Rubik’s cube as a benchmark validating MRROC++ as an implementation tool for service robot control systems

C. Zielinski, W. Szynkiewicz, T. Winiarski and M. Staniak
Institute of Control and Computation Engineering, Warsaw University of Technology, Warsaw, Poland

W. Czajewski
Institute of Control and Industrial Electronics, Warsaw University of Technology, Warsaw, Poland, and

T. Kornuta
Institute of Control and Computation Engineering, Warsaw University of Technology, Warsaw, Poland

Abstract
Purpose – This paper seeks to develop universal software (a programming framework) enabling the implementation of service robot controllers. The software should distinguish the hardware-oriented part of the system from the task-oriented one. Moreover, force, vision as well as other sensors should be taken into account. Multi-effector systems have to be considered.

Design/methodology/approach – The robot programming framework MRROC++ has been implemented as a hierarchical structure composed of processes, potentially consisting of threads. All of the software is written in an object-oriented manner using C++ and is supervised by a QNX real-time operating system. The framework has been verified on several systems executing diverse tasks. Here, a Rubik’s cube puzzle-solving system, consisting of two arms and utilizing force control and visual servos, is presented.

Findings – The presented framework is well suited to tasks requiring two-handed manipulation with force sensing, visual servoing and online construction of plans of actions. The Rubik’s cube puzzle is a reasonable initial benchmark for validation of fundamental service robot capabilities. It requires force sensing and sight coupled with two-handed manipulation and logical reasoning, as do the majority of service tasks. Owing to the use of force sensing during manipulation, jamming of the faces has always been avoided; however, visual servoing could only cope with slow handing over of the cube due to the volume of computations associated with vision processing.

Research limitations/implications – The proposed software structure does not limit the implementation of service robot controllers. However, some of the specific algorithms used for the solution of the benchmark task (i.e. Rubik’s cube puzzle) need to be less time-consuming.

Practical implications – The MRROC++ robot programming framework can be applied to the implementation of diverse robot controllers executing complex service tasks.

Originality/value – A demanding benchmark task for service robots has been formulated. This task, as well as many others, has been used to validate the MRROC++ robot programming framework which significantly facilitates the implementation of diverse robot systems.

Keywords Benchmarking, Robotics, Software tools

Paper type Research paper

Introduction
Service robots are meant to operate in human-oriented environments, because their primary purpose is helping people in their natural surroundings. Human-oriented environments impose certain requirements on the robot. Those requirements add up to the necessity of building a device with similar capabilities, in some respects, to those of human beings, e.g.: acquiring objects by using eyesight, coordinating the motions of the hand with what the eyes see, two handed manipulation, force sensing enabling avoiding jamming of mated objects, logical reasoning about the actions to be performed. To build and test a robot with such capabilities one has to define a benchmark task that will require all of the above actions to successfully accomplish the goal. One such task can be the solution of a Rubik’s cube puzzle. At first sight it might seem as an easy task,
but in reality it is not so. The robot has to acquire a very metamorphic object (a Rubik's cube can change its outward appearance very considerably, there are no assumptions as to the background or lighting conditions). It requires two hands to rotate its faces, and moreover, the faces can jam if not aligned properly (this requires position-force control). The cube is handed to the robot by an operator, thus neither its location is known \textit{a priori}, nor the cube is static when being acquired (this requires visual servoing). Once grasped, the state of the cube must be identified, and out of that information a plan must be deduced, i.e. the sequence of rotations of the faces leading to the state in which each of the faces contains tiles of the same colour (this requires artificial intelligence search algorithms for solving the puzzle). It is important that the plan of actions cannot be produced off-line – it must be defined while the task is being executed. Each of the above steps cannot take too long, thus time-efficient algorithms must be utilized. Above all, all of those capabilities must be integrated into a single system. This requires an adequate software tool. To produce, in a reasonable time, a controller that can drive a two-handed robot equipped with: cameras, force-torque sensors, etc. and capable of logical reasoning, one has to rely on some ready software platform. Best suited for those purposes are robot programming frameworks, i.e. libraries of software modules for programming robots, together with a set of patterns being a guide to assembling those modules into a ready control system. As the library might not have all the necessary components, therefore there are tools included for producing extra ones. The new components are coded in the base language of the framework (e.g. C or C++).

The approach to controller implementation by utilising a robot programming framework has been followed by many researchers. Some of the recently developed frameworks are: GeNoM (Fleury and Herrb, 2001), DCA (Petersson et al., 2001), OROCOS (Bruyninckx et al., 2003) or the player/stage suite of software (Gerkey et al., 2003, 2004). The Authors, while working on a number of controllers for prototype robots executing very diverse tasks (Zielinski et al., 2003, 2005), have come up with MRROC++ – a Multi-Robot Research-Oriented Control programming framework (Zielinski, 1999). Its base language is C++ as thus addition of new components is rather straightforward. The resulting controller is a multi-process system supervised by a real-time distributed operating system (QNX Neutrino).

The paper describes the structure of MRROC++ in general, and the components of the controller solving the Rubik's cube puzzle (Szynkiewicz et al., 2006; Zielinski et al., 2006), as well as the way in which they have been integrated into a single system. Finally, the results of experiments conducted on the finished system are summarised and the generality of the solution is discussed in the concluding section.

\textbf{The structure of MRROC++ based systems}

MRROC++ is a tool for implementing either single- or multi-robot hierarchical control systems. It provides a library of software modules (i.e. classes, objects, processes, threads and procedures) and design patterns according to which any multi-robot system controller can be constructed. This set of ready made modules can be extended by the user by coding extra modules in C++. The freedom of coding is, however, restricted by the general structure of the system. New modules have to conform to this general structure. Even if a single-robot controller is designed it is assumed that it can work in a multi-robot environment, so its controller really has the capability of controlling several robots. The same applies to sensors. Regardless of the fact, whether they are necessary for the execution of the user's task, the potential for their utilization always exists in the resulting system. The structure of MRROC++ is due to formal considerations presented in Zielinski (2001, 2006). Those considerations have been instrumental not only in the specification of the framework, but also in its implementation (Zielinski et al., 1998).

A MRROC++ based control system is implemented as a set of processes (Figure 1):

- **UI** – \textit{user interface process}. A single system configuration dependent process, responsible for the communication with the human operator.
- **MP** – \textit{master process}. A single process responsible for the coordination of all of the effectors present in the system.
- **ECP** – \textit{effector control process}. Responsible for the execution of the task allotted to a single effector – several of those processes can coexist (the number depends on the number of distinct effectors present in the system).
- **EDP** – \textit{effector driver process}. Responsible for controlling the hardware associated with the effector – there are as many EDPs as there are ECPs.
- **VSP** – \textit{virtual sensor process}. Responsible for performing data aggregation on real sensor readings and thus producing a virtual sensor reading – zero or more of those processes can be associated with MP and any of the ECPs.

Some of those processes have a multi-thread structure. The processes fit a hierarchical structure and cooperate with each other using client-server model of data exchange. They run concurrently in the nodes (PC type computers) of the QNX network. Owing to the distributed architecture and parallel execution, the computational power of the designed system can be adjusted to the needs of the task at hand. Synchronous message passing is the main method of communication during rendezvous.

In the usual case the user's task is coordinated by the MP. It is assumed that, in general, there can be any number of effectors in the system (e.g.: manipulators). From the point of view of the executed task the MP is the coordinator of all effectors present in the system. It is responsible for trajectory generation in multi-effector systems, where the effectors cooperate tightly. In the case of loose cooperation it just synchronizes the effectors from time to time. In the case of fully independent actions of the effectors the MP remains dormant.

The general specification of the task for all of the effectors is concentrated in the variable elements of MP. Each effector has two processes controlling it: ECP and EDP. The former is responsible for the execution of the user's task dedicated to this effector and the latter for direct control of this effector. The EDP is responsible for direct and inverse kinematics computations as well as for both position and force servo-control. The EDP has been divided into several threads: EDPM (communication with the ECP including interpretation of its commands), EDPT (selection of force and position control directions), EDPS (motor position servo-control) and EDPF (force measurements). The UI depends only on the number of effectors constituting the system. It is responsible for presenting the system status to the operator.
and enables the operator to start, pause, resume or terminate the execution of the task.

Both the EDPs and the VSPs depend on the associated hardware, whereas MP and ECPs are hardware independent, but depend on the task that is to be executed by the system (Figure 1). This division highly simplifies coding. When the hardware configuration is fixed the programmer with each new task modifies only the MP and the ECPs. Only when this configuration has to be changed, e.g. by introducing a new type of manipulator, a new EDP has to be appended to the system. New sensors or new methods of data aggregation require new VSPs. Processes driving devices are implemented as resource managers, which deliver certain services to the process-clients.

The MP, ECPs and VSPs are structured into: the shell (which is composed of the code responsible for the communication with the other processes and performing error handling) and the kernel (which is responsible for the execution of a specific task and thus is delivered by the user). In the case of MP and the ECPs the kernel is usually composed of the move instructions. The move instructions invoke motion generators. The move instructions of the MP pertain to all of the effectors while the move instructions of the ECPs deal with single effectors. In both cases those instructions cause the computation of a command that is being transmitted to the lower control layers, e.g. the MP sends appropriate data to the ECPs, and ECPs to the EDPs, and finally the EDPs cause the execution of the command. The main argument of the move instruction is a motion generator. A motion generator computes the next effector position or the force to be exerted by it taking into account the current state of the effector, the associated VSP readings and the data obtained from the higher layers of the control system. It generates the set value for the next motion increment (a low multiple of the servo sampling period, i.e. 2 ms). The programmer wanting a specific motion or behaviour of the effectors produces, according to a pattern, a new motion generator and causes a move instruction to use it as its argument. The motion generator is a transition function governing the behaviour of the whole system (in the MP) or a single effector (in an ECP). It uses the inputs (i.e. the current effector pose expressed in one of the position/orientation spaces, the force/torque readings and the virtual sensor readings), as well as the data stored in the variables of the kernel of the process (either in the MP or the ECP, respectively) to compute the values that will be transferred to the outputs (i.e. the desired effector pose and virtual sensor configuration command). As a side effect the data contained in the process kernel might change as well. It is the responsibility of the motion generator to produce the outputs on the basis of the inputs and the contents of the internal memory. The responsibility of the move instruction is to supply the current input values and to dispatch the results of computations (i.e. the outputs to the other system components). To perform those computations it invokes the motion generator. Error diagnostics and recovery depend on: type of error detected, place in the code that it had occurred, structure of the system. To facilitate error handling three classes of errors have been distinguished in MRROC++: non-fatal, fatal and system errors. Non-fatal errors are caused by computational problems or wrong arguments of commands/replies within the inter process communication (IPC). Fatal errors are caused by malfunction of the effector (e.g. over-current in the motor or hitting a limit switch). System errors are caused by problems with IPC or process creation. The shell of each of the processes handles the detected classes of errors. Among others it informs the UI about the occurrence of an error.
The system solving the Rubik’s cube puzzle

The Rubik’s cube puzzle is a model task that one would expect a service robot to be able to execute. The system executing this task is composed of: two modified IRp-6 robots, each having six dof, two electric grippers with eye-in-hand cameras, one stand-alone camera, two force/torque sensors mounted in the wrists of the robots (Figure 2). A network of PC computers connected by Ethernet, QNX Neutrino ver.6.3 real-time operating system, MRROC++ based software supervised by the QNX RTOS – these are the building blocks out of which the controller was constructed. The task is executed in the following way. When the stand-alone camera spots the Rubik’s cube handed over by the operator, the VSP associated with this camera localizes the cube in 3D space – for that the knowledge of the dimensions of the cube are used. The VSP monitors the location of the cube at the rate of 25 Hz, delivering this data to the MP, and so the arm is visually servoed to the vicinity of the cube. The final approach to the grasping position is done when the cube is virtually at stand still (the arm should not inadvertently hit the operator).

In the near future this stage of servoing will be accomplished by using the in-hand camera, not the stand-alone one. Finally, the gripper jaws are closed. At this stage the state of the cube (i.e. the arrangement of the colour tiles on each face of the cube) is identified. This is accomplished by presenting each of the unobscurred faces to the camera mounted in the gripper that is not holding the cube, thus the cube must be regrasped by the two hands several times. Whenever a closed kinematic chain is formed (i.e. when the two manipulators simultaneously hold the cube) position-force control is used to avoid excessive forces to mount (i.e. one of the manipulators is partially compliant). Upon identifying the state of the cube the puzzle solving algorithm is invoked. It generates the sequence of the face rotations leading to a fully ordered state (i.e. each face holding the tiles of only one colour). This plan is realised by executing sequences of Move instructions which compose each face turn. The Move instructions use the wrist mounted force sensors to avoid jamming. Owing to the necessity of regrasping the cube, when the layers that have to be rotated are at right angles to each other, the roles of the manipulators switch. During each face rotation one of the manipulators executes the turning action, while the other is partially compliant, thus preventing jamming. Once the cube is fully ordered it is handed back to the operator. During the task execution either pure position control or position-force control is used, depending on the current task execution stage. Typically only those execution stages are position controlled, in which there in no simultaneous contact between the two end-effectors and the cube or between one of the end-effectors and the cube held by the operator. The stages, where such contact is present or expected to occur, are position-force controlled. The position-force controller is based on the task frame formalism (TFF) (Bruyninckx and Schutter, 1996). The controller itself is located in one of the threads of the EDP, but it can be commanded by either the MP (indirectly) or the ECP (directly) (Figure 3). Those commands are produced by the motion generators, either in the MP or the ECP. The command has several arguments. It specifies the coordinate frame relative to which both the position and force/torque are specified – this is either the robot base coordinate frame (absolute mode) or the tool coordinate frame (relative mode).

The tool coordinate frame is user defined relative to the wrist of the manipulator and can be changed by an adequate command during the operation of the system. Moreover, there is a selector that singles out the force/torque and position controlled directions relative to the frame being the argument of the command. Finally, the command execution time is specified.

Once the EDP receives the command it decodes and executes it. It divides its execution into servo sampling period increments. For each such increment it computes the position error in the position driven directions and the force/torque error in the force driven directions. The latter is transformed into an equivalent position error by multiplication by a scaling constant (this is equivalent to the introduction of damping) (Winiarski and Zielinski, 2005). The resulting position error is integrated with the current pose and transformed by the manipulator inverse kinematics into the joint error which is subsequently used by the joint controllers. The tasks of localising the cube and identification of its state (i.e. finding the positions of its tiles) are disjoint. First, the cube is localised for the purpose of grasping, and once grasped, it is presented to the cameras in such a way as to ensure fail-proof identification of its state. This can be accomplished by moving the cube to such a position that the camera image will mainly contain the cube. Localisation of the cube handed over by the operator is done by using the stand-alone camera, while the cube state identification is carried out by the in-hand cameras. It should be noted that the cube is very metamorphic, thus although it is only a cube it is not at all so easy to localise. The pattern of tiles (i.e. colours) on each face of the cube is more or less random, due to the enormous number of permutations of the puzzle. Thus, the features that are looked for are the coloured tiles on each of the cube’s faces, rather than the whole faces of the cube. Localisation of the cube in the image is based on matching colours and filtering out the excess of visual information. Once the features are found, it is possible to calculate the 3D pose of the object with respect to the camera coordinate system and subsequently the robot base coordinate system (the position of the camera relative to the

Figure 2 The two-handed system manipulating the Rubik’s cube and its gripper with an in-built camera
Because usually one of the faces dominates the image the system looks for the four corner tiles of that face and uses their centroids to localize the cube in the 3D space (Horaud, 1987), (the size of the cube is known a priori, so the depth information can be estimated relatively easily). Since, geometry, colour and distribution of the features of the cube is known, a colour image segmentation routine is used, which finds all adjacent quadrilateral and convex objects of almost homogeneous colour. All other shapes are disregarded.

Region segmentation of colour images is a crucial step in many vision applications. During that process individual pixels of an image are classified into one of a finite number of colour classes and pixels belonging to the same class are grouped together for further high-level processing. The most common solutions to this problem are: linear colour thresholding, nearest neighbour classification, colour space thresholding and probabilistic methods (Bruce et al., 2000). These approaches enable either precise (or human-like) colour segmentation at low-speed or real-time processing with relatively poor accuracy (in comparison with human performance). In the presented case, however, both accuracy and speed are vital, for the cube is to be identified, and followed by a robot manipulator in real time. An interesting solution to this problem was proposed in Bruce et al. (2000). This method of feature extraction has been modified, because it does not allow for colour variations and defines colours as constant rectangular blocks in the colour space, thus it often fails to properly separate neighbouring regions of the same colour. The modified method is far more robust to subtle colour changes. It is equivalent to applying the original method several times, but without the most computationally intensive part of the
algorithm (colour classification), which yields a significant performance gain. The most popular colour spaces used in vision applications are those in which chrominance is coded in two of the dimensions and intensity is coded in the third (HSV, HSI, YUV, etc.). These transformations minimize the most significant correlations between colours and the resulting space can be partitioned into separate hyperrectangles for all of the colour classes by constant value thresholding in each dimension. The implementation proposed in Bruce et al. (2000) uses a Boolean valued decomposition of the multidimensional threshold, which is stored in arrays. There are as many array elements as there are values of each colour components. Therefore, class membership of a pixel can be computed as a bitwise AND operation of the elements of each array (that results in two AND operations for a three dimensional colour space). After the colour classification and before region merging with an efficient tree-based union-find with path compression method the classified image is run length encoded to speed up further processing.

After colour segmentation an excessive number of regions is obtained, hence they are filtered by taking into account the following properties of the regions: area, circularity, boundary length, number of vertices and number of neighbours (secluded quadrilaterals are disregarded) (Figure 4).

Usually after filtration process only the cube tiles are left. It does not mean, however, that all the tiles are successfully detected. Occasionally some of the tiles are not detected and in a few situations there are some quadrilaterals found that do not belong to the cube. After segmentation and filtering the cube faces have to be identified in order to be able to attach a coordinate system to one of them, preferably to the corner tiles for higher accuracy. This is done by a number of simple heuristic rules such as: central tiles have the greatest number of neighbours, corner tiles have the least number of neighbours, tiles on one face are approximately of the same area. Even if some of the tiles are not detected the algorithm is able to find the biggest possible square present on the cube and calculate the cube’s pose with respect to the camera. With a cube image of roughly $100 \times 100$ pixels the localization precision is $\pm 1$ mm in the plane perpendicular to the camera and $\pm 5$ mm along the camera axis. Cube images smaller than that imply that the cube is outside the work-space of the robot, so are uninteresting. In the next stage of processing the pose of the cube is calculated from the image features (centroids of the four tiles of one face of the cube) that have been obtained by the above described method. All of those computations are the responsibility of the VSP associated with the stand-alone camera. Both the camera intrinsic and extrinsic parameters are taken into account. The usually large error between the current end-effector pose and the goal is subdivided into smaller increments by the motion generator of the MP. The VSP delivers to the MP the pose of the goal frame with respect to the global reference frame. The motion generator realises a position-based, end-effector open loop servo (a servo type in which the end-effector does not appear in the camera image or is not being recognised when it does appear) with stand-alone camera. It shifts the end-effector to the vicinity of the cube and waits simultaneously following the cube until it becomes motionless and then finally grasp it. Once grasped the cube is presented to the in-hand camera located in the other gripper for identification of the state. To identify the states of all faces the cube is regrasped several times, thus two VSPs associated with the two in-gripper cameras deliver the information about the state of each of the faces of the cube to the MP.

**Figure 4** Detected regions matching the cube colours and consecutive filtering results (left to right, top to bottom): initial segmentation with no filtering (1950 regions detected), area and boundary length filter (36 regions left), additional circularity and quadrilaterals filter (20 regions left), additional neighbours filter (18 regions left)
The Rubik's cube puzzle solver is located in the MP. Here, the information about the state of the cube is merged. Subsequently, the MP uses two algorithms to solve the puzzle. One is due to Korf (1997) and the other due to Kociemba (2006). The former uses Iterative Deepening A* with a heuristic estimate of the cost to reach the goal state contained in three tables (one for the corner pieces and two for six edge pieces each). If the optimal solution is not more than 16 steps away this algorithm suffices (it is claimed that the longest path to the goal state from any scrambled state of the cube should be no more than 20 steps). As the search for the solution with this algorithm for more than 16 moves becomes too long the system switches to Kociemba’s Two-Phase Algorithm (Kociemba, 2006), which gives a suboptimal solution in much shorter time. The algorithms produce their results in the form of a sequence of rotations of the faces. This is further transformed into a sequence of move instructions. The subsequences take into account the regrasping operations and that while rotating a face one of the effectors has to be partially compliant to avoid jamming. The experiments have shown that this approach never results in jamming or exerting an excessive force. The whole of the planning process takes less than 20 s – usually much less. The motions defined by this sequence are realised by the move instructions located in the MP. Only four distinct motions are needed, so only four motion generators are used: pure position control of an effector and three position-force generators: grasping the cube (in this phase the jaws of the gripper are closed while the effector accommodates its position), securing the cube (here the cube is slightly forced into the palm and finally the jaws are clenched), turning the face of the cube. The fifth generator, which forms the visual servo controller, is not used at this stage. It is used only while acquiring the cube from the operator.

Conclusions
Numerous experiments with diverse tasks (Zielinski et al., 2003, 2005) (not only the ones with the Rubik's cube presented here) show that MRROC++ is very well suited to producing controllers for prototypes of service robots. In choosing tasks both the form of cooperation and coordination was taken into account. The form of cooperation is defined by the requirements of the task itself, stating whether, and if yes, then how often do the effectors need to execute joint and simultaneous actions, thus cooperation can be: tight, loose, independent. Whereas the form of coordination is defined by the way that the control system operates, reflected in the frequency of synchronizing actions executed by the coordinating entity of the control system, thus coordination can be: continuous, sporadic, absent. The current version of the controller assumed that the two effectors will have to tightly cooperate and that the MP would have to coordinate them continuously. Hence, the move instructions of the MP used several motion generators to perform the job. In this case the ECPs only transmitted the set values computed in the MP to the EDPs for execution. However, during experimentation it turned out that in reality although tight cooperation is indeed necessary to perform the task, the coordination does not have to be exerted so often – it can be sporadic. Coordination is needed only when subtasks have to be switched (e.g. transition from a pure position control to the hybrid position-force one). By the way, this underscores that cooperation and coordination are in reality separate and independent criterions. Currently a new controller is being implemented, where the MP sporadically coordinates the two ECPs in which the move instructions use their own single-effector motion generators. This implies that the VSPs deliver their aggregated measurements to the ECPs and not to the MP, as it was in the previous solution. As the side effect of this change, motion primitives emerged. Previously the MP commanded detailed motions of the whole system. In the new version the MP only invokes certain behaviours of the effectors. Those behaviours are encoded in the motion generators of the ECPs. It should be stressed that there are tasks needing both tight cooperation and continuous coordination, which can be solved only by the previous structure. However, sporadic coordination suffices in the case of the Rubik's cube puzzle solving system.

In comparison to the code of the MRROC++ framework that was reported in Zielinski (1999) the code implementing the Rubik's cube solving controller had to be enhanced by including position-force control capability into the EDP. Moreover, specific code forming the kernels of the VSPs processing images obtained from the cameras had to be produced. Above all the code performing reasoning about the motions of the faces solving the puzzle had to be included. However, the general structure of the software has been retained, thus coding was enormously simplified in relation to staring from scratch. In this case, separately prepared modules were inserted into the general structure.

The performance of the components of the system is as follows: visual servo period (cube localisation) – 40 ms, solution of the puzzle – 0.1 to 10 s depending on how well is the cube scrambled recognition of the state of the visible face of the cube – 10 ms, position-force control sampling period – 2 ms. Time taken by the robots to grasp and turn a face is about 120 s – slow motion is used in the experiments for safety reasons.

References
Rubik’s cube as a benchmark validating MRROC++

C. Zielinski et al.

University, available at: http://playerstage.sourceforge.net/doc/Player-manual-1.5.pdf


Zielinski, C. (2001), “By how much should a general purpose programming language be extended to become a multi-


Corresponding author

W. Szyrkiewicz can be contacted at: W.Szyrkiewicz@ia.pw.edu.pl