *AM* signals the end of trajectory recording.

[1] The velocities are calculated using the current, the previous pose and the known interpolation period.

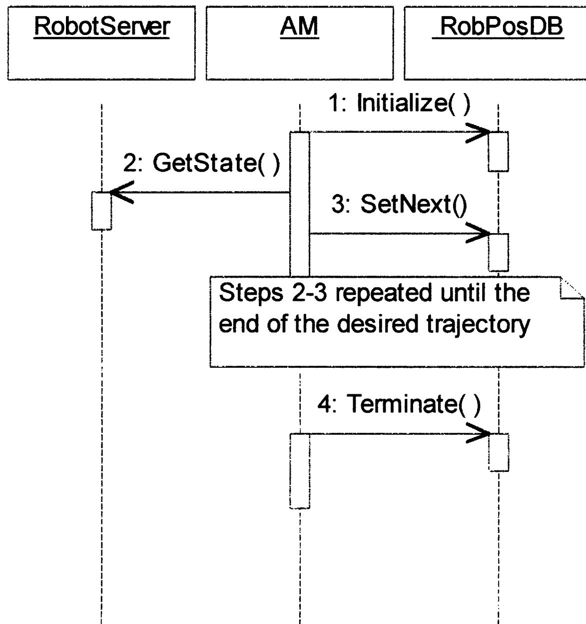

Fig. 6. The subsystems' interactions during teaching. The user moves the robot along the desired trajectory, the system receives synchronously from the robot controller each intermediate pose and then the system stores the pose vectors into the robot pose database.

After recording, the trajectory can be stored permanently for future use. Additionally, tasks requiring sensor feedback may be inserted on intermediate nodes of the trajectory through the user interface.

4.3.2. Training

4.3.2.1. Subsystems' interaction. A sub use case of parameterization with particular interest is the training (experimental calculation of the feature Jacobian). During training the taught robot trajectory is read synchronously from the pose database and is executed, until an intermediate pose is reached, from which a task that requires visual feedback is going to be trained. Then all the subsystems load the parameters that were defined offline for this task (in a previous parameterization procedure). Then the end-effector moves stepwise along a predefined training trajectory for each degree of freedom of the task space. During each training step the sensor subsystem measures the features on the images that are acquired synchronously by the cameras. When the training movement is completed, the feature Jacobian matrix for the task is calculated based on the feature measurements; then the inverse feature Jacobian is stored into the corresponding database.

The participating subsystems are the *AM*, *SAI*, *SEC*, *SM* and the databases *RobPosDB*, *SAIDB*, *SECDB*, *SMDB*. Additionally the *TrainingDB* is used, which is the database that holds the data defining the training movement. The training interaction using a single *SM* instance is presented in detail in Fig. 7 and has as follows:

1. *AM* reads from *RobPosDB* the next stored EEP, at which no sensory feedback is required.
2. *AM* calculates the next EEP eventually using a control law.
3. *AM* sets the next EEP to the robot. The current EEP is returned.
4. Next pose(s) is (are) read from *RobPosDB* and in this case sensory feedback is required.
5. *AM* reads from *AMDB* the required parameters.
6. *SAI* reads from *SAIDB* the required parameters.
7. *SEC* reads from *SECDB* the required parameters.
8. SM reads from *SMDB* the required parameters.
9. *AM* requests (blocked) from *SAI* the offset vector that defines the next training EEP.
10. *SAI* reads the *TrainingDB*, where the consecutive training EEPs are stored. When the offset is read the *SAI* and the *AM* become unblocked.
11. Next EEP is calculated using a control law.
12. *AM* sets the next EEP to the robot. The current EEP is returned.

    Steps 11–12 may be repeated until the distance between the current and the target training pose becomes smaller than a predefined threshold or a timeout occurs. In the second case there is an error condition. We assume the first case:
13. *AM* signals to *SAI* (blocked) that the desired pose is reached and the measurement can be executed.
14. *SAI* requests from *SEC* a measurement (blocked).
15. *SEC* requests measurement from *SM* (blocked).
16. *SM* executes the measurement. After measuring the *SEC* becomes unblocked.
17. *SEC* stores the measurement vector. Then *SAI* and *AM* become unblocked.

    Steps 9–17 are repeated for each training step for all degrees of freedom of the task space. After loop completion:
18. *SEC* calculates the generalized inverse Jacobian.
19. The inverse Jacobian is sent to *SECDB* where it is stored.

After step 19 training is finished. The same procedure is repeated for all tasks that need training. The period of steps 1–3 equals the robot interpolation cycle. The steps 9–11 where sampling and processing take place last for this system about 38 ms (CCIR camera). The rest steps last for fractions of 1 ms.

4.3.2.2. Subsystems' programming. The programming of the main training loop of the *SM* subsystem is presented in Table 2. The image is acquired (2), the feature corner is extracted through the `Track` method of the object `corner` of the class `CCompFeature` (4), the measurement is read (5) and finally sent to *SEC* (7). At

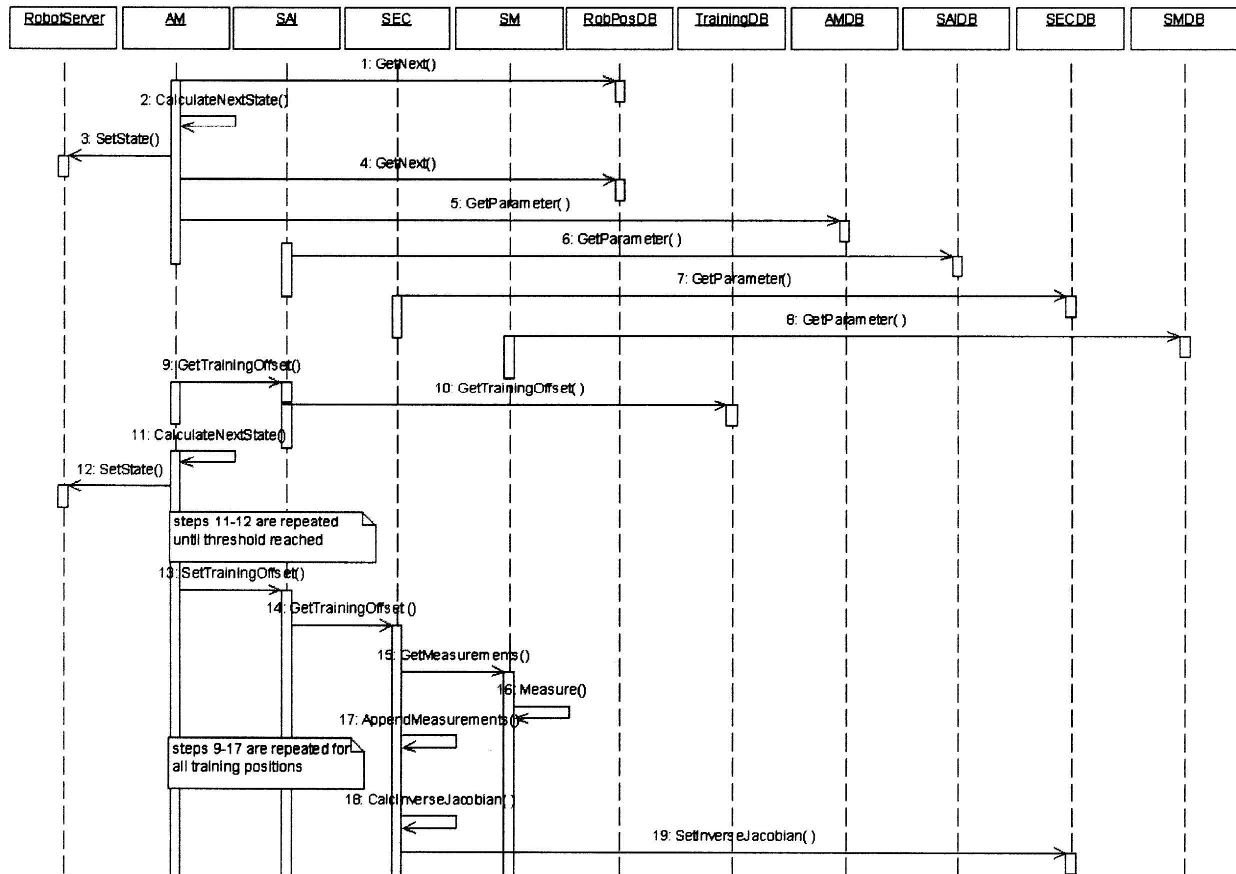

Fig. 7. The subsystems' interaction during training. The taught robot trajectory is executed and when a task requiring visual feedback is to be trained all the subsystems load the parameters from a database, the end-effector moves stepwise along a predefined training trajectory for each degree of freedom of the task space and the sensor subsystem measures the image features. When the training movement is completed the feature Jacobian is calculated and its inverse is stored into the database.

Table 2
Implementation of the training and execution loop of the SM subsystem

```
1 while {CONDITION} {
2   if {camera.Sense ( )! = SNS_NO_ERROR) {EXCEPTION HANDLING)
3   else {
4     if (corner.Track(&pCfg) {EXCEPTION HANDLING}
5     if (corner.GetCurrentState(&measurement) != FTR_NO_ERROR) {EXCEPTION HANDLING}
6 }
7   {SEND MEASUREMENT TO SEC}
8 }
```

the initialization (see Table 1) the `corner` is defined (by combining two `CLine` objects), as well as the sensor through the object `camera` of the class `CMonoCamera` (see Appendix B for description).

The main training loop of *SEC* (Table 3) includes the reception of the next EEP from the *SAI* (2) as well as the previous measurements from *SM* (3). For each intermediate training pose the measurements are set to the object `jc` of the class `CJacobianCalcConstant` (4—see Appendix B for description). The columns of the Jacobian matrix are calculated automatically internally in the `jc`. At initialization the `jc` object becomes initialized and at exit (see Table 1) the calculated Jacobian matrix is inverted and stored into the *SECDB*.

4.3.3. Execution

4.3.3.1. Subsystems' interaction. During execution the taught robot trajectory is read synchronously from the pose database and is executed, until an intermediate pose is reached, from which a task that requires visual feedback is going to be performed. Then all the subsystems load the parameters that were, defined

Table 3
Implementation of the training loop of the SEC subsystem

1	while (CONDITION) {
2	{RECEIVE POSE frame FROM SAI}
3	{RECEIVE MEASUREMENTS meas FROM SM}
4	if (!jc.SetMeasurements (&frame,&meas)) {EXCEPTION HANDLING}
5	}

offline for this task. Then the *AM* requests from the *SAI* the measured error of the EEP. Due to the difference between the robot and the camera sampling rate the measurement may not be available; in that case a zero pose error vector is received, else from the error vector a pose correction vector is calculated (using a PID regulator) and sent to the robot controller. The procedure is repeated until the pose error becomes very small.

The task execution is performed in real time using the parameters defined during parameterization and training (Fig. 8). The participating subsystems are the *AM*, *SAI*, *SEC*, *SM* and the databases *RobPosDB*, *AMDB*, *SAIDB*, *SECDB*, *SMDB*.

The Robot Server, *AM* and *RobPosDB* operate at robot interpolation rate, while the *SEC* and *SM*, operate at sensor sampling rate. The *SAI* communicates asynchronously with *AM* and *SEC*. Therefore, the relative order of the signals between subsystems operating in different rates is just indicative, e.g. 12 is not certain to precede 13.

The steps 1–8 are identical with those presented in Section 4.3.2.

9. *AM* requests an EEP error vector from *SAI*. The *SAI* has no available error, due to the fact that no sensor data have been acquired yet and thus returns a zero vector.
10. *SAI* requests the error from the *SEC* asynchronously.
11. *AM* calculates from the error the desired EEP using a PID regulator.

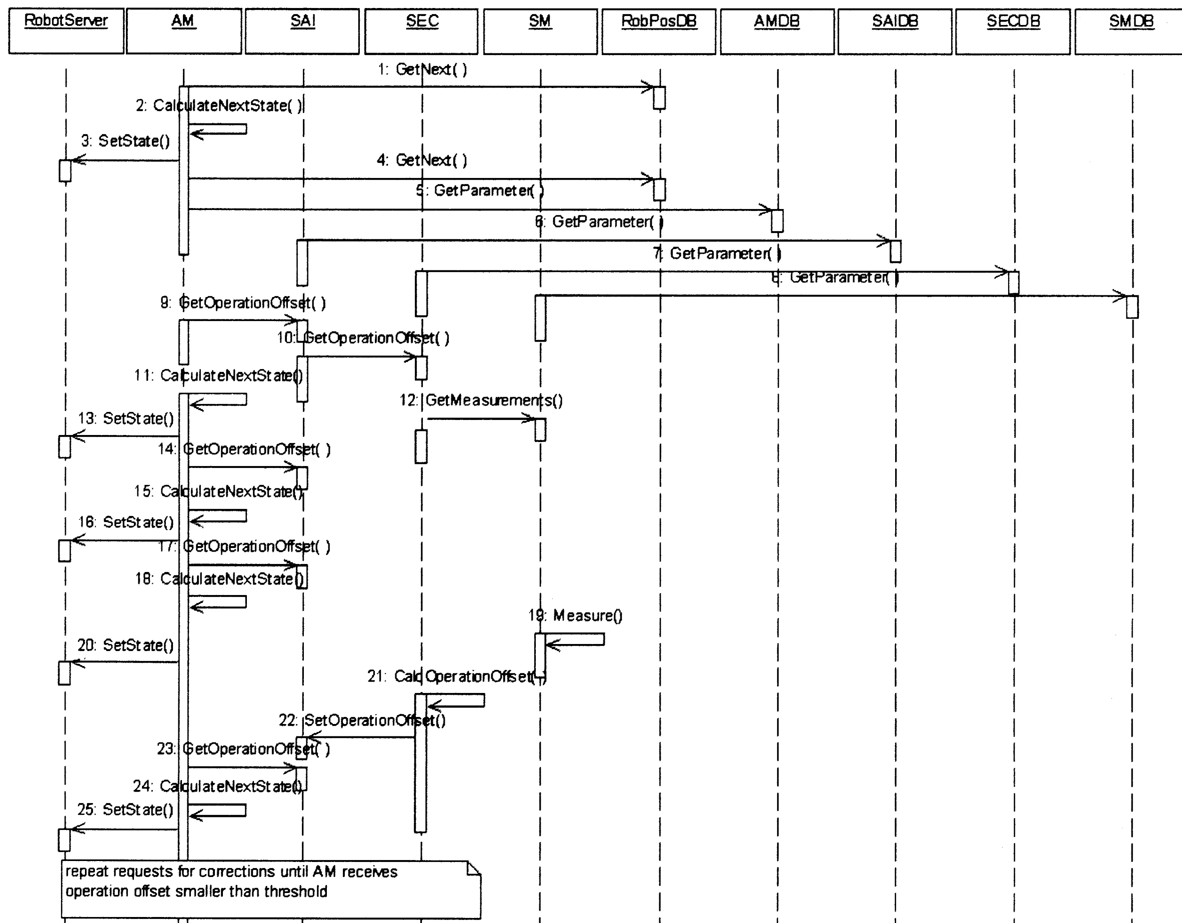

Fig. 8. The subsystems' interactions during execution. The taught robot trajectory is executed and when a task requiring visual feedback is to be performed all the subsystems load the parameters from a database, the AM requests from the SAI the measured error of the EEP, the SAI returns the current pose error or zero (due to the different camera and robot interpolation rate) and the regulated pose is sent to the robot controller. The procedure is repeated until the pose error becomes very small.

12. *SEC* requests a measurement from *SM* synchronously, so that *SEC* will be able to calculate the pose error.
13. *AM* sends to the robot the correction vector.
14. The call is executed similarly to (9) and zero error is returned since there is no measurement available yet.
15. Similarly to (11) the next EEP is calculated using zero error.
16. The next pose is set to the robot.
17. Similar to (14) since no measurement available.
18. Similar to (11).
19. Data acquisition and processing are finished and *SM* returns to *SEC* the feature measurement vector.
20. Similar to (16).
21. *SEC* calculates the EEP error based on the new measurements.
22. *SEC* finished the calculations and calls *SAI* asynchronously and sets the current error vector.
23. *AM* requests from *SAI* the correction for EEP. *SAI* returns the calculated error vector.
24. *AM* calculates the desired EEP from the input error using a PID regulator.
25. *AM* sends to the robot the correction calculated in (24).

4.3.3.2. Subsystems' programming. The code that implements the execution loop for *SEC* and *AM* is presented in Tables 4 and 5, while for the *SM* the code presented in Table 2 applies.

During the main execution loop of *SEC* (Table 4) the measurements from *SM* are read (2); then they are set to the filter (3), the filter cycle is executed (4) and the new state is read (5); finally, the state that expresses the pose error is sent to the *SAI* (6). During initialization (see scheme presented in Table 1) the *SEC* reads from the *SECDB* database the Kalman filter parameters and the corresponding Jacobian matrix and sets them to the filter object (of class `CKalmanFilter`); it also makes a first estimation for the next system state and the state noise covariance.

The execution loop of *AM* (Table 5) is implemented by reading initially the pose error from SAI (2). Then for each degree of freedom it sets the corresponding error to the corresponding regulator (4) and calculates/reads the desired correction (5). Finally, the correction is sent to the robot controller (7). The initialization function (according to the scheme presented in Table 1) reads the *AMDB* database and sets the appropriate parameters to the `regulator [i]` objects.

Many experiments have been performed to verify the proper system function. The results for a typical case are presented in Fig. 9; it illustrates the system response for initial displacement $(x, y, z, a, b, c) = (10\,\text{mm}, 5\,\text{mm}, 10\,\text{mm}, 0.01\,\text{rad}, 0.01\,\text{rad}, 0.01\,\text{rad})$ from the reference EEP for the six degrees of freedom of the task space. Time is expressed in sensor cycles. The rise time was quite fast (circa 10 sensor cycles) and the overshooting was small (less than 25% of the initial value). The error in the final state was practically zero. From the experiments we inferred that given the proper algorithms and a proper parameterization the system was asymptotically stable.

Table 4
Implementation of the execution loop of the SEC subsystem

```
1  while(CONDITION) {
2    {RECEIVE MEASUREMENT VECTOR meas FROM SM}
3    if (!filter.SetF(&meas)) {EXCEPTION HANDLING}
4    if (!filter.PerformCycle()) {EXCEPTION HANDLING}
5    if (!filter.GetW(state)) {EXCEPTION HANDLING}
6    {SEND ESTIMATED ERROR TO SAI}
7  }
```

Table 5
Implementation of the execution loop of the AM subsystem

```
1  while(CONDITION) {
2    {READ EXECUTION OFFSET X[DOF_NUM] FROM SAI}
3    for (dof=0; dof<DOF_NUM; dof++) {
4      if (!regulator [dof] . SetXin(&X[dof])) {EXCEPTION
   HANDLING};
5      if (!regulator[dof] . GetXout(&Y[dof])) {EXCEPTION
   HANDLING};
6  }
7    {SET CALCULATED CORRECTION TO ROBOT}
8  }
```

5. Gap measurement application

The next application built using the MD-SIR methodology was a system for visual gap inspection on the automobile production line. Although it was a non-robotic application it was used for extending the library with reusable components mainly concerning stereo vision.

5.1. Overview

The inspection of produced cars is a significant part of the manufacturing process that includes the dimensional inspection of the openings (gaps) between the surface panels of the vehicle. By employing a visual gap inspection system, the task can be executed very quickly and accurately, without any physical contact or human interference, achieving 100% production monitoring and the measurements can be automatically stored into databases.

We used a measurement head that includes a pair of cameras used for stereo vision and a pair of infra red

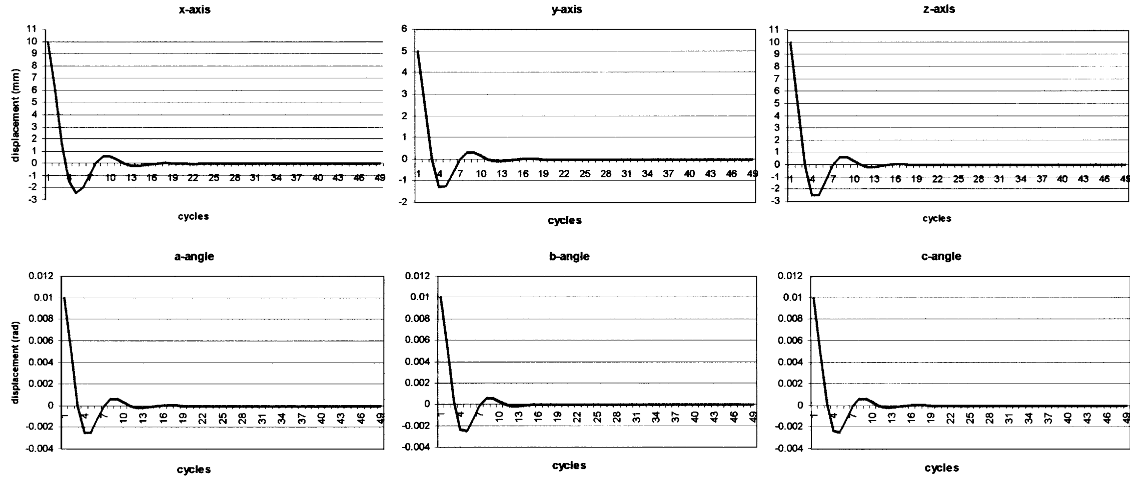

Fig. 9. The system response for initial displacement $(x, y, z, a, b, c) = (10\,\text{mm}, 5\,\text{mm}, 10\,\text{mm}, 0.01\,\text{rad}, 0.01\,\text{rad}, 0.01\,\text{rad})$ from the reference EEP for the six degrees of freedom of the task space. Time is expressed in sensor cycles.

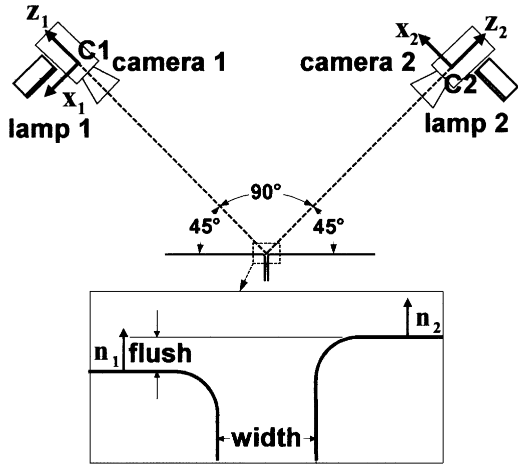

Fig. 10. Definition of gap dimensions (width, flush) and a gap measurement head configuration including two pairs of cameras and lamps.

LED lamps to highlight the edges of the gap, thus achieving independence of the target color and immunity to environmental illumination (Fig. 10). The measurement was triggered by a light barrier. The system had to measure the width and the flush of the gap, with an accuracy better than 0.1 mm. The measured values were compared with the desired ones and then the system indicated corrective actions.

5.2. Software

The software structure at the subsystem level is presented in Fig. 11 and resembles the architecture presented in Fig. 5. The image processing requirements were covered using four parallel processes (P7-P10), each of them hosting a pair of instances of the *SM* subsystem (enhanced compared to the one presented in the first application) and an *SMDB* database instance. Each of the *SM1-SM8* processes the data acquired by one camera. The four instances of the subsystems Gap Calculator (*GC*) (located in processes P3–P6 along with the database *GCDB)*, calculate the gap dimensions using stereo techniques. The subsystem Event Manager (*EM*) and the corresponding database *EMDB* are used for initiating the measurement after receiving triggers from light barriers and will not be further examined.

For the implementation of the *SM* subsystem we have used many of the classes that were implemented for the purposes of sunroof fitting (e.g. `CMonoCamera`, `CImage`, `CFeature`, `CFeatureState`, `CFramegrabber`—see Appendix B). Thus, the development effort and costs were reduced by the corresponding amount required for the development of the reused entities. Additionally, we have implemented through inheritance the classes for the ellipse feature (`CEllipse`) and its state (`CEllipse-State`) for identification of circles that were used during calibration (an object of `CCompFeature`, was defined for recognition of the calibration pattern). For the identification of the gap, which was defined through the `CFeatureGroup⟨T⟩` template.

The subsystem *GC* executes gap calculation using the measurements from a camera pair and provides the means for their calibration. The library has been extended through classes regarding the stereo system (`CStereoSystem`) and the 3D gap (`CGap3D`). The `CStereosyStem` includes the model of the two cameras and enables calculation of 3D points and lines from pairs of two-dimensional (2D) measurements using the camera models (`Calculate3DPoint`, `Calculate3DLine` methods) that implement the calculation procedure described in [9], provided that the calibration has been executed using a proper algorithm (e.g. [25]). The `CGap3D` includes the attributes `Width`, `Flush` and the pose `Pose`. If the cameras are calibrated and the 2D gaps

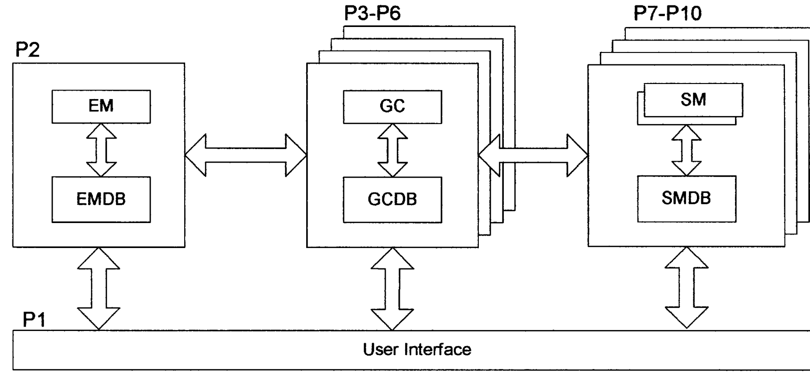

Fig. 11. The software architecture of the static gap-measuring system. One instance of the EM, four instances of the GC and eight instances of the SM subsystems are used.

are known the 3D gap dimensions and pose can be estimated using stereo techniques.

5.3. Implementation of tasks

The tasks include the *parameterization* for image processing and sensor and the *execution*, during which the measurement is performed. These are analyzed in the following.

5.3.1. Parameterization

It regards mainly the parameterization of the vision system (e.g. image feature definition, thresholds). The subsystems used are the *SM1–SM2* for sensor management, *GC* for gap calculation, and the corresponding databases *SMDB*, *GCDB*. The latter stores parameters regarding the 3D recognition and the respective CAD models for the gap.

In the initialization cycle for *SM* the framegrabber, the camera and the gap to be measured are initialized. The parameterization loop (Table 6) for measuring the gap is performed in a similar fashion as the corner measurement in Section 4.3.2 (Table 2). Here, the feature to be recognized (2D gap) is not set directly by the user but through the projection of the 3D gap on the camera plane that is received from the *GC* (line 3).[2] At the end of the loop the measurements are sent to the *GC*.

As regards the *GC*, during initialization the subsystem reads from the database the camera models and the rest 3D processing parameters, the 3D CAD models of the gap and the cameras. The parameterization cycle (Table 7) includes data reading from the database, (2) reception of the 2D measurements regarding the gap from *SM1–SM2* (3) and updating of the related parameters including the camera model (4). Then the 3D gap (`gap3D`) is calculated (5) based on the 2D gaps and the configuration of the stereo system. The measurement is read (6) and written into the database (7). Finally, the projected features (projections of the 3D gap model on the camera planes) are sent to the *SM1–SM2* (8).

5.3.2. Execution

During execution the gap is measured using the values that were defined in the parameterization phase, thus, there is no need for an update of the fixed parameters by reading the databases in each cycle. Furthermore, the projected gap features are not transmitted from *GC* to *SM1–SM2* because they were also fixed during parameterization. For each of the *SM1*, *SM2* the execution loop is implemented by the same code that was presented in Table 6, with the difference that lines 2–3 in Table 6 are omitted. Similarly, for the *GC* subsystem the execution loop is implemented by the same code that was presented in Table 7, with the difference that lines 2 and 8 are omitted.

The validity of the system that has been composed through the methodology is verified by the experimental results, which reveal a typical mean accuracy of 0.07–0.08 mm and an RMS error of 0.09 mm for width and flush correspondingly [14]; thus the user requirements are fulfilled.

5.4. Potential reuse: the gap-measuring robot

From the MD-SIR library new applications can be derived. One of them is the gap-measuring robot. The setup and maintenance overhead of running a static system with separate measurement heads for each measurement position is quite big and the system is unable to measure gaps on the front and rear car panels (except for the ones that are visible from the top) due to the assembly line movement that allows only measuring from the side. Finally, the vehicle displacement from the reference position on the assembly line is not allowed to be very big due to calibration inaccuracies and illumination considerations. These constrains can be

[2] The camera calibration procedure, which is not described here defines the relative pose between camera and 3D gap.

Table 6
Implementation of the parameterization loop of the SM subsystem

```
1while (CONDITION) {
2  {UPDATE PARAMETERS BY READING DATABASE}
3  {RECEIVE PROJECTED GAP FROM GC}
4  if (camera.Sense( )! = SNS_NO_ERROR) {EXCEPTION HANDLING}
5  else {
6     if (gap2D.Ttack (&ipconfig) ) ! = FTR_NO_ERROR) {EXCEPTION HANDLING}
7     if (gap2D.GetCurrentState(&measurement))! = FTR_NO_ERROR) {EXCEPTION HANDLING}
8}
9  {SEND MEASUREMENT TO GC}
10}
```

Table 7
Implementation of the parameterization loop of the GC subsystem

```
1 while (CONDITION) {
2    {READ APPROPRIATE DATA FROM DATABASE}
3    {RECEIVE GAP MEASUREMENTS FROM SM1,SM2}
4    {UPDATE GC PARAMETERS}
5    if (!gap3D.Calculate(&gap1_2D,&gap2_2D,&stereo))
   {EXCEPTION HANDLING}
6    if (!gap3D.GetMeasurements(&measurements)) {EXCEPTION
  HANDLING}
7    {WRITE GAP MEASUREMENT TO DATABASE}
8    {SEND DESIRED PROJECTED FEATURES TO SM1, SM2}
9  }
```

overcome if we use a robot with a gap-measuring head (robotic tool) mounted on its hand.

Similar to the sunroof fitting system, the tasks of *teaching*, *parameterization* and *execution* are required. During execution the robot follows the recorded trajectory blindly and when the measurement position is reached it measures the gap and its relative gap pose. If it is not within the acceptable limits a correction pose is generated and the procedure is repeated in the new pose until the pose becomes valid for measurement.[3] Then the measurement is forwarded for storage.

It is clear that the functionality required for building a gap-measuring robot has been already implemented and thus the system can be composed by assembling properly existing subsystems (Fig. 12). The pattern recognition requirements are identical with the ones presented in the static gap-measuring system and the SM subsystem defined in Section 5.2 can be reused. The error of the tool pose can be calculated using the relative pose of the tool with regard to the gap using the reference pose defined in calibration. Therefore, the whole *GC* subsystem defined in Section 5.2 can be reused to implement the State Error Calculator (*SEC*). The functionality that was developed for the *AM* subsystem of Section 4.2 covers the regulator requirements. The same applies to the *SAI* subsystem. Thus, the most difficult and time-consuming tasks of developing the subsystems can be by-passed by reusing the existing and well-tested components of the MD-SIR library on the basis of the proposed architecture.

6. Conclusions and future work

In this paper, we have presented a new methodology for developing software for robotic applications with sensory feedback at the end-effector level, provided that an open controller architecture is available. The presented methodology leads to the creation of a library of reusable entities. The entities can be applied to compose robotic applications that employ sensory feedback and the library evolves as new modular applications are built.

The library entities (subsystems and objects) belong to the control object layer of the open architecture thus ensuring maximum portability. Their functionality includes sensor interfacing, sensor modeling, pattern recognition, pose estimation and state regulation. The main advantage of the proposed approach is that it may potentially decrease dramatically the cost of development of new applications, with minimal intervention both in the programming and in the mechanical aspects. By following the interaction paradigms and by using objects from the library classes the processing loops of the subsystems can be easily implemented. The system programming following the MD-SIR methodology can lead to high quality systems that fulfill the user requirements, provided that the system has been properly parameterized and that the proper algorithms are employed. The proposed architecture promotes modularity and reusability; subsystems or objects can be easily substituted by other similar ones with no other system alteration, provided that the same interfacing rules are maintained. The library is developed dynamically in extent with implementation of new functionality, but also in depth with the easy incorporation of new, more robust and efficient algorithms. The new

[3] If the relative pose of the measurement head and the gap is far from the reference one then the measurement error is unacceptable due to the calibration error and due to the displacement of the highlighted lines (specular reflection).

algorithms can be easily integrated in industrial systems and a closer collaboration between industry and research can be established.

We have demonstrated the proposed methodology by composing two real applications, a sunroof fitting robot and a gap-measuring installation. In the latter the biggest part of sensor managing objects were reused—thus significantly reducing the developing costs—and the library was enriched with algorithms for recognition of new visual features, camera calibration and stereo vision. The benefits of reuse may become much more obvious in the future, in the implementation of a gap-measuring robot. The presented applications served as paradigms and demonstrated the interaction rules (interfacing) between the modules. The methodology was demonstrated for integration of multiple cameras, but it can be extended to incorporate other popular sensors, since the architecture poses no strict constraint regarding the data communicated between the *SM*, *SEC* and *SAI*; these data are simply the vectors of the sensor measurements and the state error.

Anyone who deals with the development of robotic applications can benefit from this work, especially those that seek to couple the mechanical flexibility of industrial robots, with the flexibility to "build" various diverse applications with common characteristics.

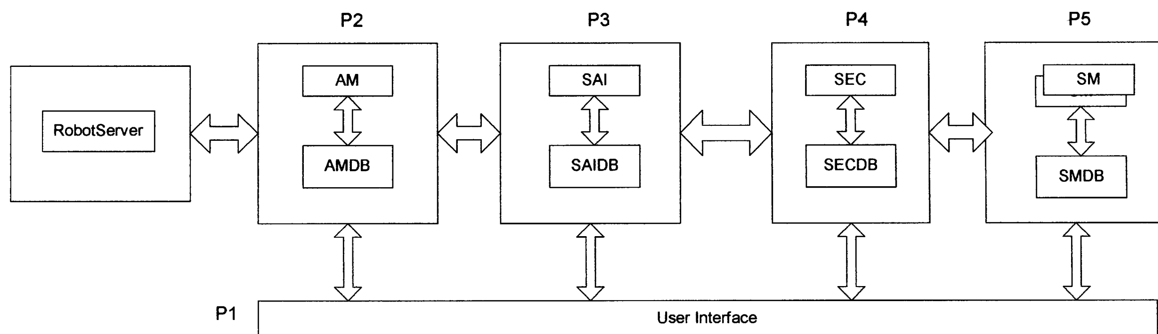

Fig. 12. The software architecture of a robotic gap-measuring system. One instance of AM, SAI, SEC and two instances of the SM subsystems are

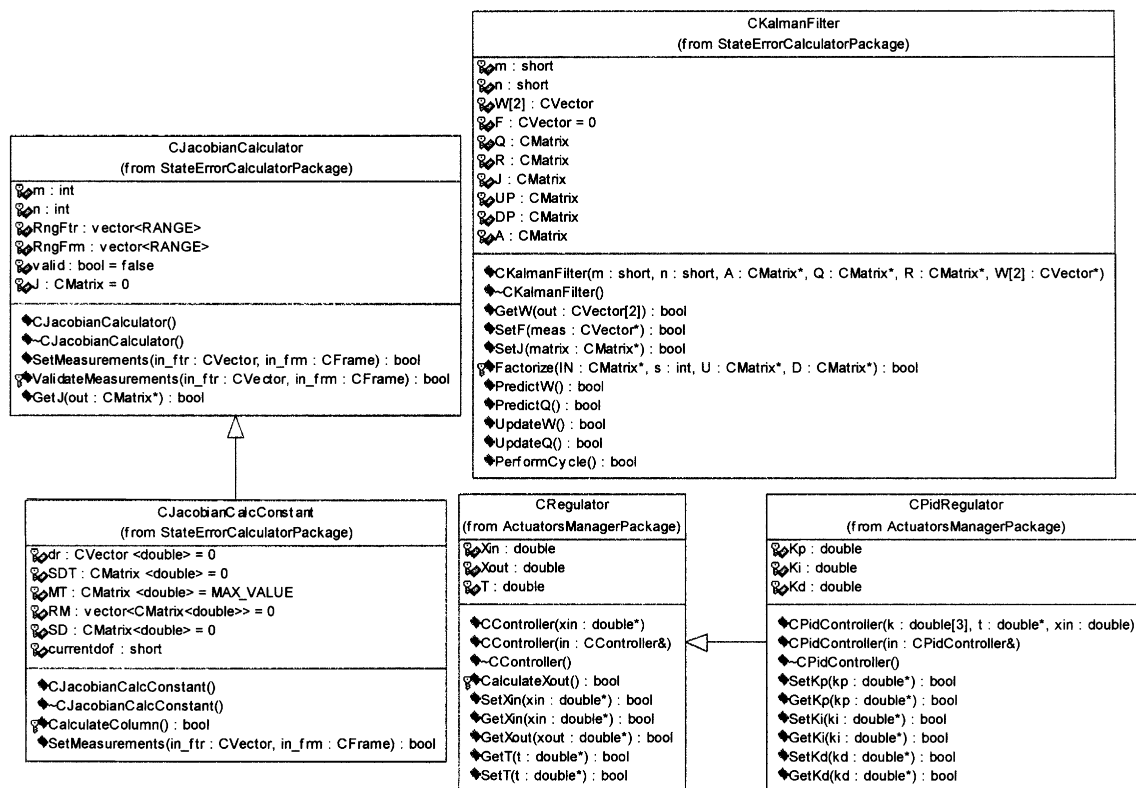

Fig. 13. The classes implementing the Jacobian Calculator (CJacobianCalculator, CJacobianCalcConstant) the Kalman filter (CKalmanFilter) and the Regulator (CRegulator, CPidRegulator); the most important attributes and methods are presented.

Previous efforts such as OSACA and OROCOS have been considered and it is the aim of the authors that this work will trigger further investigation and will inaugurate interactions with these research groups.

In the future the need for more intelligent industrial robots that will operate using sensory feedback will become more obvious. Therefore, methodologies like MD-SIR are expected to gain in interest. Since it is well known that no single sensor type is able to guarantee full perception of the environment [5], the library has to be extended with classes that will provide the infra structure for sensor fusion. A long-term objective is the implementation of an open integrated environment for the development and testing of the control objects. This environment will support the use of many sensors and actuators and will include their models for simulation purposes along with pattern recognition and control algorithms.

Appendix A

The abbreviations used in the paper are presented in Table 8.

Appendix B

Some of the most typical classes and class templates included in the MD-SIR library are briefly described in the following.

Table 8

AM	Actuator manager
AMDB	Actuator manager database
EEP	End effector pose
EM	Event manager
EMDB	Event manager database
GC	Gap calculator
GCDB	Gap calculator database
RobPosDB	Robot pose database
SAI	Sensor actuator interface
SAIDB	Sensor actuator interface database
SEC	State error calculator
SECDB	State error calculator database
SM	Sensor manager
SMDB	Sensor manager database
TrainingDB	Training database
UI	User interface

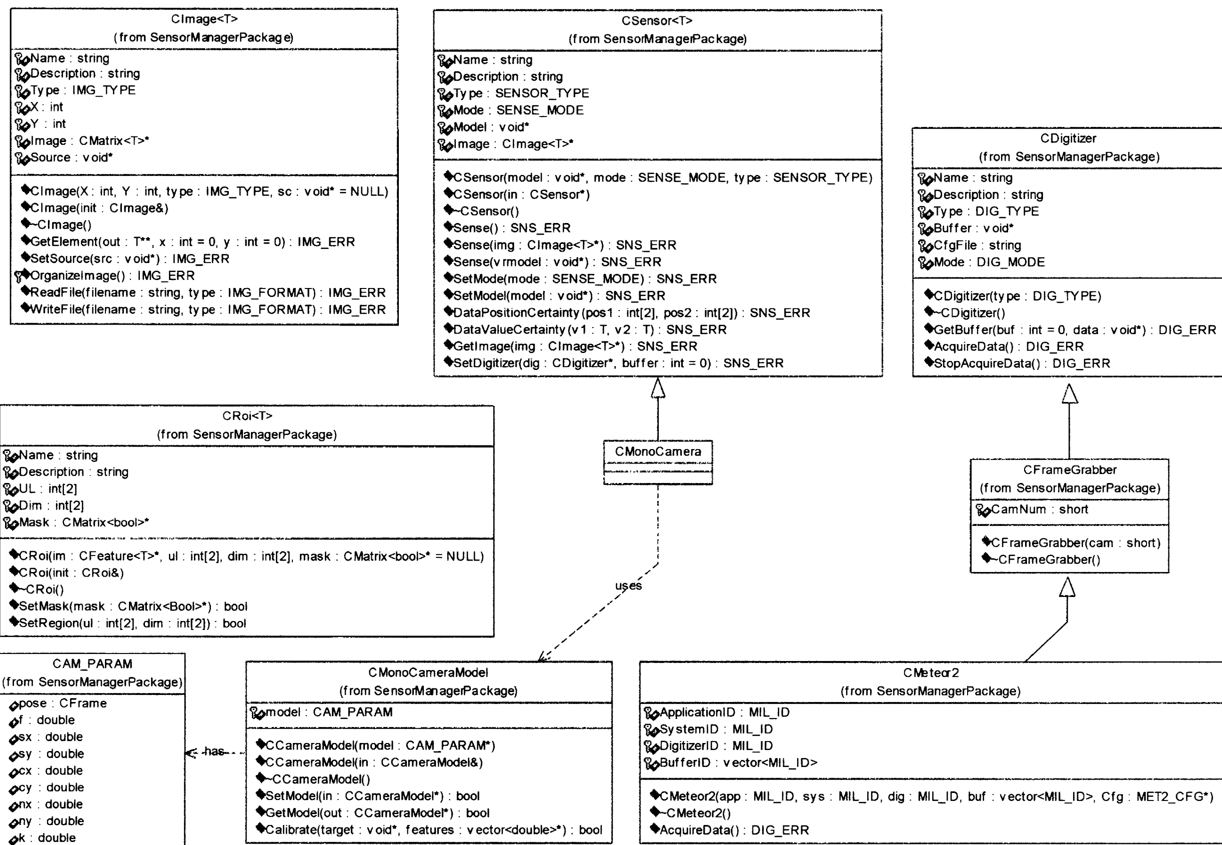

Fig. 14. The classes, class templates and structures implementing the sensors (CSensorc⟨T⟩, CMonocamera, CMonoCameraModel, CAM_PARAM), the image (CImage⟨T⟩), the ROI (CRoi⟨T⟩) and the digitizers (CDigitizer, CFramegrabber, CMeteor2); the most important attributes and methods are presented.

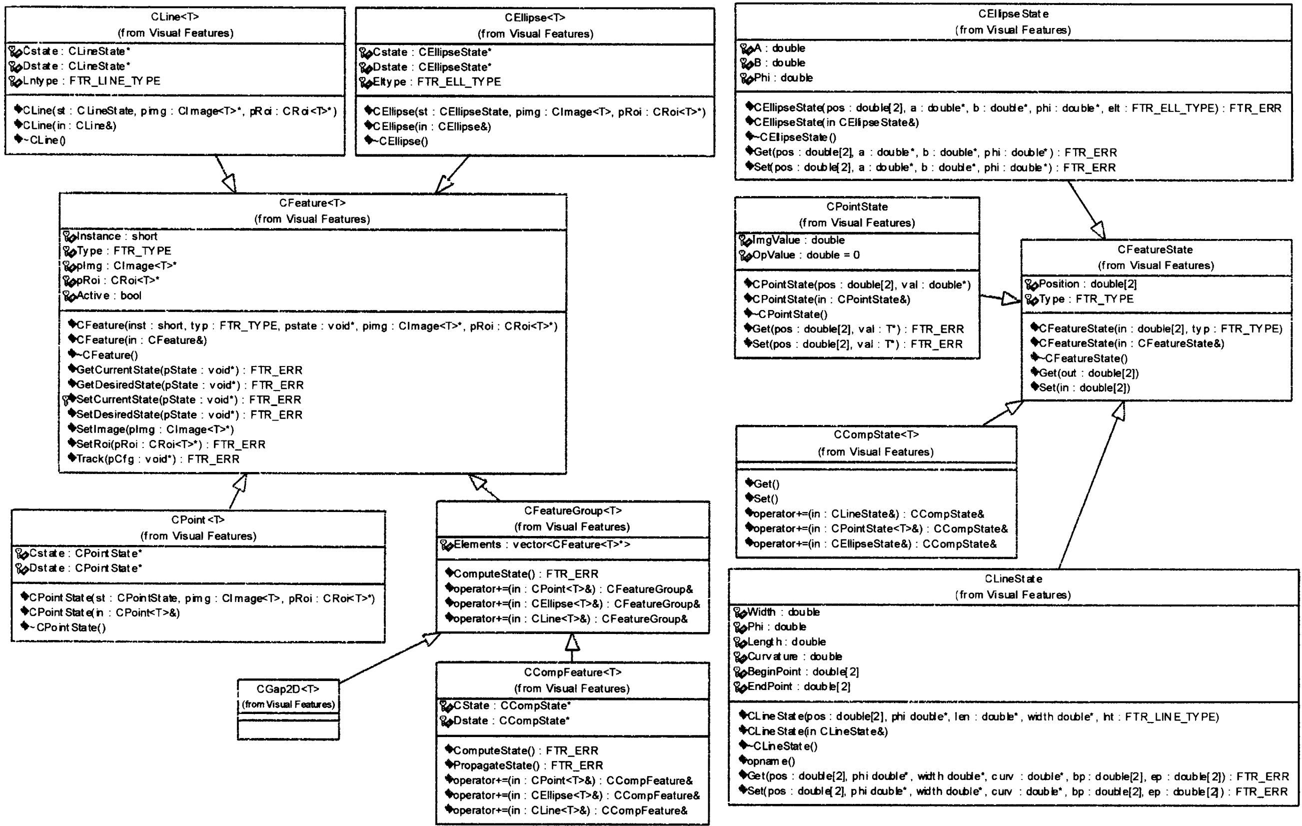

Fig. 15. The classes and class templates implementing the features (CFeature⟨T⟩, CLine⟨T⟩, CEllipse⟨p⟩, CPoint⟨T⟩, CFeatureGroup⟨T⟩, CCompFeature⟨T⟩) and their states (CFeatureState CLineState, CEllipseState, CPointState, CCompState⟨T⟩); the most important attributes and methods are presented.

B1. Actuator manager and sensor–actuator interface

The regulator abstract class `CRegulator` is presented in Fig. 13 and defines the input and the output and methods for reading–writing and calculating the output. The `CPidRegulator` class is derived from `CRegulator` and implements a PID regulator. It additionally includes, the proportional–integral–derivative factors (Kp, Ki, Kd), and the interpolation cycle attribute `T`.

B2. State error calculator

The Jacobian matrix calculation is performed using the `CJacobianCalculator` abstract class, from which the `CJacobianCalcconstant` is derived[4] (Fig. 13). The most important attributes of the base class are the calculated matrix `J` and the value range of the input features and the input pose. In the `CJacobianCalcConstant` class we have defined the matrices for the desired minimum values of the Jacobian elements `SDT` and the threshold for the allowed standard deviation `MT` and the array vector `RM` that holds all the training measurements.

The class definition that has implemented the Kalman filter is presented in Fig. 13 `CKalmanFilter`. It includes the measured and the predicted state `W`, the measurement vector `F`, the state and measurement noise covariance matrices, `Q`, `R`, the Jacobian `J` that linearizes the non-linear system and the system matrix `A`. The filter cycle is performed by calling the `PerformCycle` method (or the user can call separately the `SetF`, the `Update` and the `Predict` methods).

B3. Sensor manager

The class template `CSensor⟨T⟩` uses a pointer to the sensor model `Model` (CMonoCameraModel for monochrome cameras) to acquire data through the `Sense` method and the data are stored into the location pointed by `Image` (Fig. 14). The sensor images are implemented in a unified fashion by the class template `CImage⟨T⟩`. The `T` can be `unsigned char` for monochrome camera images, `double` for range images, etc. The `Source` pointer is the buffer of the digitizer. The region of

[4] From the reference pose the tool performs predefined step movements for each degree of freedom and the measured features are combined with the intermediate poses to calculate the Jacobian elements using linear regression.

interest (ROI) is a subset of the image, in which we search for particular features and it is implemented by the `CRoi<T>` template. By applying a mask to it (`Mask`) we are able to limit the search within any custom-defined region in the ROI. The digitizers are implemented by the abstract class `CDigitizer`, from which type specific (`CFrameGrabber`) and vendor specific classes (`CMeteor2`) classes can be derived through inheritance.

The class templates of some 2D features that we recognize in the sensory data are presented in Fig. 15. From the abstract class template `CFeature<T>` various feature class templates can be defined. It includes pointers to the image (`PImg<T>`) and to the related ROI (`PRoi<T>`). Furthermore, it defines the virtual function `Track`, which recognizes the feature in the image within the ROI, given the processing parameters (pointed by `pCfg`). When integrating new pattern recognition algorithms it is the only method that has to be reprogrammed. The features may be geometrical, e.g. point (`CPoint<T>`), line `CLine<T>`) or algorithmic; each feature has its own desired and measured state (`CPointState`, `CLineState`). Groups of features may be specified through the `CFeatureGroup<T>` template and an example is the `CGap2D<T>` that uses line features to detect gaps in 2D. The concept of feature group is extended to define a composite feature (`CCompFeaturec<T>` class template). The composite feature state depends on the individual feature states but also gives feedback to them [8].

References

[1] Anderson RJ. SMART: a modular architecture for robots and teleoperation. IEEE International Conference on Robotics and Automation, Atlanta, Georgia, 1993.

[2] Berry G. A quick guide to ESTEREL. Technical Report, Ecole des Mines de Paris and INRIA, 1997.

[3] Bierman GJ. Factorization methods for discrete sequential estimation. New York: Academic Press, 1977.

[4] Borelly J, Maniere E, Espiau B, Kapellos K, Pissard-Gibollet R, Simon S, Turro N. The ORCCAD architecture. Int J Robotics Res 1998;17(4):338–59.

[5] Chandrinos KV, Kosmopoulos DI, Spyropoulos CD. Vision-based navigation for indoor service robots. European Workshop on Service and Humanoid Robots, Santorini, Greece, 2001.

[6] Common Object Request Broker Architecture, http//:www.corba.org.

[7] Coste-Manière E, Turro N. The MAESTRO language and its environment: specification, validation and control of robotic missions. IEEE Int Conf Intelligent Robots Systems 1997;2:836–41.

[8] Hager G, Toyama K. XVision: a portable substrate for real-time vision applications. Comput Vision Image Understanding 1998;65(1):14–26.

[9] Hager GD. A modular system for robust positioning using feedback from stereo vision. IEEE Trans Robotics Automation 1997;13(4):582–95.

[10] Hutchinson S, Hager G, Corke P. A tutorial introduction to visual servo control. IEEE Trans Robotics Automation 1996;12(5):651–70.

[11] Intel Corporation: Open source computer vision library, http://www.intel.com/research/mrl/research/opencv/.

[12] Kapoor C. A reusable software architecture for advanced robotics. Ph.D. dissertation, University of Texas, Austin, 1996.

[13] Kass M, Witkin A, Terzopoulos D. Snakes: active contour models. Int J Comput Vision, 1988;1(4):321–31.

[14] Kosmopoulos D, Varvarigou T. Automated inspection of gaps on the automobile production line through stereo vision and specular reflection. Comput Ind 2001;46:49–63.

[15] Lange F, Hirzinger G. Is vision the appropriate sensor for cost oriented automation? Sixth IFAC Symposium on Cost Oriented Automation, Berlin, Germany, 2001.

[16] Mallet A, Fleury S, Bruyninckx H. A specification of generic robotics software components: future evolutions of GenoM in the OROCOS context. IEEE International Conference on Intelligent Robots and Systems, Hawaii, HA, 2001.

[17] Marchand E. ViSP: a software environment for eye-in-hand visual servoing. IEEE International Conference on Robotics and Automation, Detroit, MI, vol. 4, 1999. p. 3224–30.

[18] Mazer E, Boismain G, Bonnet des Tuves JM, Douillard Y, Geoffroy S, Doubourdieu N, Tounsi M, Verdot F. START: an application builder for industrial robotics. IEEE International Conference on Robotics and Automation, Leuven, Belgium, 1998.

[19] Morales ER. GENERIS: the EC-JRC generalized software control system for industrial robots. Ind Robot 1999;26(1):26–32.

[20] MVTec Software GmbH, http://www.mvtec.com/.

[21] Real Time Innovations: ControlShell, http://www.rti.com/products/controlshell/CS.html.

[22] Sperling W, Lutz P. In: Enabling open control systems—an introduction to the OSACA system platform, robotics and manufacturing, vol. 6. New York: ASME Press, 1996.

[23] The OROCOS project http://www.orocos.org.

[24] Toyama K, Hager G, Wang J. Servomatic: a modular system for robust positioning using stereo visual servoing. International Conference on Robotics and Automation, Minneapolis, 1996. p. 2636–43.

[25] Tsai RY. A versatile camera calibration technique for high-accuracy 3D machine vision metrology using off-the-shelf TV cameras and lenses. IEEE Trans Robotics Automation 1987;3(4):323–44.

[26] Valavanis KP, Saridis GN. Intelligent robotics systems: theory, design and applications. Kluwer: Boston, 1992.

[27] Vxl/TargetJr standard software platform, http://www.esat.kuleuven.ac.be/~targetjr/.

[28] Wilson WJ, Williams Hulls CC, Bell GS. Relative end effector control using Cartesian position based visual servoing. IEEE Trans Robotics Automation 1996;12(5):684–96.