3rd Int. Workshop on Robot Motion and Control, RoMoCo'02, 9–11 November 2002, Bukowy Dworek, Poland.

Reaction to Errors in Robot Systems

C. Zieliński
Warsaw University of Technology (WUT)
Institute of Control and Computation Engineering
ul. Nowowiejska 15/19, 00-665 Warsaw, POLAND, e-mail: C.Zielinski@ia.pw.edu.pl

Abstract

The paper[1] *analyzes the problem of error (failure) detection and handling in robot programming. First an overview of the subject is provided and later error detection and handling in* `MRROC++` *are described. To facilitate system reaction to the detected failures, the errors are classified and certain suggestions are made as to how to handle those classes of errors.*

1 Introduction

Although the problems of faults, failures and errors in control and computer systems have been studied extensively it is acknowledged that there is no common terminology regarding that subject, e.g. [17]. The main reasons are that those terms are strongly interrelated and difficult to define precisely. All of them pertain to expected or unexpected abnormal situations or difference between an expected state and measured value. This paper does not aspire to resolve this ambiguity and will rely on their intuitive and descriptive meaning. It will look more closely at how errors are dealt with in robot programming frameworks.

In general, software frameworks are application generators for a specific domain of problems [20]. This paper concentrates on robot programming frameworks, i.e. libraries of software modules (procedures, objects or processes) supplemented by design patterns for robot system controllers. A design pattern is composed of the so called frozen spots and hot spots. Frozen spots are the unalterable parts of the generated software and hot spots are the variable parts. Code of hot spots is delivered by the system builder or extracted from the library of ready modules. Although the terms: *framework* and *hot* or *frozen spots* are fairly recent, the concept of providing a library of ready or modifiable modules that can be assembled according to a certain pattern into a ready to use application (e.g. controller), is quite old. Formerly frameworks used to be called simply robot programming or control libraries or languages, but both of those terms are not adequate. A library does not imply an associated software pattern into which the modules should be inserted, and a language is usually associated with a specific grammar (e.g. syntactic structure). As general purpose languages, such as `Pascal` or `C` have been used as the development tools for those libraries, so no new language was being defined. Although initially specialised programming languages had been favoured, they lost they appeal, when it turned out that in the robotics domain the variability of equipment causes an ever greater demand for extensibility of those languages. Any extensions force a modification of the compiler or the interpreter of the language rendering the alteration more costly. Moreover, soon it became obvious that the specialised robot programming languages have to provide nearly all the capabilities of a general purpose programming language. Under those circumstances it was more reasonable to use a general purpose programming language and a library of modules specific to robot control and to define a general pattern according to which they should be assembled. Thus, although specialised robot programming languages gained considerable popularity (e.g. `WAVE` [28], `AL` [22, 31], `VAL II` [42], `AML` [29], `RAPT` [2, 31], `SRL` [4, 6] `TORBOL` [30, 31]) robot programming frameworks have been especially favored by the research community (e.g. `RCCL` [11], `Multi-RCCL` [15, 16] `KALI` [12, 13, 3], `ARCL` [8], `PASRO` [5, 6], `RORC` [33, 32], `MRROC` [33, 34], `MRROC++` [37, 39]). This paper concentrates on how the robot programming frameworks handle errors, so first the means of detecting errors must be discussed.

Robot systems use two kinds of sensors: internal (equivalent to interoceptors and proprioceptors in animals) and external (similar to exteroceptors in animals). The former are responsible for delivering in-

[1]This work was supported by CATID, WUT. Dr.W.Szynkiewicz supported the author in the implementation of `MRROC++`.

formation about the internal state of the the system (e.g. current in the servomotor or reading of a joint position encoder), and the latter for gathering information from the environment (e.g. tactile or vision data). Regardless of the source of information this data can be perceived by the system as expected or unexpected. To make this distinction clearer let us define unexpected situations as those that the control program does not handle by its routine measures. The expected information is utilised by the system to execute its task under normal conditions. The unexpected information has to be handled separately.

The unexpected situations are generally associated with diverse failures, errors or events unanticipated by the user. Any system should be able to detect those situations, but also an explanation of the cause of the error should be offered. It would be even better, if the system could respond to those events in such a way that it could continue functioning correctly. Obviously here we can pose a philosophical question: whether the system had encountered an unexpected situation at all, if it was able to handle it in such a way that it continued to realise its task? This is the main reason why the definition of unexpected events (and thus errors) is so difficult and relatively ambiguous.

Specialised robot programming languages initially treated errors in a similar way that errors were dealt with in interpreted general purpose programming languages. Early languages [6] dealt only with exceptional situations defined by the system (i.e. the system detects an a priori defined abnormal situation and the programmer can supply a handler for that situation) or the programmer (i.e. the user supplies a conditional expression that upon satisfaction triggers a supplied handler). The latter is equivalent to terminal condition monitoring (e.g. stopping a motion on exceeding a predefined force), thus it is difficult to decide wether it is an error at all. In `AL` [22] runtime errors required operator response, who could: start the program again, continue from a statement that followed the one that caused the error, or retry the statement that created the problem. In the case of `RPS` [27] an error message containing: the line number in which the error was detected, type of error, and the line numbers of the 10 preceding executed statements were displayed. Subsequently the user could use trace printouts, single-step execution or breakpoints to find the cause. Unfortunately in the current multi-process systems those debugging tools would be very ineffective. Error recovery in early systems consisted in: retrying the current statement, continuing from the next one, restarting the program, aborting or trying an alternate program. In industrial systems, such as `VAL II` [42] special instructions (e.g. `REACTE error_handling_program`) could be inserted into the code of the user's program. If such an instruction was not activated upon detecting an error the program halted. On the other hand, if `REACTE` was active, `error_handling_program` was executed. A function `ERROR` was used to find the cause of the error. `RETURN` within the handler caused the reexecution of the instruction that caused the problem. An error within the handler aborted the program all together.

Relatively much attention has been concentrated on error recovery in the case of task planning, especially dealing with assembly, disassembly and obstacle avoidance. For instance in [14] objects in the work cell model can be in one of the three states: valid (known), unvalidated (unknown, because a transformation is in progress) or invalid (mislocated). The last state triggers error recovery and if that fails replanning is performed. In [19] the assembly process is modelled as a discrete event system, where tokens within a Petri net represent contact points between objects (vertices, edges and surfaces). The assembly process is broken down into transitions between states of contact. The trajectory of desired markings and transitions in the Petri net forms the assembly strategy. An error is detected when an event occurs that causes a marking that is not in the prescribed trajectory. The process monitor detects and identifies errors and thus a new path through the Petri net can be found – in effect implementing error recovery. In [7] a generic architecture is presented, that provides functions for dispatching actions, monitoring their execution, diagnosing failures and recovering from errors. The monitoring function using sensors tracks the execution of assembly plans and detects non-nominal feedback – both in discrete event and continuous domains. The generated plan contains the information about what sensor feedback should be tracked. Subsequently a diagnosis function confirms if a failure has been detected and tries to explain it. To do that it uses the model of the task, the system and the environment. Errors have been classified into: system faults (abnormal behavior of the system hardware or software – treated as unrecoverable), external exceptions (abnormal occurrences in the cell environment – e.g. unexpected objects) and execution failures (deviation of the state of the world from the expected state of execution of actions – e.g. collision, part missing). The recovery function tries to find a strategy for bringing the execution to a nominal state. [17] distinguishes two main recovery strategies: backward error recovery (finding a previously traversed error free state) and forward error recovery strategy (looking for an error free state that the system should eventually reach). The task level of the Aramis system (A Robot And Manufacturing Instruction System) [18] sets reference values for objects, what is equivalent to commanding the objects to change their state. The execution level acts as a virtual servomechanism matching the real world and world model states. Sometimes this is not possible, so an error is detected and error recovery has to be implemented. Unfortunately from the point of view of the task level the control level does not behave in transactional way, i.e. it should either attain the requested state or report a failure. Sometimes some intermediate state is reached, e.g. half-done seam neither can be finished nor undone. This leads to the necessity of simultaneous ridding of the work-cell of the faulty workpiece and bringing the controller into a normal state from which the task can be continued, i.e. performing resynchronisation.

Closed structure systems use specialised programming languages for expressing the user's program. The program is compiled and executed or interpreted on an instruction by instruction basis. The only sources of errors are: the user's program, hardware failure or an unexpected event in the environment. In the case of open structure systems that are created by using programming frameworks the situation is aggravated by the possibility that the designer delivers a flawed module or inserts a module in violation of the rules of the general pattern. Because of that error detection and diagnosis are of utmost importance in frameworks – this capability has to be built into them.

$\mathsf{G^{en}oM}$ (Generator of Modules) [9, 1] is a tool for automatic generation of frozen spots and the shells of the hot spots of the control system execution level (functional level) from the builder's specifications of modules. A module is an entity providing services upon request. Control requests influence the execution of a service (e.g. parameterize, abort, interrupt it), whereas execution requests start services. A running service, called an activity, exchanges data with other modules through posters. Services are provided by execution tasks. Besides one or more execution tasks, a module contains a single control task that is responsible for asynchronous communications with the clients of the module, checking the validity of the incoming requests and for initiating the service execution. Execution tasks and the control task contain hot spots called codels (code elements) – those are delivered by the system builder. Currently codels are implemented as two kinds of `C` functions, returning either: status (`OK` or `ERROR`) or the designator of the next state of the module (e.g. `START`, `EXEC`, `END`, `FAIL`, `TERMINATE`, `ZOMBIE`). The normal execution sequence of the states is: `START` → `EXEC` → `END` → `TERMINATE`. `END` is reached when an activity has been finished, but this activity can be invoked once again. `TERMINATE` means that the activity is no longer available. `FAIL` results when the module cannot cope with an error. In that state a cleanup can be performed by the codel and the module goes into the `ZOMBIE` state – becomes unavailable. This is a situation that should not arise after debugging the module. During normal life of a module, even if it detects abnormal functioning of the hardware, it has to go into `TERMINATE` state, within which it is able to inform other modules requesting a service that the hardware is in an abnormal state. At the end of each service a report is returned to the client. For nominal execution the report contains `OK`. For situations in which errors were detected reports that had been listed in the description file of the module are used. This enables some recovery actions to be undertaken by other modules or upper control layers. It should be noted that automatic generation of the controller from a specification provided to $\mathsf{G^{en}oM}$ significantly reduces the possibility of introducing bugs into the system software.

2 MRROC++ based controllers

This section describes the general structure of the object-oriented version of the Multi-Robot Research-Oriented Controller: `MRROC++` framework. It has been used to control ASEA type IRb-6 robots (one of them mounted on a track), prototype serial-parallel structure RNT robot [23, 41], and a prototype fast robot – Polycrank [24]. All of those robots require specialised hardware controllers [38]. Force, ultrasonic and infrared sensors, CCD cameras and a conveyor belt have been included in the implemented systems. `MRROC` and `MRROC++` have been used to implement, e.g.: cooperative transfer of a rigid body by two robots [35], calibration of robot kinematic models using two high precision electronic theodolites [10], engraving inscriptions in soft materials by a robot equipped with a milling machine [21], polishing metals, grasping moving objects detected by a 3D image capture ultrasound matrix [25, 26].

`MRROC++` is a robot programming framework for designing multi-robot controllers tailored to the tasks at hand. Thus it is a library of software modules (i.e. classes, objects, processes and procedures) that can be used to construct any multi-robot system controller. This set of ready made modules can be extended by the user by coding an extra module in C++. The freedom of coding is, however, restricted by the general structure of the framework. New modules have to conform to this general structure. Theoretical reasons for selecting this structure can be found in [40, 36, 33]. It is divided into three layers: hardware dependant,

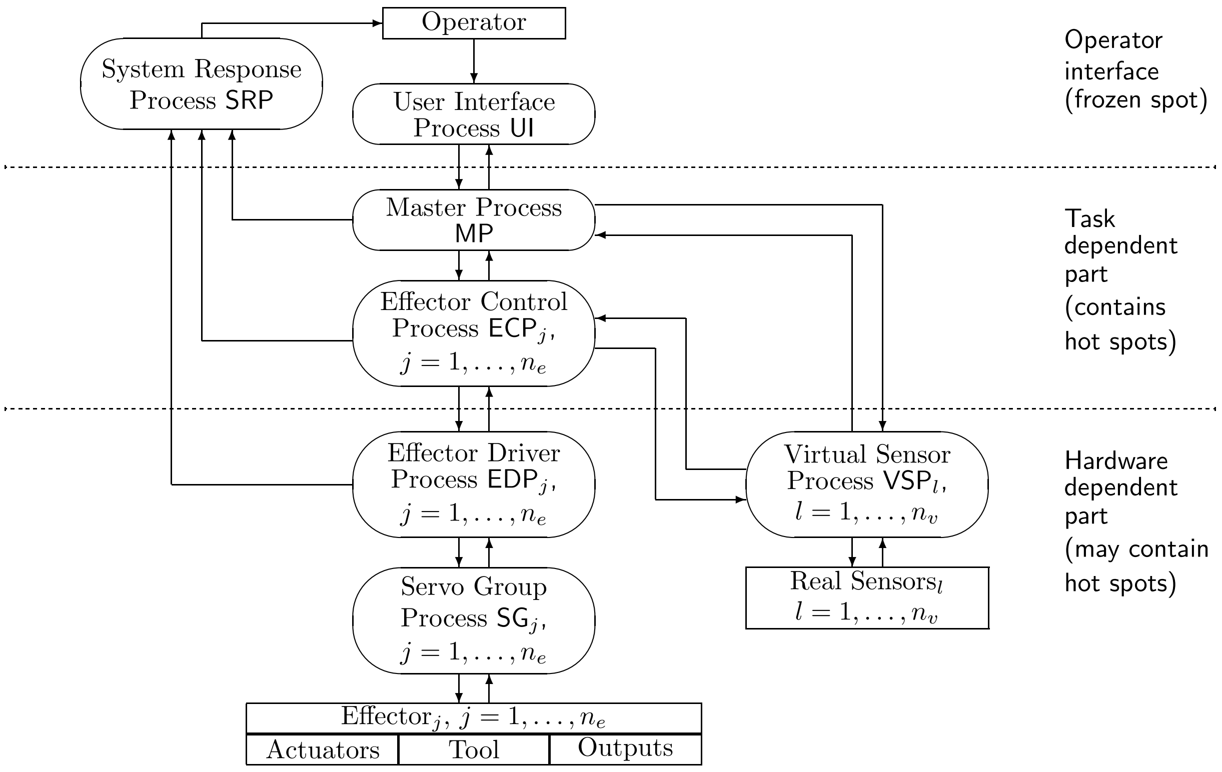
Operator
System Response Process SRP
User Interface Process UI
Operator interface (frozen spot)
Master Process MP
Task dependent part (contains hot spots)
Effector Control Process ECP_j, $j = 1, \dots, n_e$
Effector Driver Process EDP_j, $j = 1, \dots, n_e$
Virtual Sensor Process VSP_l, $l = 1, \dots, n_v$
Hardware dependent part (may contain hot spots)
Real Sensors$_l$ $l = 1, \dots, n_v$
Servo Group Process SG_j, $j = 1, \dots, n_e$
Effector$_j$, $j = 1, \dots, n_e$
Actuators
Tool
Outputs

Figure 1: Structure of a MRROC++ based controller

task dependant and the operator interface (fig. 1).

The operator interface layer is composed of User Interface Process UI and System Response Process SRP. They should be treated as frozen spots of the framework. UI is responsible for delivering the operator commands to the system (e.g. STOP, PAUSE, EXECUTE, RESUME). SRP collects the messages from all the other processes of the system, decodes them, and displays them in a window on the screen of the monitor. The messages pertain to the current system states and the detected errors.

User's task is coordinated by a single process called Master Process MP. It is assumed that there are n_e effectors in the system, where an effector can be, e.g.: a robot, cooperating device or a limb of a walking machine. From the point of view of the executed task MP is the hierarchically highest process, as UI and SRP just provide supplementary services. MP is responsible for trajectory generation in multi-effector systems, where the effectors cooperate tightly. In the case of loose cooperation it just synchronises the effectors from time to time. The general specification of the task for all the effectors is concentrated in the hot spots of MP.

Each effector (either a robot or a cooperating device) has three processes controlling it: Effector Control Process ECP, Effector Driver Process EDP and Servo Group Process SG. The first one is responsible for the execution of the user's task dedicated to this effector, and the other two for direct control of this effector. The hot spots of ECP contain the part of the task specification that pertains to a single effector. EDP is responsible for coordinate transformations, tool definitions, etc., thus it presents the effector from the kinematical point of view. As in MRROC++ the user can provide his/her own kinematic models or kinematic model correctors for certain areas of the work-space, this process also can contain hot spots. SG is responsible for servo control. Because MRROC++ can maintain many switchable servo control algorithms with diverse sets of parameters, and both the algorithms and the parameters can be provided by the users, this process also can contain hot spots. Nevertheless, for the majority of tasks EDP and SG are not modified.

Data obtained from real (i.e. hardware) sensors usually cannot be used directly in robot motion control. For instance, to control the arm motion, only the location of the centre of gravity of an object to be grasped would be necessary. In the case of such a complex sensor as a camera a bit-map has to be processed to obtain the above mentioned location. In some other cases a simple sensor in its own right would not suffice to control the motion (e.g. a single strain gauge), but several such sensors deliver meaningful data. The process of extracting meaningful information for the purpose of motion control is named data aggregation and is performed by a virtual sensor. Data aggregation is done by Virtual Sensor Processes VSPs. It is assumed that there are n_v virtual sensors in the system. Both MP and ECPs can use the data provided by the VSPs.

The user's parts of both the MP and the ECPs are composed of the `Move` and `Wait` instructions (procedures). On the MP level these instructions take (as their arguments) lists of robots and virtual sensors. On the ECP level, for each ECP only a single robot exists, so this robot and a list of sensors are the arguments of these instructions. Moreover, the `Move` instruction has an object named trajectory generator as the third argument, and the `Wait` instruction has an object named the condition as the last argument. The generator is responsible, on the MP level, for the generation of trajectories of the end-effectors of all the robots on the list forming the argument of the `Move` instruction. On the ECP level the generator creates a trajectory for a single robot. The condition, being the argument of the `Wait` instruction, if true, terminates the waiting, and if not, causes the system to pause. For each `Move` and `Wait` instruction the user writes in `C++` his/her own generator and condition objects. In this way, usually only very small portions of MP and ECPs have to be rewritten when the task changes. The modifications are cumulated in the separate code of specific generators and conditions. Errors are dealt with within the whole system by exception handling, so the user needs not deliver the program code responsible for that. Neither need he/she worry about the code dealing with the inter-process communication.

3 Handling of errors in MRROC++

To the frozen spots of the system the user appends the code forming hot spots (i.e. invocations of `Move`, `Wait` instructions, trajectory generators, initial conditions and task specific `C++` code). This process is error-prone. Moreover, it cannot be assumed that the hardware will run without failure. The processes of the system can be distributed over a network of computers, so inter process communication problems can arise. Both the code provided by the user and the code of the frozen spots has to be able to detect errors. Upon detecting an error `MRROC` used function return values to inform the the higher control layers within a process about that fact. For the same purpose `MRROC++` throws exceptions that by default are caught at the uppermost process level. From that point the information about the errors is further propagated by inter process communication (IPC) means, i.e. message passing, if only IPC still operates.

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 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 standard procedure of dealing with errors in `MRROC++` is very simple. For any piece of code that can generate or detect an abnormal situation the user provides an error condition testing instruction. If the error condition is fulfilled an exception is thrown with an argument describing the reason of the problem (class of error and its specific type) depending both on the direct cause of the error and the place of its occurrence. Moreover, at the uppermost process level an extra `case` in the `switch` instruction of the `catch` section is inserted. This will deal with this error adequately. Only SG, which has a relatively fixed structure (only new servo-control algorithms can be inserted) does not use this method . It uses function arguments to return error codes. At the soonest possible contact with EDP (the contact is always initiated by EDP) this code is transferred. Detection of an error does not free SG from performing servo control the best it can.

It should be pointed out that there always exists a limit to which a system can endure failures. In the case of robust systems this limit is located fairly high, and so they can handle a wide spectrum of unexpected events, but even those systems can be mutilated or even destroyed by deliberate malicious actions or random failure of their vital components. It is economically viable to place that limit in such a way that, on the one hand, the system will be robust enough to carry out its task under reasonable conditions, and on the other hand, the fabrication of this system is cheap enough to attract buyers.

The majority of `MRROC++` based controllers dealt with the detected errors in the same way. The fatal and non-fatal errors caused the error messages to propagate throughout the system, the executed user's task was interrupted, and each of processes was left in a state enabling the operator to resume the actions of the system as a whole. The exceptions were caught at the uppermost level of each process. From there the error message was dispatched to SRP and the nearest superior in the process hierarchy (this was done at the earliest contact initiated by the superior), e.g. EDP informed ECP and ECP informed MP. In this way all the processes participating in the execution of the task were informed about the error and each one of them in turn reported the error to SRP, which displayed adequate messages for the operator to act upon. The operator could work out the probable cause of the error, because each process reported the error as it perceived it, thus enabling the user to figure out the state that each of the processes was in when the failure occurred.

Obviously, as exception handling is used, and the user is the one who provides the user's program, the exceptions can be caught before they reach the topmost level of the process. Thus the user can change the standard method of dealing with particular errors or more probably with certain classes of errors. There are a few standard methods of treating error recovery, e.g.:

- retrying the action that failed,
- retrying the action that failed, but with a different set of parameter values,
- trying another action.

It should be noted that the classes of errors pertain to the ability of the system to retain its own composure and not to the ability to correct them. For instance, a non-fatal error detected as a computational error (e.g. negative argument of a `sqrt` function within inverse kinematic procedure), but resulting from an object that is to be grasped being out of the workspace, cannot be corrected by the system (the object is simply too far), but the system should retain such a state that it will be able to execute other actions. The ability of correcting fatal errors depends on the redundancy of the system, e.g. malfunction of the manipulator in a single robot system cannot be remedied, but in a multi-robot one sometimes can be corrected. Thus, whether errors can be handled depends also on the overall structure of the system. Nevertheless, after a fatal error the system should be left in such a state that at least it is able to inform the operator about the reason for its malfunction – it should not disintegrate.

System errors cannot be dealt with by the above described methods. They are caused by the disintegration of the control system itself, so it is not realistic to assume that the inter process communication will be intact at that moment. Hence in MRROC++ the process that detects such an error displays an error message by itself and subsequently aborts. The operator is left with the task of cleaning up. In fixed structure systems such a situation usually does not arise (except when some piece of hardware is destroyed), but we are dealing with an open structure system. The user can insert his/her own code, so there is an unlimited opportunity of introducing bugs, including the ones leading to disasters (e.g. killing off one of the processes). A well tested system will be free of such bugs, but we are speaking about a situation when the system is under development. At that stage the operator has to be informed about the mishap, so that the program can be debugged quickly.

4 Conclusions

Robot programming frameworks have to provide very accurate error detection capabilities, because the user builds the system out of modules with partially or fully unknown inner workings. Thus it is vital that the assembled system informs the user upon detecting any abnormality, so that it can be corrected. Moreover, framework designers should try to provide error diagnostic capabilities. Otherwise the user, after an error is detected, is left to guess what went wrong and that in a complex system is a formidable task. As error recovery strategies heavily depend on the executed task, they have to be provided by the user of the framework. Nevertheless, a provision should be made by the framework designer to enable the user to add on such error recovery procedures in a simple way.

It should be noted that in some systems there are situations where some problems might go on undetected. For instance, if the gain of the servo is low or the friction in the mechanism causes slight slip-stick, the system will function, but the quality of its actions will be compromised. Such situations are not treated as errors, but a good robot programming framework should be equipped with tools that can monitor the operation of the system and provide adequate reports. The analysis of those reports should lead to the detection of the problems and finding a remedy. Usually such problems arise at the lowest control level. In MRROC++ the Servo Group Process is provided with the capability of constant monitoring of variables selected by the user. Usually a single value of a variable is not meaningful. Only trends give adequate hint as to what is the cause of a problem. Unfortunately it is not possible, due to memory limitations, to collect the whole history of changes of variable values (e.g. current servo position or its set-value changes at sampling rate). Thus a cyclic buffer is used that is able to hold the last 10 – 30 seconds of selected data. As file operations are rather slow, SG which is responsible for servo control, cannot dump the contents of the cyclic buffer to a file directly. Thus Reader Process (RP) is created by SG upon system initialisation. RP is capable of interacting with the operator, so he/she can select the data that is needed. Upon the operator request made to the RP, it contacts SG which transmits the contents of the cyclic buffer using standard IPC means. Subsequently RP (at low priority) dumps the selected data into a file. The file is in a text format so any reasonable plotting program can be used to display the trends.

MRROC++ in itself does not ensure that the created controller will be error free, but by providing a pattern (ready structure) and at least some ready made and tested software components makes the introduction of errors less likely. Besides that it makes the system designer aware of the three error classes, thus enables him/her to group the specific errors accordingly. In doing so the programmer decides the system response to a specific error. Moreover the handling of those errors is provided within the pattern of the framework, so the designer does not have to busy herself with that. The addition of the above mentioned servo tracking capability enables fine tuning of the servo control algorithms. All those provisions simplify the life of the control system designer, but by no means does that imply that nothing will hinder the implementation. Much further research work is required to provide satisfactory solution to the issues brought forward by this paper.

References

[1] Alami R., Chatila R., Fleury S., Ghallab M., Ingrand F.: *An Architecture for Autonomy*. Int. J. of Robotics Research. Vol.17, no.4, April 1998, pp.315–337.

[2] Ambler A. P., Corner D. F.: *RAPT1 User's Manual*. Department of Artificial Intelligence, University of Edinburgh, 1984.

[3] Backes P., Hayati S., Hayward V., Tso K.: *The KALI Multi-Arm Robot Programming and Control Environment*. Proc. NASA Conf. on Space Telerobotics, 1989.

[4] Blume C., Jakob W.: *Design of the Structured Robot Language (SRL)*. in: *Advanced Software in Robotics*, Eds. Danthiene A., Géradin M., Elsevier, North Holand, 1984. pp.127–143.

[5] Blume C., Jakob W.: *PASRO: Pascal for Robots*. Springer-Verlag, Berlin 1985.

[6] Blume C., Jakob W.: *Programming Languages for Industrial Robots*. Springer-Verlag, 1986.

[7] Camarinha-Matos L. M., Lopes L. S., Barata J.: *Integration and Learning in Supervision of Flexible Assembly Systems*. IEEE Transactions on Robotics and Automation. Vol.12, No.2. April 1996. pp. 202–219.

[8] Corke P., Kirkham R.: *The ARCL Robot Programming System*. Proc. Int. Conf. Robots for Competitive Industries, Brisbane, Australia, 14-16 July 1993. pp.484-493.

[9] Fleury S., Herrb M.: G^{en}oM *User's Guide*. Report, LAAS, Toulouse, No. 01577, December 2001.

[10] Frączek J., Buśko Z.: *Calibration of Serial and Serial-Parallel Robot Systems Using Electronic Theodolites and Error Correction Procedure*. 10th World Congress on Theory of Machines and Mechanisms, Oulu, Finland, 20–24 July 1999. Vol.3. pp.972–977.

[11] Hayward V., Paul R. P.: *Robot Manipulator Control Under Unix RCCL: A Robot Control C Library*. Int. J. Robotics Research, Vol.5, No.4, Winter 1986. pp.94-111.

[12] Hayward V., Hayati S.: *KALI: An Environment for the Programming and Control of Cooperative Manipulators*. Proc. American Control Conf., 1988. pp.473-478.

[13] Hayward V., Daneshmend L., Hayati S.: *An Overview of KALI: A System to Program and Control Cooperative Manipulators*. In: *Advanced Robotics*. Ed. Waldron K., Springer-Verlag, 1989. pp.547–558.

[14] Kelly R. B.: *Knowledge-Based Robot Workstation: Supervisor Design*. In NATO ASI Series Vol.66: *Sensor-Based Robots: Algorithms and Architectures*. Ed. C. S. G. Lee. Springer-Verlag, Berlin, 1991. pp.107–128.

[15] Lloyd J., Parker M., McClain R.: *Extending the RCCL Programming Environment to Multiple Robots and Processors*. Proc. IEEE Int. Conf. Robotics and Automation, 1988. pp.465-469.

[16] Lloyd J., Hayward V.: *Real-Time Trajectory Generation in Multi-RCCL*. J. of Robotics Systems, 10 (3), 1993. pp.369–390.

[17] Loborg P.: *Error Recovery in Automation*. AAAI'94 Spring Symposium on Detecting and Resolving Errors in Manufacturing Systems, Stanford, California, USA. 1994.

[18] Loborg P., Thörne A.: *Manufacturing Control System Principles Supporting Error Recovery*. AAAI'94 Spring Symposium on Detecting and Resolving Errors in Manufacturing Systems, Stanford, California, USA. 1994.

[19] McCarragher B.J.: *Task Primitives for the Discrete Event Modeling and Control of a 6-DOF Assembly Tasks*. IEEE Trans. Robotics and Automation. Vol.12, No.2. April 1996. pp.280–289.

[20] Markiewicz M. E., Lucena C. J. P.: *Object Oriented Framework Development*. ACM Crossroads, 7(4), 2001, http://www.acm.org/crossroads/xrds7-4/frameworks.html

[21] Mianowski K., Nazarczuk K., Wojtyra M., Szynkiewicz W., Zieliński C., Woźniak A.: *Application of the RNT Robot to Milling and Polishing*.

Proc. of the 13-th CISM-IFToMM Symposium on Theory and Practice of Robots and Manipulators Ro.Man.Sy'13, 3–6 July 2000, Zakopane, Poland.

[22] Mujtaba S., Goldman R.: *AL Users' Manual*. Stanford Artificial Intelligence Lab., AI Memo 323, January 1979.

[23] Nazarczuk K., Mianowski K., Olędzki A., Rzymkowski C.: *Experimental Investigation of the Robot Arm with Serial-Parallel Structure*. Proc. 9-th World Congress on the Theory of Machines and Mechanisms, Milan, Italy, 1995, pp. 2112-2116.

[24] Nazarczuk K., Mianowski K.: *Polycrank – Fast Robot Without Joint Limits*. Proc. of the 12-th CISM-IFToMM Symposium on Theory and Practice of Robots and Manipulators Ro.Man.Sy'12, 6-9 June 1998. Springer-Verlag, Wien, pp.317-324.

[25] Pacut A., Brudka M., Jaworski M.: *Neural Processing of Ultrasound Images in Robotic Applications*. Proc. of the IEEE Int. Workshop on Emerging Technologies, Intelligent Measurements and Virtual Systems for Instrumentation and Measurements ETIMVIS'98, St. Paul, USA, May 1998. pp. 59–66

[26] Brudka M., Pacut A.: *Intelligent Robot Control Using Ultrasonic Measurements*. Proc. of the 16-th IEEE Instrumentation and Measurement Technology Conference IMTC/99, Venice, Italy, vol. 2, May 1999. pp. 727–732.

[27] Park W. T.: *The SRI Robot Programming System (RPS)*. Proc. 13th International Symposium on Industrial Robots, Chicago, USA, 1983. pp.12.21–12.41.

[28] Paul R.: *WAVE – A Model Based Language for Manipulator Control*. The Industrial Robot, March 1977. pp.10–17.

[29] Taylor R. H., Summers P. D., Meyer J. M.: *AML: A Manufacturing Language*. International Journal of Robotics Research, Vol. 1, No. 3, 1982. pp.842–856.

[30] Zieliński C.: *TORBOL: An Object Level Robot Programming Language*. Mechatronics, Vol.1, No.4, 1991. pp.469-485.

[31] Zieliński C.: *Object Level Robot Programming Languages*. **In:** *Robotics Research and Applications*. Ed.: A. Morecki et.al., Warsaw 1992. pp.221-235.

[32] Zieliński C.: *Flexible Controller for Robots Equipped with Sensors*. 9th Symp. Theory and Practice of Robots & Manipulators, Ro.Man.Sy'92, 1-4 Sept. 1992, Udine, Italy, Lect. Notes: Control & Information Sciences 187, Springer-Verlag, 1993. pp.205-214.

[33] Zieliński C.: *Robot Programming Methods*. Publishing House of Warsaw University of Technology, 1995.

[34] Zieliński C.: *Control of a Multi-Robot System*, 2nd Int. Symp. Methods and Models in Automation and Robotics MMAR'95, 30 Aug.–2 Sept. 1995, Międzyzdroje, Poland. pp.603-608.

[35] Zieliński C., Szynkiewicz W.: *Control of Two 5 d.o.f. Robots Manipulating a Rigid Object*, IEEE Int. Symp. on Industrial Electronics ISIE'96, 17–20 June 1996, Warsaw, Poland. Vol.2, pp.979–984.

[36] Zieliński C.: *Object-Oriented Robot Programming*, Robotica, Vol.15, 1997. pp.41–48.

[37] Zieliński C.: *Object–Oriented Programming of Multi–Robot Systems*, Proc. 4th Int. Symp. Methods and Models in Automation and Robotics MMAR'97, 26–29 August 1997, Międzyzdroje, Poland, pp.1121–1126.

[38] Zieliński C., Rydzewski A., Szynkiewicz W.: *Multi-Robot System Controllers*. Proc. of the 5th International Symposium on Methods and Models in Automation and Robotics MMAR'98, 25–29 August 1998, Międzyzdroje, Poland, Vol.3, pp.795–800.

[39] Zieliński C.: *The MRROC++ System*, 1st Workshop on Robot Motion and Control, RoMoCo'99, 28–29 June, 1999, Kiekrz, Poland. pp.147–152.

[40] Zieliński C.: *By How Much Should a General Purpose Programming Language be Extended to Become a Multi-Robot System Programming Language?*. Advanced Robotics, Vol.15 No.1, 2001. pp.71–95.

[41] Zieliński C., Szynkiewicz W., Mianowski K., Nazarczuk K.: *Mechatronic Design of Open-Structure Multi-Robot Controllers*. Mechatronics, Vol.11 No.8, November 2001. pp.987–1000.

[42] *User's Guide to VAL II: Programming Manual*. Ver.2.0, Unimation Incorporated, A Westinghouse Company, August 1986.