Self-Management in a Robot Control Architecture

Jochen Maß, Jens Steiner, Jürgen Hesselbach, Ursula Goltz

Abstract—To reach good maintainability, extendibility and reusability of components in the design of robot control architectures is a major challenge. In this paper the authors introduce an architecture for handling and assembly applications featuring self-management techniques as an approach to tackle these problems.

The existing architecture features a layered design, dividing robot-specific components from those which are reusable for a class of applications. Present and future concepts of self-management applied to this architecture such as self-configuration, self-optimization, self-healing and self-protection are outlined. These properties are realized by the integration of self-managers within crucial system components in a future distributed version of the architecture.

I. INTRODUCTION

The ever-growing complexity of robot control architectures has been identified as a major challenge. Maintainability, extendibility and reusability become problems besides the original realization task. Intricate errors in large, monolithic systems often lead to huge efforts in order to prevent the robot control from unpredictable or disastrous behavior. Several, mostly nature inspired, approaches have been proposed to tackle complexity and entailled problems. In the year 2001, IBM drafted their vision of Autonomic Computing and recommended autonomy on different hierarchy levels in a system. They later detailed this idea [5] and outlined how cooperating management components for different system parts could monitor, analyze and manipulate system behavior in their area of responsibility and realize self-managed systems, striving for optimization and capable to configure, heal and protect themselves. Autonomic Computing has become a huge research area and similar endeavors such as Organic Computing [11] have been brought to life lately. A model driven development (MDD) of the software parts of a system can also help to catch some of the complexity and can be the foundation for miscellaneous quality assurance measures [15].

The authors introduce present and future concepts of self-management applied to a robot control architecture designed for sensor-guided handling and assembly tasks. This architecture is used for the control of parallel kinematic machines developed by the Collaborative Research Center 562 of the German Research Foundation (see Fig. 1 for examples). In the following section, the architecture in focus is described. It features the capability to perform modern control algorithms in both, internal and external loops. Internal loops such as the computed-torque approach and feedback linearization [14] are required to benefit from the robot’s dynamic performance advantage compared to a classic approach. External loops for compliant motion with respect to e.g. a force sensor or a vision system [12] are crucial in light of performing error-tolerant assembly tasks [16].

Furthermore, the architecture features a powerful task frame formalism supporting skill primitive programming. Introduced in 1981 by Mason [10] and taken up by deSchutter [13], the task frame formalism is a major step towards task orientated assembly programming [17]. The difference compared to traditional motion description is that the motion is defined in the task frame, which is allowed to become placed arbitrarily in the robots’ work cell. Our implementation of the skill primitive programming interface, a concept which first was proposed by Hasegawa in 1992 [2], allows the robot operator to define a separate control algorithm for each of the six d.o.f. the task frame provides, causing hybrid moves of the robot’s end effector [7].

An important design principle applied to the architecture is the separation of algorithms and communication as well as OS issues, respectively. This allows for an easy integration of self-management features. Most complex control systems are designed in a collaborative environment, incorporating experts in different fields of knowledge. Hence, the control engineer should not be constrained by communication or operating system restrictions. Our solution to this problem is the application of a communication middleware. Further improvements can be achieved by providing a complete,
platform independent programming framework [9], featuring all required (abstract) interfaces as well as communication and synchronization mechanisms as presented in section II-B.

II. ARCHITECTURE DESIGN

In this section, the robot control architecture, where self-management features are in the process of integration, is described. A basic functional overview is given in Fig. 2. The control scheme named “RCA562” (Robot Control Architecture of Collaborative Research Center 562) consists of two layers. On the right, the robot specific layer is depicted: It shows the subordinate drive controller receiving a fully qualified (not only the desired pose \( x_d \) but also its derivatives \( \dot{x}_d, \ddot{x}_d, \)) Cartesian trajectory information as its input. This layer can be designed and put into operation independently from the layer depicted on the left. This layer shows the robot-independent trajectory generator, which is the central element of this control architecture. It consists of two types of reusable, task-specific modules: The control core together with its coordination unit interacts with the skill primitive programming interface and dynamically establishes communication paths between the core and the attached motion and sensor modules according to the motion task specified by the current skill primitive. This is possible by using the middleware MiRP A-X designed especially to meet these demands. This middleware and its features are briefly explained in subsection II-B.

1) Motion Modules: The important data elements of the interface to the motion modules are discussed in the following. The actual hand frame position, velocity and acceleration are passed to the motion modules. Additionally, the desired values for the control algorithm and an array of flags, which d.o.f. of the task frame should be controlled, are passed. These data are followed by a list of pointers to shared memory areas where the module can read its required sensor data that it has previously announced to be required by its profile. The mentioned profiles are needed for autonomous configuration of the control system as described in section II-C.

The motion module’s algorithm returns an array of flags representing the d.o.f. for which a result has been calculated successfully and the calculated output position, velocity and acceleration. The required motion modules for the execution of a specific skill primitive are notified by a per-module command message. This allows the motion modules to operate on distributed computing platforms in order to balance computational load. At the end of each trajectory generator execution cycle, the control core determines which modules’ output values are selected to be transformed into the robot’s base frame, using the success flags. If none of the modules has been able to calculate an appropriate result for a d.o.f., a braking trajectory generator [4] is used by the core. All these functionality is decoupled from the design of the trajectory control algorithms by utilizing abstract interfaces as shown in Fig. 3.

To maintain deterministic behavior and to become immune against failure of motion modules and sensor modules or
even process termination on distributed computing platforms, the control core needs to get the CPU privileged to the other modules at crucial points of time in the control cycle. This is achieved by making use of the priority inheritance concept of the real-time operating system.

2) Sensor Modules: The interface to the sensor modules is simpler. A sensor module provides one or more data structures in a shared memory area maintained by the communication middleware. This structure contains a flag, indicating if the sensor module is requested to provide a sensor value and the returned sensor value itself. They are activated by a broadcast message in the beginning of a trajectory generator cycle. Woken up on the message reception, they could calculate signal processing algorithms if necessary or just copy sensor information from a hardware interface into the above mentioned shared memory area. Here, it does not matter if the measurement comes from a real sensor or just an internal state of the drive controller, e.g. in order to perform adaptive control using the tracking error as a performance criterion. In the latter case these sensors are called virtual sensors and are driven by the controller itself. Before becoming blocked again, the sensor modules decrease a counter previously set up by the control core with the number of requested sensor values needed by the motion modules and the robot program. When the counter becomes zero, the control core subsequently activates the motion modules.

B. Real-time Middleware MiRPA-X

In order to facilitate the development of a control framework as it is described in the preceding subsections, the usage of a flexible and highly efficient communication and synchronization infrastructure is necessary. As it is the key functionality of all commercial middleware approaches (CORBA, DCOM, etc.), MiRPA-X can be used simply as a communication object server to handle data and procedure requests in robot control systems transparently [1]. In order to meet hard real-time requirements derived from the application within complex parallel robot systems, it provides additional mechanisms for hard-real-time data processing, such as shared memory communication as well as process scheduling and synchronization.

Fig. 4 shows a schematic overview of the communication system used to interconnect the control system components. Besides, the control modules (motion and sensor modules) mentioned in section II-A, exemplary further modules are situated on the application level. In order to facilitate real-time data exchange between the central control unit and external robot components (drives, sensors and actuators), a high-speed communication bus (currently based on the IEEE 1394 standard) is used together with a specially tailored communication protocol (Industrial Automation Protocol, IAP). A detailed description of the overall communication system is given in [6].

1) Message-Based Data Transfer: MiRPA-X uses QNX message passing internally as the basic mechanism for putting synchronous and asynchronous communication services into reality. According to this, MiRPA-X can be ported to any real-time operation system that supports this mechanism. Application processes providing control level services are regarded as servers, while service requesters are regarded as clients. According to these roles, servers block on the reception of specific queries and instructions. As soon as they unblock by message reception, data interpretation and calculation is performed and in general concluded with an acknowledging reply to the requesting client. All message-based communication is performed and controlled by the key component of MiRPA-X, the object server. It is an independent process within the system that evaluates inquiries issued by clients and forwards them to the respective servers without data interpretation. The forwarding strategy is based on a name service specifying application services (e.g. MotionModule_CalcAlgo). Servers are thus identified using a representation for the service name based on a hash mechanism. This kind of name service facilitates a transparent and flexible way of implementing modular and even redundant application services without reference to the
actual server and server location providing the service.

2) Shared Memory and Synchronization: Alongside the message-based communication, MiRPA-X provides a communication mechanism based on shared memory usage. As for messages, the shared memory mechanism uses the MiRPA-X name service. The usage of shared memory facilitates a high-speed, non-blocking data transfer between application processes, without the object server being involved in the actual communication task. After registration, application processes directly read and write fast inter-process data using memory pointers provided by MiRPA-X. For each shared memory region they act as publishers and subscribers, respectively. In the case of multiple application processes accessing the same shared memory region, especially for complex control systems, data integrity has to be ensured at any time. As shared memory access is neither blocking nor synchronized by itself, MiRPA-X provides mutex and condition variable mechanisms using the same name service as for messages and shared memory.

C. Self-Management at the Present State of Development

At startup, the control core broadcasts a profile request command to all motion modules and they deliver their names and specific capabilities back into a prepared synchronized shared memory area. The most important properties of the motion modules are whether they are able to operate in a hybrid control context and if they are suitable to be used in the task frame formalism. Using this information, the control core is able to decide whether the requested module combination specified by the current skill primitive is executable or not during runtime and which transformations have to be performed during execution. This ensures perfect scalability of the presented architecture: The programmer simply starts up the processes of the motion and sensor modules he wants to use within his program, no further effort is required. Furthermore, the maximum and minimum execution frequency of the modules is read from the modules’ profile. So, if at the desired cycle frequency the computational load has been too high in the current control cycle, motion modules can be rescheduled to a lower task execution frequency during runtime in order to prevent the system from algorithm overload.

III. ADVANCED SELF-MANAGEMENT

Even though our software architecture is very generic and modular [9], the self-management problem is gaining new degrees of freedom with the next expansion stage of our architecture, which is depicted in Fig. 5. Here several networked control PCs are used to distribute the algorithmic load in order to implement advanced capabilities such as integrating vision sensors. The figure also shows the next evolution step of the middleware MiRPA-X, where we have added a “D” to take account for its distributed operation. Since we have very rigid real-time constraints, for instance a change of the cycle frequency and the resulting necessary rearrangement of our hard real-time processes a deeper understanding of module inter-dependencies, expert knowledge about the communication system and extensive testing to see if the constraints still hold after the change are required.

Here, rearranging can mean distribution to remote processors and topology changes in terms of added or removed control PCs. Therefore, instead of manually optimizing the system effectiveness according to different criteria as e.g. performance or robustness, we want to integrate self-managers capable of performing this task. Those managers will capsule expert knowledge in evaluation functions and manipulation patterns and will be capable of judging and influencing the behavior of components. In addition to self-optimization, those management components will provide other so called self-* properties [18] like self-protection and self-healing. They follow the design suggestions of [5] and incorporate different threads for monitoring, analysis, complex planning and execution tasks, all using a common knowledge base (see Fig. 6).

By embedding the self-manager threads in different real-time layers we ensure that no real-time constraints are hurt by the integration. The only part of the management component working in the hard real-time context is the monitor. Even though fuzzy inference and case based reasoning techniques can be applied in deterministic time amounts, we are only integrating self-management in the soft and no real-time layers at the moment.

In the following, we describe the first realization of a self-manager for our system as depicted in Fig. 6. Its area of responsibility is the control core in the new distributed version of our architecture (see Fig. 5) in which control components can be spread over several control PCs. By splitting the algorithmic load capsuled in motion and sensor modules higher cycle frequencies and better algorithms become possible. The information which control part is running on which PC is stored in so called distribution patterns. The self-manager is able to seamlessly adapt the control system to topology changes, can find and verify beneficial distribution

![Fig. 5. Next expansion state of our software architecture PROSA-X. The algorithmic load is distributed between several control PCs.](image-url)
patterns, and trigger the migration of control components to remote processors. Its integration realizes self-* properties for this system part that are detailed in the following.

A. Self-Optimization

After system start, the knowledge base of the self-manager already contains a few distribution patterns. Those patterns are known to result in a working system and include, of course, the starting pattern. There are also patterns in the knowledge base that already got verified in prior runs, failed to pass the verification, or caused errors before. We are working on a pattern verification realized with the schedulability analysis tool SymTA/S [3] that is running on a separate analysis PC. Resources such as busses or processors, tasks assigned to them, task dependencies, priorities and deadlines are used for a formal verification of the real-time constraints. The verification is realized with different algorithms delivering worst case response times for isochronous and asynchronous IEEE1394-based communication. Those algorithms are an extension of a priority based principle by Lehoczky [8].

The verification process can also be executed during the runtime of the system. The TCP communication with the analysis PC is demand driven and has a small time consumption. A SymTA/S model on the main control PC constantly gets refined with measured execution times, topology information or changing task dependencies. When the self-manager searches for ways to optimize the behavior of the control core he also evaluates if the component could perform better after a rearrangement of its different tasks.

This consideration is triggered by its constant striving for self-optimization. According to its aim specification and evaluation function, an optimization could be more performance (e.g. a higher cycle frequency) or more robustness (e.g. redundant execution, freeing execution time in the cycle). It also could be triggered by a topology change where, for instance, additional control PCs got connected. The self-manager searches its knowledge base for an appropriate distribution pattern and if there is none, one to be evaluated is transferred to the SymTA/S instance running on the analysis PC. After verifying if real-time constraints still hold after a rearrangement of tasks or not, the distribution pattern gets transferred back to the self-manager and becomes part of its knowledge base. Its executor component then can acquire necessary resources, reconfigure the communication system and trigger the migration or remote startup of tasks via MiRPA-XD.

B. Self-Protection

The monitoring thread of the self-manager constantly records all kinds of information relevant for the evaluation of the control core’s behavior. By keeping track of execution and response times, the analyzer thread is capable to recognize when the system is approaching a critical system state. Several critical settings are thinkable.

The motion and sensor modules of RCA562, bearing most of the algorithmic load, reside within the soft real-time layer of the architecture. Violations of real-time requirements, meaning late responses of a module in most occurrences can be tackled with fallback solutions. Nevertheless, this generally signalizes a bad system performance. By measuring the unused cycle time at the end of the outer control cycle which serves as a buffer catching jitter caused by fluctuating calculation or transfer times, the self-manager’s analyzer can recognize that the system is approaching a performance problem and search for a solution within its planner before it occurs.

Another parameter capable to cause problems is the bus load and the response times of remote modules that depend on this load. Such bad response times can be caused by a prior migration of a calculation module that resulted in an heavily increased bus load because of strong dependencies between the migrated and a local module. An appropriate reaction of the self-manager after noticing a critical bus load would be to reverse the previous migration or generally a concentration of modules on less PCs.

Self-protection within our system does not mean protecting the system against malicious external influences but to anticipate error-prone system states and to initiate countermeasures before an error actually occurs.

C. Self-Healing

Even with self-protection capabilities in place, errors can occur that could not be foreseen and have a critical impact on the system. In the hard real-time layer a violation of real-time requirements inevitably leads to an error where the robot is stopped in a controlled way and a global error state is entered. This also can happen when a real-time constraint in
the soft real-time layer is hurt and no fallback solution exists that can compensate the missing response of a module.

From within the error state the self-manager can analyze the situation and try to return to normal operation by a rearrangement of modules within the distributed architecture and an corresponding reconfiguration of the communication system. Therefore, it can use already verified distribution patterns from within its knowledge base or trigger a new verification if no patterns are available corresponding to the actual or planned situation. Since the system is in a safe and stable state after an error occurred, time consumption of the formal analysis does not constrain this approach.

By the integration of self-* properties via self-managers, a very flexible control core is realized that is able to seamlessly adapt to environmental changes, to anticipate and prevent errors, and to reconfigure itself after errors to return to normal operation. The integration of a formal analysis verifies the adherence to real-time requirements before a rearrangement of components within the distributed architecture is conducted. Another self-management capability already inherent in our system is the self-configuration that takes place in different parts of the system after startup as previously described in section II-C.

IV. CONCLUSION AND FUTURE WORK

In this paper design principles for building a modular and self-managing control architecture for handling and assembly tasks have been presented. An example architecture consisting of a significant portion of reusable components already taking benefits from these principles could be successfully applied to two different parallel kinematic robots.

In the near future, distributed execution of sensor and motion modules will become available by developing a distributed middleware called MIRPA-XD based on the IEEE 1394 standard as well as on Ethernet transceivers for PC interconnection. This allows for the implementation of more demanding motion and sensor modules in terms of computation time. Furthermore, it is planned to extend the existing self-management capabilities and realize self-optimization mechanisms during runtime. A distributed architecture will be able to plan and optimize the scheduling with respect to the modules’ profile information and the measurement of their real behavior during execution autonomously.

The next challenge is to find the relevant process information for robot control components, required for the self-manager’s knowledge base and to capsule it in analysis and planning algorithms.

The main aim for the future is to enable the control designer to put less effort in architecture and component design, allowing him to concentrate on the control algorithms and assembly strategies themselves.

V. ACKNOWLEDGEMENTS

The authors are thankful for the given and still granted support on Collaborative Research Center 562 by the German Research Foundation (DFG). Acknowledgements are given to QNX Software Systems Inc. for providing free software licenses.

REFERENCES


