

Runtime Analysis and Adaptation of a Hard Real-Time Robotic Control System

Jens Steiner, Matthias Hagner and Ursula Goltz
 Institute for Programming and Reactive Systems
 Technical University of Braunschweig, Germany
 Email: {steiner, hagner, goltz}@ips.cs.tu-bs.de

Abstract—The increasing complexity of technical systems often leads to problems when they are to be maintained, changed or extended. Nature inspired concepts like self-management or self-organization have found their way into technical systems to overcome the complexity problems. In turn those systems exhibit beneficial self-* properties like self-optimization, self-healing or self-protection.

This paper presents a software architecture for the control of parallel kinematic machines and its evolvement to a self-adaptive system that strives to optimize, protect and heal itself. A software engineering approach for the development of self-managing components is introduced that is supported by behavior validation in a specialized simulation environment. The first realization of a self-manager, responsible for the distribution of control components, is described in detail to show that self-management is feasible in robotic control. The self-manager uses formal analysis techniques during the runtime of the system to make sure it always conforms to its real-time requirements.

Index Terms— Self-Management, Autonomic Computing, Self-* Properties, Robot Control Architecture, FireWire, Schedulability Analysis

I. INTRODUCTION

Complexity and its handling has become a major issue over the last few years. The increasing size and connectivity of distributed and possibly heterogeneous systems has led to difficulties in maintainability and extendibility. Intricate errors in technical systems leading to unpredictable and in cases catastrophic behavior are still a common issue, even though huge quality assurance efforts are undertaken.

Several proposals have been made to handle the ever increasing system complexity and the according problems. An important class of them tries to handle the complexity with nature inspired concepts. IBM suggested an autonomic approach in the year 2001, later refined in [1], and Organic Computing has established as new research field [2]. By creating technical systems

consisting of multitudes of self-managed components that adapt their behavior according to environmental changes and aim specifications it is possible to achieve such advantageous properties as self-healing, self-optimization, self-protection or self-configuration, also summarized as self-* properties [3]. As a result one does not only gain control of the complexity but can create very robust and fault-tolerant systems that can adapt to changing conditions or even protect themselves from environmental hazards.

Often, Autonomic Computing systems derive their macroscopic behavior from local observation and communication of their agent-like subsystems.

Due to this and the ever possible changing of component aims or behavior, it is very difficult to formally verify properties for autonomic computing systems. Some model-based approaches use assume/guarantee or compositional verification [4], some use simulation techniques [5].

It is still unclear what kind of engineering approach is the best for autonomic or organic systems even if advances have been made in that area that transfer some of the classic system engineering approaches to this new domain [6, 7, 8].

This paper introduces self-management components in a software architecture for robotic control that allow the realization of self-* properties for its components. It further shows how functional and real-time correctness can be ensured by the application of formal and lightweight formal methods during the design and later during the runtime of the system.

In Section II, our software architecture, its components and its real-time layers are described. Section III focuses on the creation, validation and integration of the management components. A self-management case study, where the adherence to real-time requirements is ensured with a runtime schedulability analysis, is presented in Section IV. Section V concludes.

II. SOFTWARE ARCHITECTURE

The software architecture PROSA-X (Parallel ROBots Software Architecture - eXtended) [9], developed in our interdisciplinary research project, has been designed for the control of Parallel Kinematic Machines (PKMs). With their closed kinematic chains and low moved masses,

Based on "Runtime Analysis of a Self-Adaptive Hard Real-Time Robotic Control System", by J. Steiner, and M. Hagner which appeared in the Proceedings of 4th IEEE Workshop on Engineering of Autonomic and Autonomous Systems (EASe 2007), Tucson, USA, March 2007. © IEEE.

those robots come with beneficial attributes like high structural stiffness, outstanding precision and a high payload to robot mass ratio [10]. PKMs are able to move very fast. The high accelerations and velocities require high control cycle frequencies and induced several hard real-time constraints for our software architecture. The real-time capabilities were one of the cornerstones of our architecture, another one was generality. As a result PROSA-X is capable to control totally different robots without the need to swap behavioral parts.

The whole engineering process was interwoven with an extensive model-based approach that allowed several different quality assurance measures [11]. A deployment diagram visualizing the topology of the new distributed version of our architecture is shown in Fig. 1. Each control PC runs its own instance of the middleware *MiRPA-X* and the real-time bus protocol *IAP*. The robot task can be passed to the interpreter (*RoboPret*) via a TCP/IP interface which is also used to connect a dedicated analysis PC.

At the moment we are integrating image processing and singularity proximity algorithms to enhance the fault-tolerant execution of robot tasks. Since this induces demand for more algorithmic power, while the control cycle frequency has to be retained, we decided to integrate further control PCs. This approach does not only allow the distribution of the algorithmic load and thus even higher cycle frequencies but also paves the way for self-management capabilities to increase system performance and robustness.

One of the central components of our architecture, the control core (*RCA 562*), is now split and has parts running on several control PCs. Those distributed parts control and coordinate the execution of all cyclic algorithmic load. The according tasks, e.g. sensor pre-evaluation, trajectory planning, position and force control, are encapsulated in different motion and sensor modules that are promising candidates for a distributed control approach. Details about the control core in its single PC version can be found in [12].

Communication of the distributed system parts is realized by the same communication system we have used to connect sensors and actors until now.

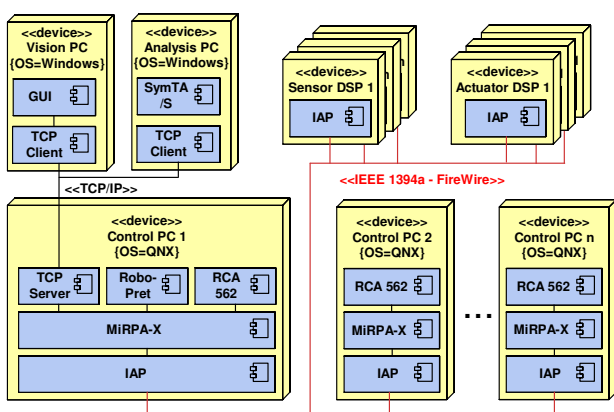


Figure 1. Distributed version of PROSA-X

Drive amplifiers or force sensors are connected via DSP devices that communicate over a FireWire field bus. Since there was no commercial communication system available that fulfilled our rigid bandwidth and performance requirements, we developed a new one in our research project. It is based on a standard component IEEE 1394 (FireWire) [13] bus solution, an according bus protocol (*IAP*) and a real-time middleware (*MiRPA-X*). An in depth description of the communication system can be found in [14].

On each of our control PCs, several real-time layers exist that safe-guard the validity of real-time constraints during execution. The hard real-time layer, for instance, contains a deterministic scheduler provided by the middleware, the FireWire bus protocol, and subordinate control and monitoring processes. Failure or time exceedance here leads to an emergency break and the entering of a global error state. The second layer is the soft real-time layer where non conformance to real-time limitations is noticed and can be treated by the system. Here for example, a skill primitive interpreter (*RoboPret* in Fig. 1), used to realize the error tolerant execution of robot tasks [15], resides. There is also a non real-time layer where demand-oriented system parts, like the TCP server, are working.

All tasks despite the ones in the hard real-time layer can be preempted by tasks with higher priority. The priority inheritance mechanisms of the target real-time operating system QNX are used to avoid priority inversion problems. Hard real-time tasks are synchronized to a hardware timing signal and leave some time at the end of the cycle. That timeslot is used by the soft and non real-time tasks that usually take five of those timeslots to finish their periodic work.

One self-* property already inherent to our system is self-configuration. Several system parts find and offer resources over the middleware. The control core acquires and activates necessary motion modules for an actual task on-the-fly. The *IAP* configures its distributed protocol instances according to current communication needs. Bandwidth reservation, bus time slot coordination, and adaptation of data marshalling structures all happen automatically.

The new distributed version of PROSA-X, however, enables the realization of further self-* properties. For instance, self-optimization becomes possible, where performance oriented analysis and planning leads to the migration of control components and thus enables higher cycle frequencies. Self-protection is feasible, meaning monitoring the processor load and migrating calculation intensive parts before real-time constraints are actually violated.

The other way around, communication intensive parts can be concentrated on one PC to decrease bus load prior to a bandwidth related jitter. By rearranging system components after an error state is entered, for instance because the time slice for the soft real-time layer was too small, the system can heal itself and may return to normal operation.

The next section describes how we realize the self-management capabilities for our architecture by the preparation and integration of management components.

III. REALIZATION OF SELF-MANAGEMENT

In the first version of our architecture [9], multitudes of parameters in several components developed by different partners in our interdisciplinary research project influenced the behavior of the system. The amount of code lines containing instructions had already passed the 100.000 and experts from several groups were needed to tune the behavior of the system. We decided to introduce self-configuration in several system parts to reduce the efforts necessary to maintain the system, to configure it, and to integrate new hardware and software parts.

The control core *RCA 562* (see Fig. 1) knows of available motion modules because they publish a feature profile via the middleware when they are started. The control core constantly evaluates if it can handle the current robot task retrieved from the skill primitive interpreter (*RoboPret* in Fig. 1). It then activates only the necessary motion and sensor modules and awaits their results. Thus, it constantly reconfigures itself according to the current robot task [16].

The communication system used to connect sensors and actuators (*IAP* in Fig. 1) also exhibits self-configuration. The only information that has to be provided prior to the system start is the unique hardware ID of each node and the data dependencies between nodes. When the system is started, each node publishes its ID which is recognized by a master node that reserves all necessary bandwidth and channels. A bus reset (occurs for instance when a new device is connected) can cause ID changes. They are recognized and the bus system and all connected nodes get reconfigured automatically.

To increase the robustness, performance and flexibility of our system we decided to integrate further self-management capabilities. Instead of having several experts tune system parameters to achieve an adaptation to new circumstances, we wanted our system to realize adaptations and optimizations itself. Therefore subcomponents of PROSA-X should be provided with capabilities to observe and evaluate their own behavior as well as their environment. A common approach to realize such autonomic computing features via constant observation, evaluation and reaction, are specialized management components that get connected to components or system parts. IBM refers to such management parts as autonomic managers [1], others call similar components Operator Controller Module [17]. We use the term self-manager and basically follow the design suggestions of [1]. The structure of our self-managers is depicted in Fig. 2. A *Monitor* component records relevant parameter values and stores them in a knowledge base. An *Analyzer* uses the monitored values to execute evaluation functions whose results influence decisions of a *Planner*. An *Executor* realizes the planned adaptation. All this happens in a control loop that is interwoven with one or several real-time layers.

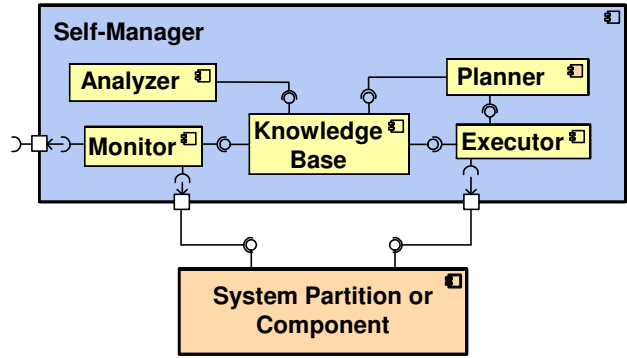


Figure 2. Self-Manager structure

Our object oriented design for PROSA-X self-managers is shown in Fig. 3. It shows a pattern that we use for the manual creation of self-managers. The only non abstract class is *SelfManager*. In its constructor a *SelfManagerPartFactory* object is passed. Its concrete factory class (example in Fig. 4) has to extend the abstract *SelfManagerPartFactory* and has to override its virtual creation functions. The concrete part factory creates concrete part objects. They are instances of classes that are derived from the abstract part classes (*Monitor*, *Analyzer*, *Planner*, *Executor*, *Data*). The design pattern that is used here is called *Abstract Factory* [18]. Its purpose is to specify an interface to a family of related or dependent objects without the need to specify their concrete classes. By using it, totally different self-managers can be realized without the need to change the *SelfManager* class. The interface to all its parts stays the same. The *SelfManager* object calls the creation functions of the factory object it got passed and triggers the behavior in its parts by calling methods defined in the abstract part classes.

Another design pattern that is used here is the *Observer* pattern [18]. The data objects that get created by a *SelfManagerPartFactory* provide functionality for observers to become registered. The observers in our case are the concrete part objects of the self-manager. Upon a change of the data encapsulated in a data object, it notifies all registered observers. The abstract class *Data* also contains virtual lock and unlock methods that can be overridden to ensure data integrity in multi threaded environments.

Fig. 4 shows a concrete *SelfManagerPartFactory* that creates the parts and data objects for a self-manager that is described in detail in Section IV. Several of its parts are derived *Observer* classes that get notified whenever the data object they depend on is changed.

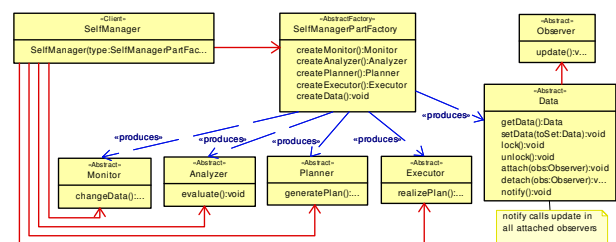


Figure 3. Abstract design for self-manager components

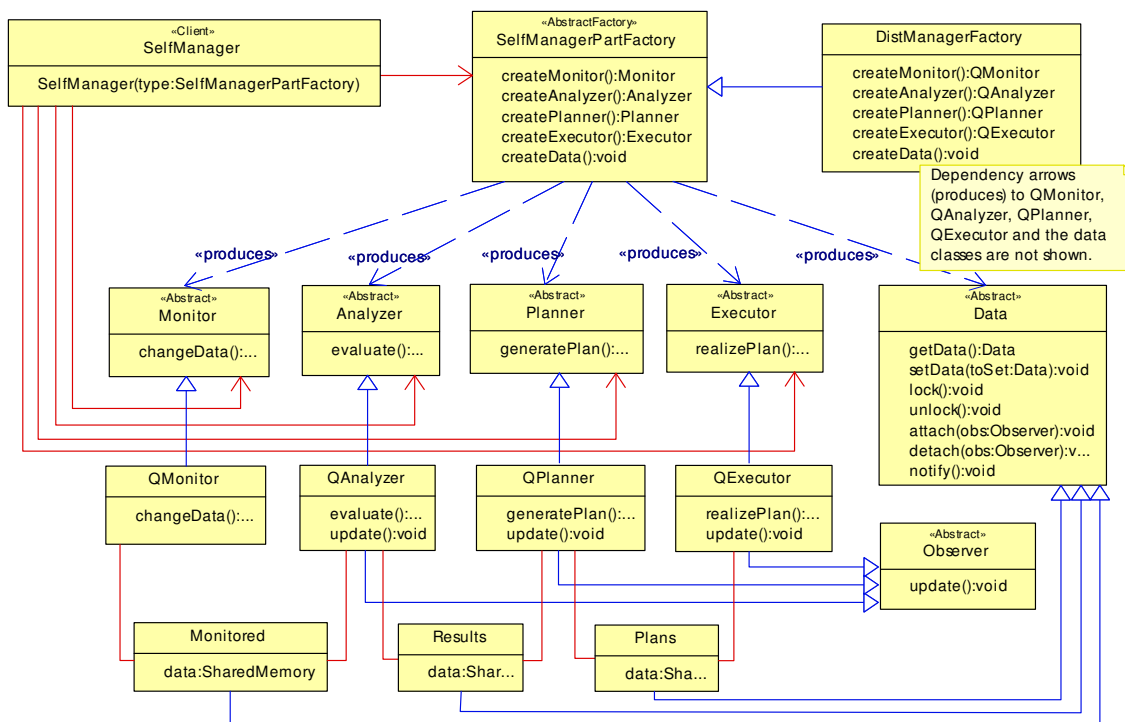


Figure 4. Concrete *SelfManagerPartFactory* creating the parts for the self-manager introduced in Section 4

During the development of PROSA-X, behavioral models for nearly all its core components have been created. Based on these models several quality assurance measures could be applied to validate and verify the behavior of the system prior to an implementation [11]. We intend to use these approved model-based techniques to ensure that our system still fulfills its requirements after the integration of self-managers that adapt the system according to changing external stimuli. We have developed a simulation environment that mimics most of the key features of our target real-time operating system QNX. It serves as a virtual operating system that can be connected to the CASE¹ tool Rhapsody² in which UML state diagrams can be executed and visualized in a simulation mode. The executed behavioral models behave like threads that are run on a QNX PC. This way complex thread interactions, synchronization and communication can be examined and validated based on models. To make this approach feasible for a validation of a PROSA-X with self-management capabilities, behavioral models (state diagrams) have to be created that describe the behavior of the self-managers to be integrated in the system. Those behavioral models have to contain the expert knowledge that is necessary to evaluate the behavior of components and to react accordingly. At the moment those models have to be created manually by extending the part classes as seen in Fig. 4 and by describing their behavior as state machines. We are working on a tool to automate the creation of behavioral models out of specified expert knowledge.

The idea is that component designers specify resources relevant for the evaluation of the component and for decisions about behavioral changes. This can be execution times, shared memory areas or externally visible component attributes. Pointers to that information will later be part of the knowledge base of the according self-manager. Afterwards, the expert specifies an evaluation function providing one or more index values revealing how the component currently performs. This evaluation can also trigger complex analysis or planning procedures as later will be shown in Section IV. A planning algorithm has to be specified that, depending on the evaluation results, manipulates the system parts in the according area of responsibility. The model generation tool will create derived part classes of the self-manager parts (see Fig. 4). As done manually now, the tool will override the virtual interface functions and determine an appropriate real-time layer and priorities. While things like case based reasoning or fuzzy inference can be performed in a hard real-time context, more complex tasks have to be realized in the soft or non real-time layers.

Out of the behavioral self-manager models we already generate QNX compliant source code with Rhapsody that can be compiled and executed in our simulation environment [11], together with behavioral models of our architecture. The simulation can be controlled manually or by scenario based test cases capturing system requirements. This way, a first validation of the system behavior with integrated self-management components is achieved. The simulation approach can determine if there are racing conditions for shared resources or if functional and real-time requirements are met.

¹ CASE – Computer Aided Software Engineering

² Rhapsody is developed by Telelogic (www.telelogic.com)

In addition to the lightweight validation prior to a system execution, we want some of the self-managers to base their decisions on formal analysis. If the analysis of a new situation is performed prior to an actual reaction of the self-manager, this reaction may be a bit slow. For critical system parts where optimization is not existential but a goal subordinate to safety, meaning functional and real-time correctness, we consider this feasible. The formal analysis can take part on a separate PC, whereas the result evaluation happens in a soft real-time layer on a control PC. An example for this approach is described in the following section.

IV. RUNTIME ADAPTATION

This section presents the first realization of a self-manager for our software architecture. Its purpose is to analyze, plan and execute the distribution of control components, especially the motion and sensor modules mentioned in Section II. We wanted the new distributed control system to adapt itself to topology changes automatically. Newly connected control PCs should be integrated seamlessly and become available for component migration. Broken down or disconnected PCs should not inevitably lead to fatal system errors.

By the integration of this self-manager we wanted to create a control system capable to rearrange itself according to changing external conditions. Depending on a goal specification the system should be capable to enhance its robustness, for instance by redundancy. It should also be capable to increase its performance, e.g. achieve a higher cycle frequency, by rearranging real-time layers and component distribution. In addition, we wanted the distribution of system components and adaptation to topology changes to be fully automatic.

The self-manager we have created to realize all that (see Fig. 5) has to consider many parameters, e.g. execution times, available bandwidth and current or predicted topology. It has to decide which components to place on which control PC and, most of all, it has to determine if the system is able to run without hurting its real-time constraints. Therefore, we have created a connection to the schedulability analysis tool SymTA/S.

A. SymTA/S Introduction

SymTA/S (**S**ymbolic **T**iming **A**nalysis for **S**ystems [19]) is a scheduling analysis tool developed by Symtvision³. We have used it in the past for feasibility analysis and the model-based verification of real-time properties. While our prior usage was an offline one, we needed to create a way for an online integration of the schedulability analysis to make it usable by our self-manager at runtime. To make sure an assignment of control components to control PCs, we refer to it as distribution pattern, does lead to a stable situation where all real-time requirements are met, the self-manager verifies distribution patterns with SymTA/S before actually realizing them and migrating any components.

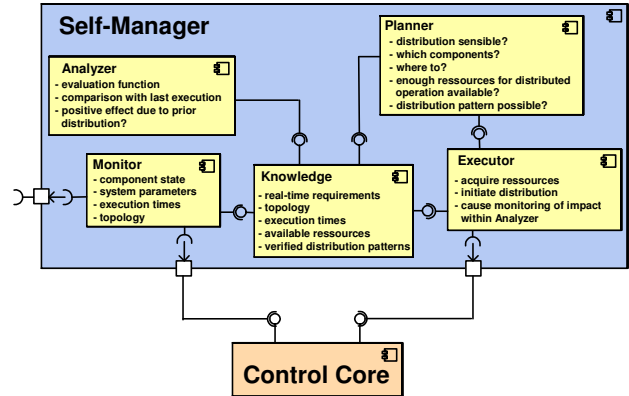


Figure 5. Self-Manager for the control core

A schedulability analysis by SymTA/S requires a configuration description of the system to be analyzed containing all relevant parameters. Such system configuration has to be provided as a model for SymTA/S and contains resources, tasks, channels and event streams (see Fig. 6). A resource can be a processor or a bus. Tasks are assigned to processors while channels are assigned to buses. Functional dependencies are modeled by event streams.

The properties of all model elements can be set individually. Tasks are described by properties like their core execution time and their priority. Properties of a processor are, for instance, the scheduling algorithm or the time required for a context switch. A typical bus property is the used scheduling algorithm, and channels have properties like the packet size.

The schedulability analysis checks whether the configuration of the system is schedulable or not and therefore has to calculate the worst case and best case response times of all tasks. SymTA/S analyzes the utilization of all resources and examines whether all tasks are executed or may be prevented from execution by higher priority tasks. If tasks have deadlines, the analysis also examines whether those deadlines are met. SymTA/S calculates and observes possible execution paths and checks if a task chain is executed sufficiently fast, if there are user defined bounds for the chain. The elements in a chain are connected by event streams and are called a path.

SymTA/S uses known and established analysis algorithms and connects them over event stream to analyze complex and distributed systems. At first every resource is analyzed individually and the worst and the best case response times are examined for all tasks that are scheduled on the resource. Through these values and the given *Input Event Model* the *Output Event Model* is calculated and propagated over event streams. If there are cyclic dependencies, the system is analyzed from a starting point iteratively until the results converge.

B. Analysis Integration

The whole configuration is stored in an XML file. Which paths are to be observed and what constraints are to be kept on them can also be defined in that file. For the execution of SymTA/S, we connect an additional PC to

³ www.symtvision.com

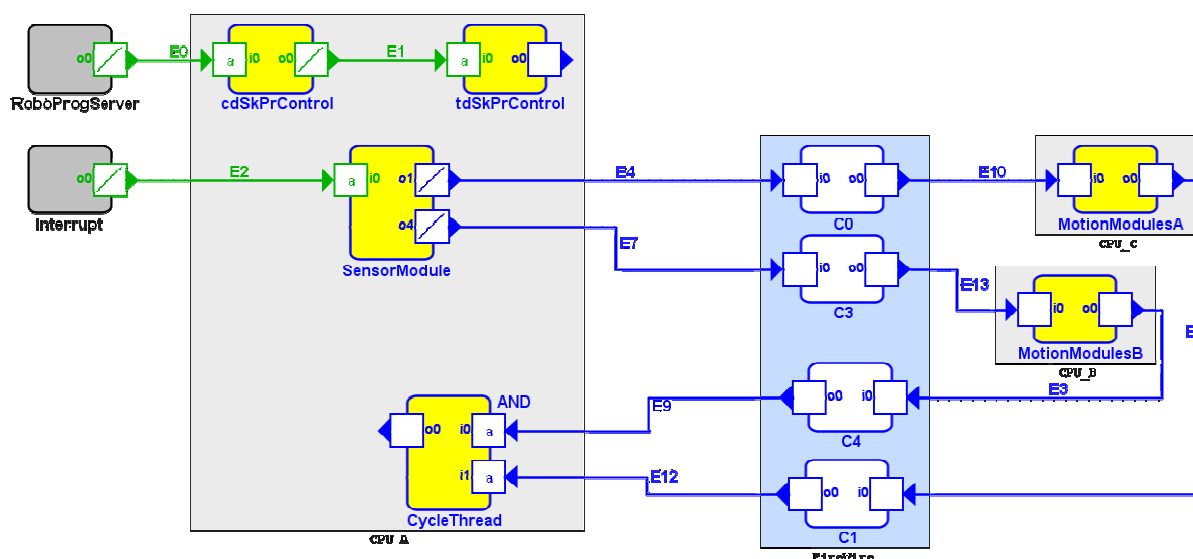


Figure 6. Example model representation in SymTA/S

our system (see Fig. 1). The self-manager of the control core reads the system configuration out of the XML file when the system starts and stores it in its knowledge base. When the self-manager decides to evaluate a possible distribution, it manipulates the XML representation of the configuration in memory, transfers it to the analysis PC via a TCP connection and thus triggers the schedulability analysis. A wrapper tool starts SymTA/S using its TUI (Textual User Interface). The wrapper tool also transfers the analysis results back to the control PC. There they become part of the knowledge base of the self-manager and it can found decisions about component migration on it.

The accuracy of the analysis can be improved online as measured execution times become available. By monitoring, for instance, time consumption of the FireWire driver for isochronous or asynchronous data transfers, the self-manager can improve the model of the system configuration before it triggers the analysis by SymTA/S. Those time amounts depend on the node ID assigned during the FireWire root contention protocol and can change for instance when a new component is connected to the bus. The online surveillance of that parameters and a constant refinement of the SymTA/S model improve the accuracy of the schedulability analysis.

C. Scheduling Analysis of FireWire Communication

FireWire [13] is a serial bus interface standard, offering high-speed communications. The FireWire bus offers two different transfer modes for different requirements. One mode is for the transmission of data with real-time constrains (isochronous); another mode is for safe transmission (asynchronous).

To make an analysis of our system with SymTA/S possible, we had to create a plug-in for SymTA/S that analyses FireWire communication. Since the analysis approach of SymTA/S is based on worst and best case response times, we had to create algorithms that deliver these values. Using an extension of a busy window

technique by Lehoczky [20], we created different algorithms for isochronous and asynchronous FireWire communication. In the following we will describe the analysis of either isochronous or asynchronous communication. The analysis of a mixed communication is also possible but goes beyond the scope of this paper.

In the IEEE 1394 specification, time is divided into fixed-sized cycles. A cycle (Fig. 7) has a duration of 125 microseconds. Every 125 microseconds, a root node sends a cycle start package to signalize the start of a new cycle. The root node is an "elected" node with several management responsibilities. Fig. 7 also shows different gaps between packets. There is the isochronous gap, which occurs between the cycle start packet and an isochronous packet or between different isochronous packets. In this isochronous gap, nodes may register for an isochronous communication. If there is no such registration request within an isochronous gap, it becomes a subaction gap. Thus, the subaction gap is one that appears after the last isochronous packet or between asynchronous packets. In it nodes may transfer asynchronous data. Finally, there is the arbitration reset gap. This gap occurs after a fairness interval ends (Fig. 9) and is explained a few subsections later.

FireWire guarantees the periodic data transmission in isochronous transfer mode. Thus, this mode is well suitable for real-time applications. A node that wants to transfer data in this mode first has to request the necessary bandwidth. If there is still enough free bandwidth available, the *Isochronous Resource Manager* allocates it for the node. The reserved bandwidth is then available for the node in every cycle. It is possible to allocate 80 percent of the entire bandwidth for isochronous communication. The rest of the cycle can be used for asynchronous communication.

1) *Analyzing Isochronous Communication:* The algorithm for the worst case response time for isochronous communication is divided into three parts (Fig. 8):

- find out the critical instance

- calculate the number of necessary cycles
- analyze the last cycle.

Liu and Layland [21] defined the critical instance of a task as a situation, in which the task has the largest response time. Lehoczky [20] proved that the critical instance of a task occurs at the beginning of the *level-i busy period*. This *level-i busy period* is defined as the maximum time for which a processor executes tasks with priority higher than or equal to the task priority of the concerning task.

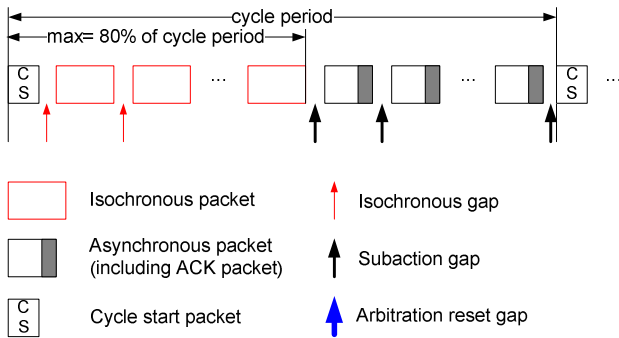


Figure 7. FireWire communication cycle

As critical instance for isochronous communication we have identified the following situation:

1. The communication is activated directly after the first subaction gap in a cycle occurs.
2. There has been no other isochronous communication in that cycle
3. In the last cycle necessary to transfer all data the according task is scheduled last.

Due to the first two points, nearly a whole cycle passes before anything can be sent isochronously again (Fig. 8).

The number n of necessary cycles can be calculated out of the total amount of data to be transferred and the maximum packet size, which are both known. From the number of cycles, the amount of necessary transmission time can be derived easily by multiplication with the cycle time.

The last necessary cycle has to be examined closely, though. It may be the case that the examined task alone is able to communicate here and thus transmission takes place directly after the cycle start telegram. The analysis algorithm checks if there are tasks in the last cycle (like Y in Fig. 8) that could send before the examined task X and may postpone it. Since we want to calculate the worst case, X is scheduled last. Even if this is the case, never the whole cycle time has to be added for the last cycle, since 20 per cent of it are reserved for asynchronous communication.

With the described critical instance situation plus the time calculated for the number of necessary cycles (under special consideration of the last cycle), the worst case

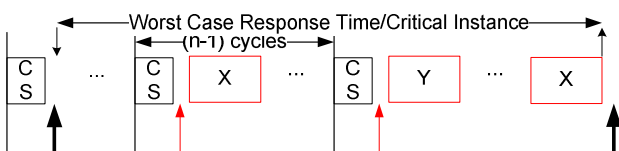


Figure 8. Critical instance for isochronous communication X

response time for isochronous FireWire communication can be computed. The complex formulaic description of the algorithm which also represents the underlying busy window technique [20] exceeds the scope of this paper and will not be shown here.

For the best case response time analysis, the critical instance is replaced by the least critical instance, namely the following situation:

1. The communication is activated directly after a cycle start packet and starts sending the first packet immediately.
2. In the last cycle, necessary to transfer all data, the according task is scheduled first.

2) *Analyzing Asynchronous Communication:* The analysis of the asynchronous mode is more difficult. A fairness interval, that exists to make sure every node can send an asynchronous packet before another node can do so a second time, has to be considered. It starts and ends with an arbitration gap. During a subaction gap, nodes may request an asynchronous transmission. Only one node can send at a time and gets a fairness mark afterwards preventing it from another request. During the following subaction gaps, other nodes may initiate the sending of asynchronous data. If a subaction gap passes without any node requesting an asynchronous transmission, the gap becomes an arbitration gap and all fairness marks are removed. In this moment, a new fairness interval begins.

Such fairness interval can be seen in Fig. 9. After node A has sent an asynchronous packet, it can not send again until an arbitration gap occurs signaling the end of the fairness interval. Until then, the other nodes (B, C and D) send asynchronous packets, but also only once in a fairness interval.

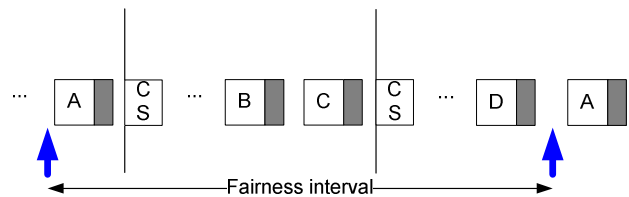


Figure 9. Fairness interval

Similar to the isochronous case, the number of necessary cycles to transfer all asynchronous data has to be calculated. In addition, the number and length of fairness intervals has to be determined. Unfortunately, the fairness intervals have no fixed length. Their length has to be calculated iteratively depending on the amount of nodes engaged in asynchronous communication, the amount of data they need to transfer and the according packet size. Out of all that, a number of necessary cycles can be calculated. That cycle number, however, defines how many cycle start telegrams are sent and thus how much the interval time is enlengthened. Due to this enlengthening, further tasks may become activated for asynchronous communication and further iterations may be needed to determine the length of the fairness interval.

Our iterative algorithm considers the critical instance for the asynchronous case, determines the number of fairness intervals and their individual length and calculates a worst case response time

As critical instance for a node interested in asynchronous communication we have identified the following case (depicted in Fig. 10):

1. The node has just finished sending asynchronous data. It can not send again in this fairness interval.
2. A cycle start telegram occurs immediately after the considered node has finished the sending as described in 1. Thus, the number of cycle start telegrams in the fairness interval is maximal.
3. All other nodes are activated and want to send asynchronously and none of them have done so in this fairness interval. After the fairness interval ends and a new one begins, all other nodes send asynchronously first. If further nodes get activated in the meantime, they send before the considered node, too. The considered node sends very last in this new fairness interval.

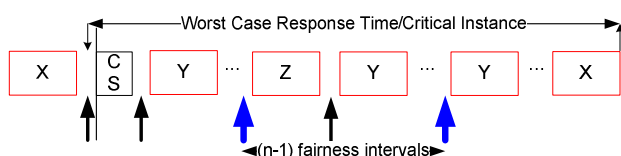


Figure 10. Critical Instance for Asynchronous Communication X

The best case response time algorithm is similar to the worst case one but starts with the least critical instance and thus always assumes the best case.

D. Self-* Properties

By the integration of the self-manager for the control core (Fig. 5) we could realize several self-* properties that go beyond the self-configuration introduced in Section III.

1) *Self-Optimization*: After system start, the knowledge base of the self-manager contains a SymTA/S model including all tasks of the control system, their interdependencies, worst case and best case execution times, resources and further information necessary for a schedulability analysis. The SymTA/S model also contains the current distribution pattern that describes which component is executed on which control PC.

This initial pattern represents a first configuration, known to be working. At the moment, patterns are not made persistent so there are no patterns available at system start that got verified in prior runs or already failed the verification by SymTA/S. A pattern gets a performance value after its verification representing resource utilization, latency of the longest path and redundancy information. By comparing the performance values the self-manager can find out if there are patterns available in its knowledge base that could perform better than the current one. The aim specification and the according evaluation function calculating the performance value define if the optimization goal is

performance (e.g. a higher cycle frequency) or robustness (e.g. redundant execution).

The parameters of the SymTA/S model are constantly monitored and the model gets refined with measured execution times and updated topology information. An optimization by a rearrangement of the different tasks can be triggered by a topology change, for instance when an additional control PCs gets connected. The planner component searches the database for a verified pattern reflecting the new topology situation. If it does not find an appropriate pattern, a different planner on the analysis PC is started that chooses a pattern candidate, tries to verify it with SymTA/S and transfers it back to the planner on the QNX PC. There, it becomes part of the knowledge base of the self-manager and can be used to adapt the control system to the new situation. Its executor component then can acquire necessary resources, reconfigure the communication system and trigger the migration or remote startup of tasks via *MIRPA-X*. The communication between the QNX and the Windows PC is realized with a TCP connection (see Fig. 1).

2) *Self-Protection*: 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. The monitoring component of the self-manager constantly records all kinds of information relevant for the evaluation of the control core and its behavior. By keeping track of execution and response times, the analyzer is capable to recognize when the system is approaching a critical system state. It knows several critical settings. The motion and sensor modules of *RCA 562*, 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 such a module can be handled here with fallback solutions but may also lead to error states. Generally, such late response signalizes a high resource utilization and 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 analyzer of the self-manager can recognize if the system is approaching a performance problem and search for a solution with its planner before an error occurs.

Another parameter capable to cause problems is the bus load. A bus load too high can delay the response of remote modules and in the worst case even cause a jitter of the cycle start telegram. A prior migration of a calculation module may have caused the high bus load because of strong data dependencies between the migrated and a local module. An appropriate reaction of the self-manager after noticing a critical utilization of a bus resource is to reverse the previous migration or more general concentrate control modules on less PCs.

3) *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 a corresponding reconfiguration of the communication system. It can use already verified distribution patterns from within its knowledge base or trigger a new verification if no pattern is available, corresponding to the actual 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.

The analyzer of our self-manager determines the current adaptation strategy. If the system is in an error state, the self-healing scenario becomes active. If there is no recognized error yet but the system approaches a critical state, the self-protection strategy is activated. And if there are neither current nor predicted errors, the self-manager tries to optimize the system behavior.

By the integration of self-* properties through a self-manager for the control core, a very flexible control system 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.

V. CONCLUSION AND OUTLOOK

We have presented a software architecture for parallel kinematic machines and our plans to integrate self-managers to encapsulate expert knowledge and create a self-adaptive system. We have introduced a software design using different design patterns for the realization of the self-managers and have briefly described how a model based simulation approach is used to validate the system behavior with integrated self-managers.

A self-manager responsible for the distribution of control components in the new distributed version of PROSA-X and the self-* properties realized by it has been explained in detail. It verifies the real-time requirements of the distributed system with the schedulability analysis tool SymTA/S prior to any rearrangement. The algorithms it uses to calculate the worst case response time of task chains containing FireWire communication have been explicated.

We have a single point of failure in the current version of our system. If the main control PC or software components running on it break down, the system will enter an error state from where it can not return to normal operation. We are working on strategies to recover at least from software failures on the central control PC.

We are also working on the tool support for a methodology to create self-managers for already existing systems, that encapsulate expert knowledge and realize self-* properties. The main focus of the methodology will be the validation and verification of properties of systems after the integration of management components.

At the moment, we have to create SymTA/S models of our system prior to an execution. Although those models get adapted later at runtime, the initial model creation task requires a lot of expert knowledge about the underlying bus architecture and the system components. There is research going on in the field of automatic model generation where all execution times, communication between components and other relevant system parameters are observed and then used to create SymTA/S models automatically during system execution [22]. This promising organic approach could greatly ease the online application of SymTA/S for schedulability analysis.

Depending on the amount of unused processor time during system execution, special online model checking algorithms as described in [23] can be applied to perform property verification during the runtime of the system. We are investigating if bounded model checking can be applied to management components in our system to verify safety properties online.

ACKNOWLEDGMENT

The authors would like to thank the DFG (German Research Foundation) for their financial support and QNX Software Systems as well as the Syntavision GmbH for the grant of free software licenses.

REFERENCES

- [1] IBM Corporation, "Autonomic Computing: IBM's Perspective on the Stand of Information Technology" available at http://www.research.ibm.com/autonomic/research/papers/AC_Vision_Computer_Jan_2003.pdf, 2003.
- [2] C. Mueller-Schloer, "Organic Computing – on the feasibility of controlled emergence", IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES + ISSS 2004), IEEE Computer Society, Washington, DC, USA, 2004, pp. 2-5.
- [3] T. De Wolf, and T. Holvoet, "A Taxonomy for Self-* Properties in Decentralised Autonomic Computing", *Autonomic Computing: Concepts, Infrastructure, and Applications*, 2006.
- [4] H. Giese, M. Tichy, S. Burmester, W. Schäfer, and S. Flake, "Towards the Compositional Verification of Real-Time UML Designs", ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering, ACM Press, Helsinki, Finland, 2003, pp. 38-47.
- [5] T. De Wolf, G. Samaey, and T. Holvoet, "Engineering Self-Organising Emergent Systems with Simulation-based Scientific Analysis", *Proceedings of the Third International Workshop on Engineering Self-Organising Applications*, Universiteit Utrecht, Utrecht, The Netherlands, 2005, pp. 146-160.

- [6] T. De Wolf, and T. Holvoet, "Towards a Methodology for Engineering Self-Organising Emergent Systems", Self-Organization and Autonomic Informatics (I), Frontiers in Artificial Intelligence and Applications, Volume 135 of Frontiers in Artificial Intelligence and Applications, IOS Press, Glasgow, Scotland, UK, 2005, pp. 18 – 34.
- [7] J. Gausemeier, U. Frank, H. Giese, F. Klein, A. Schmidt, D. Steffen, and M. Tichy, "A Design Methodology for Self-Optimizing Systems", Proceedings of AAET 2005 - Automation, Assistance and Embedded Real Time Platforms for Transportation - Airplanes, Vehicles, Trains, Technical University of Braunschweig, Braunschweig, Germany, 2005.
- [8] H. Kasinger, and B. Bauer, "Towards a Model-Driven Software Engineering Methodology for Organic Computing Systems", Proceedings of the 4th IASTED International Conference on Computational Intelligence (CI 2005), IASTED/ACTA Press, Calgary, Alberta, Canada, 2005, pp. 141-146.
- [9] N. Kohn, M. Kolbus, T. Reisinger, K. Diethers, J. Steiner, and U. Thomas, "PROSA - A Generic Control Architecture for Parallel Robots", Proceedings of Mechatronics & Robotics, Sascha Eysoldt Verlag, Aachen, 2004, pp. 56-61.
- [10] J.-P. Merlet, Parallel Robots, Kluwer Academic Publishers, Dordrecht, The Netherlands, 2000.
- [11] J. Steiner, M. Huhn, and T. Mücke, "Model based quality assurance and self-management within a software architecture for parallel kinematic machines", Proceedings of the IEEE 3rd International Conference on Mechatronics, ICM2006, Budapest, Hungary, 2006, pp. 55-60.
- [12] J. Maaß, N. Kohn, and J. Hesselbach, "Open Modular Robot Control Architecture for Assembly Using the Task Frame Formalism", International Journal of Advanced Robotic Systems, Volume 3, Number 1, ISSN 1729-8806, 2006, pp. 1-10.
- [13] D. Anderson, *FireWire System Architecture: IEEE 1394A (2nd ed)*. Addison-Wesley Longman Publishing Co., Inc., 1999
- [14] N. Kohn, and J. Steiner, "Universal Communication Architecture for high-dynamic Robot Systems using QNX" Proceedings of International Conference on Control, Automation, Robotics and Vision (ICARCV 2004), Kunming, China, 2004.
- [15] U. Thomas, W. An, and F.M. Wahl, "Sensor guided execution of robot tasks based on skill primitives", *Robotics* 2002, 2002, pp. 71-77.
- [16] J. Maaß, J. Hesselbach, J. Steiner, and U. Goltz, "Self-Management in a Robot Control Architecture", Proceedings of Second International Workshop on Software Development and Integration in Robotics (SDIR 2007), affiliated with ICRA 2007, 2007.
- [17] O. Oberschelp, and T. Hestermeyer, and B. Kleinjohann, and L. Kleinjohann, "Design of self-optimizing agent-based controllers", Proceedings of 3rd International Workshop on Agent Based Simulation, Passau, Germany, 2002.
- [18] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.
- [19] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, "System Level Performance Analysis – the SymTA/S Approach", IEEE Proceedings Computers and Digital Techniques, Volume 152, Issue 2, 2005, pp. 148-166.
- [20] J. Lehoczky, "Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines", Proceedings of the IEEE Real-Time Systems Symposium, 1990, pp. 201-209.
- [21] C. L. Liu, and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", Journal of the ACM, vol. 20, ACM Press, 1973, pp. 46-61.
- [22] S. Stein, A. Hamann, and R. Ernst. "Real-time Property Verification in Organic Computing Systems", Proceedings of the 2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, Paphos, Cyprus, 2006.
- [23] Y. Zhao, S. Oberthür, M. Kardos, and F.-J. Rammig, "Model-based Runtime Verification Framework for Self-optimizing Systems", Proceedings of the Fifth Workshop on Runtime Verification, Edinburgh, Scotland, UK, 2006, pp. 125-145.

Author Jens Steiner was born in Wittenberg in the year 1978. He studied Computer Science at the Technical University of Braunschweig from 1998 till 2003 when he received his diploma degree in computer science.

Since then he has been a research assistant at the TU Braunschweig, working in the Institute for Programming and Reactive Systems. His main research interests are Autonomic Computing and model-based analysis.

Author Matthias Hagner was born in Blankenburg, Germany in November, 1981. He studied Computer and Communication Systems Engineering at the Technical University of Braunschweig and received his Diploma in September, 2006.

During his studies he worked for NetCo Professional Services GmbH as an engineer. Currently he is a research assistant at the Technical University of Braunschweig, working in the Institute for Programming and Reactive Systems.

Author Ursula Goltz was born in the year 1954. She studied Computer Science at the Technical University of Aachen and graduated there with a diploma degree 1982. She received their Ph-D degree from the TU Aachen in the year 1988.

She worked as a scientific assistant at the TU Aachen from 1982-1985 and at the Institute on Methodological Foundations of GMD, St. Augustin from 1986-1992. After teaching activities at the Universities of Munich, Erlangen-Nürnberg, Mannheim and Bonn, she became professor for Programming at the University of Hildesheim in the year 1992. Since 1998 she is professor for Computer Science at the Technical University of Braunschweig. There she is chair of the Institute for Programming and Reactive Systems. Her main research interests are specification and system design, reactive systems, concurrency, process algebras and semantics.