A Real-Time Library for the Design of Hybrid Robot Control Architectures *

Giuseppe Beccari Stefano Caselli Monica Reggiani Francesco Zanichelli
Dipartimento di Ingegneria dell'Informazione
Università di Parma - 43100 Parma - Italy
e-mail: {beccari,caselli,reggiani,mczane}@ce.unipr.it

Abstract

The paper describes a real-time library providing facilities useful in the design of robot control architectures. The library supports structured creation of reactive and deliberative modules, dynamic modification of relevant real-time parameters, generation of timing fault handlers, measurement and monitoring of execution times. This support enables adaptation of the rate of computation of real-time modules to the rate of change of the external world, and hence better tuning of robot behavior to the world uncertainty and dynamics. The real-time library has been put at work by designing a hybrid control architecture for a robot performing a kitting task. In the prototype experiment described in the paper, a Puma 560 robot manipulator is fed with parts by a small mobile robot. The control architecture governing Puma operations dynamically allocates computational resources to reactive and deliberative modules, according to the task level priorities.

1 Introduction

The design of control architectures for autonomous robots, operating in dynamic and possibly unstructured environments, poses significant challenges to real-time computing. Adaptability and autonomous operation, in fact, mandate richer perceptual, motor, and reasoning capabilities, and are being pursued by means of architectural approaches emphasizing modularity, concurrency, and extensibility. In these architectures, predictable and smooth behavior still requires compliance to real-time constraints. However, these constraints cannot be enforced with the design techniques standard in industrial robots, largely based on resource underutilization and a limited set of guaranteed computations (often a single real-time loop).

Moreover, as autonomous robots are intended to operate also in dynamic environments, they should adapt their behavior to novel or changing operating conditions and take into account multiple, possibly interfering goals. The AI field encompasses a number of planning, reasoning, and learning techniques that may help to address the complexity and variability of the environment. These techniques are often inherently time consuming and unpredictable in the time they take to reach a solution (e.g., techniques based on state space search). Nevertheless, the control architecture of an autonomous robot should possess both the safety of guaranteed real-time computing and the flexibility and adaptability of AI techniques, where needed [10].

This paper focuses on the description of a *real-time library* aiming to support the design of the reactive layer for hybrid robot control architectures. The library offers facilities for timing fault handlers, time monitoring, periodic or aperiodic execution, and scheduling guarantees. These facilities may be available in part in a generic real-time operating system, or could be obtained from more primitive mechanisms. However, the library hides to the designer some of the complexity by providing a simplified interface. Furthermore, the real-time library allows on-line modification of real-time parameters. This feature can be used to adapt the computational requirements of the real-time subsystem at run time.

The real-time library has been built on top of Solaris 2.6, a general-purpose, low cost operating system of the UNIX family in which some support for real-time computation coexists with the standard time-sharing mechanisms. General-purpose operating systems are expected to provide increasing support to soft real-time applications. The multi-threading computing environment of Solaris (and other modern general-purpose operating systems) yields a flexible support for implementation of concurrent control architectures while retaining the ability to easily integrate the useful AI tools available in the UNIX environment.

The paper is organized as follows. The next section briefly reviews several approaches to robot control architectures and how they deal with real-time issues. Section 3 describes the real-time library which represents the main contribution of the paper. Section 4 describes a robot kitting task set up to evaluate the effectiveness of the proposed real-time techniques. The task involves a Puma 560 robot manipulator cooperating with a mobile robot. Section 5 outlines the control architecture we have developed for the kitting task based upon the library. Section 6 reports some experimental results and Section 7 presents the conclusions and projected developments.

2 Hybrid architectures and real-time

A number of structuring approaches are being pursued in the design of control architectures for autonomous robots, from purely reactive architectures to deliberative systems. Among them, *hybrid control architectures* attempt to incorporate both reactive and delib-

*Partial support for this research has been provided by MURST, Italy (Project "Design Methodologies and Tools of High Performance Systems for Distributed Applications"), by C.N.R., Italy (Special Project "Real-Time Computing in the Real-World"), and by ASI, Italy (Project No. ARS-96-182 "Dextrous Manipulation for Space Robot Tasks").

erative functionalities in a multi-layered structure. The purpose of this section is to discuss how hybrid architectures described in the literature deal with real-time issues. This review will help us to identify the current and perspective real-time requirements of these architectures. However, in many architectural proposals there is no explicit consideration of the real-time issues. The underlying system is simply expected to be able to execute all tasks fast enough.

2.1 Literature review

Several approaches pool together the reactive components of the hybrid architecture in a separate subsystem (termed the *reactor* in [9]) to ensure that these activities receive adequate computational resources. Deliberative, time consuming activities are executed elsewere or scheduled as best-effort tasks in order not to interfere with time-critical ones. In [9] the reactor is described to be a real-time system executing a collection of behaviors (sensor-action pairs). The *Task Control Architecture - TCA* [12] introduces a real-time controller and includes a preliminary analysis of task time consumption. Typical AI-oriented approaches address real-time issues only indirectly. For example, the *Procedural Reasoning System - PRS* [6] and the *Reactive Action Package - RAP* [5] propose reactive planning mechanisms which can pay attention to the modified conditions of the environment during plan computation. The *3T* architecture [4] exploits RAP to implement the sequencer, a reactive module enabling or disabling sets of "skills" (low level behaviors) based on the dynamics of the environment. This adaptation of the reactor to the current task and environmental context also helps in reducing the overall computational load. While these approaches ensure reactivity in the specific application cases, they do not explicity address priority among actions nor evaluation of the real-time constraints that should be met in the reactor.

In the "cooperative architectures" in [10] and [11] the deliberative layer plays an active role in ensuring that deadlines will be met in the reactor in each phase of the task. A feedback from the reactive layer enables the AI subsystem to reason about logical and timing constraints. Proper decisions can be taken to avoid faulty operative conditions, by adapting the set of reflexes, changing the real-time parameters, or rescaling task level goals (e.g., by decreasing robot navigation speed). As deliberative and reactive layers are uncoupled, real-time execution is always guaranteed at the reactive layer, whereas flexibility and task-level robustness is demanded to the deliberative one. This idea pursues the design of a system which can reason about real-time needs and its own bounded computational capabilities to adapt the computation to the incoming events.

Focussing on the supporting tools, a few special purpose operating systems conceived for autonomous robots offer effective support to the design of control architectures while explicitly addressing the real-time issues [3, 15]. These operating systems provide real-time features such as timing faults detection, periodic computations, scheduling guarantees, sporadic servers, priority inheritance and priority ceiling. Hartik [3] takes a hard real-time stance, whereas Chimera [15] emphasizes the need for a soft real-time approach, with explicit management of events such as overload, overruns and timing faults [14]. Our approach is more along the lines of Chimera, although we attempt to leverage upon the features of a general-purpose operating system, even at the cost of a reduced real-timeliness.

2.2 Real-time requirements

According to overviews from leading experts of the field [13], future real-time systems will be increasingly based on the integration of real-time and non-real-time computations on a single platform. This platform will be often based on standard hardware and operating system, to take advantage from low cost and performance trends. We subjectively emphasize the following requirements for the real-time subsystem of autonomous robot control architectures:

1. The system must support many threads of execution (both cyclic and acyclic, independent and interacting) with efficient interaction and scheduling mechanisms.

2. As the control architecture will be reconfigurable, sudden variations in computational load are unavoidable; appropriate thread management mechanisms (creation, suspension, modification of scheduling parameters, etc.) should be available.

3. Due to sudden load variations and many sources of non-determinism in computing, even "guaranteed" real-time threads could eventually fail [14]. Timing fault handlers should be specified to manage a smooth transition from guaranteed real-time to best-effort scheduling or dynamic modification of real-time scheduling parameters.

4. Prioritized scheduling policies, where threads fail in a known order in case of overload, are of potential interest.

5. The system should allow interaction with reasoning engines and effective integration of available AI tools.

6. Reasoning *in real-time* is difficult or impossible to ensure in general conditions; the overall control architecture should possess the ability to reason *about real-time*, so as to make the best use of the scarce computational resources available [10, 11].

In most robot control systems, hardly a single real-time deadline miss (typically from a periodic thread) will determine a global system malfunctioning. Rather, occasional deadline misses will simply affect overall quality in task execution (e.g., trajectory smoothness). However, systematic misses or starving threads will eventually lead to a global malfunctioning and make the task fail. Thus most autonomous robot control systems can be regarded as consisting of *several soft real-time* threads and possibly only *a few hard real-time* threads. This remark also justifies our emphasis on timing fault management to prompt recovery and adaptation of the computation.

3 The real-time library

The `rt-lib` real-time library is built on top of Solaris 2.6, a general-purpose operating system of the UNIX family offering partial support to soft real-time computation [17]. Of course, Solaris is not a fully real-time operating system, and hence it would not be able to

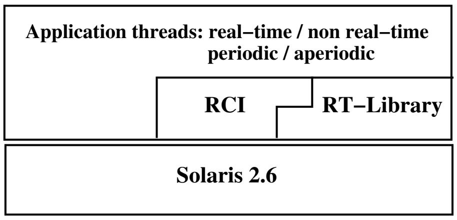

Figure 1: The integrated RCI/`rt-lib` library.

manage hard real-time applications with tight requirements.

A key requirement for structuring control architectures for autonomous robots is the availability of effective abstractions for modularization and concurrency. Several modern operating systems provide multithreading computing environments (e.g. the POSIX standard suite), where multiple threads of control share a single address space within a process entity. Threads interact efficiently by synchronizing their accesses to shared data structures using techniques such as semaphores and condition variables. Solaris provides both kernel level and user level threads, along with a number of real-time oriented features exploited in our library. These features include: reduced latency to interrupts, with frequent kernel preemption points; three separate scheduling classes for thread execution (*Real-Time, Sys, User*), where threads in the real-time class (owning a static priority) can preempt system tasks; priority inheritance for all binary semaphores; lock of process memory to avoid the paging latency; system calls and dedicated signals for high resolution real-time timers.

The `rt-lib` library implements mechanisms for periodic computations, handling of timing faults, monitoring of CPU utilization, and procedures that allow online modification of thread scheduling parameters (period, deadline and priority). The library promotes structured creation of periodic activities, including startup and release cycles, and explicit management of timing faults. Each thread is instrumented by default with tracing tools for systematic activity monitoring and off-line debugging. The types of threads that can be defined include periodic and aperiodic real-time threads, and non real-time threads. As shown in Figure 1 the library is also integrated with RCCL-RCI [8], a control library for robot manipulators.

We next outline the C language framework describing the real-time library. An `rt-lib` thread (`rt-thread`) is created by the `rt_thread_create` library procedure, invoked with the following parameters (`args`):

- activation time `A`, period `T`, and deadline `D` (in ms);
- `StartUpFnx` procedure, executed in the first cycle when specified;
- `PeriodicFnx` periodic procedure;
- `ReleaseFnx` procedure, executed in the last cycle when specified;
- `TimingFaultFnx` procedure, invoked by a timing fault signal handler when a timing error occurs; if not specified a default handler is executed;
- `arg`, pointer to a data area available to all the procedures of a `rt_thread_wrapper`.

The `rt_thread_create` procedure invokes the Solaris `thr_create` system call which executes the code implementing the real-time thread wrapper `rt_thread_wrapper` as specified in the following:

```
thread_t rt_thread_create(...,args,...)
{
 ...
 thr_create(...,rt_thread_wrapper,&args,...);
 ...
}
```

The `rt_thread_wrapper` works as follows:

```
typedef struct _ARGS {
 timespec_t A,T,D;
 void *userarg;
 void (*StartUpFnx) ();
 void (*PeriodicFnx) ();
 void (*ReleaseFnx) ();
 void (*TimingFaultFnx) ();
} ARGS;

void rt_thread_wrapper( void *args )
{
 ARGS *Data;
 Data = args;
 /* arm timing fault handler */
 evp.sigev_notify = SIGEV_SIGNAL;
 evp.sigev_signo = SIGUSR1;
 evp.sigev_value.sival_ptr = Data;
 clock_id=CLOCK_REALTIME;
 timer_create(...,evp,...);
 /* defer the activation time */
 rqtp.tv_sec = A.tv_sec;
 rqtp.tv_nsec = A.tv_nsec;
 nanosleep(&rqtp, &rmtp);
 /* exec startup */
 if(StartUpFnx!=NULL)
   state=STARTUP;
   rt_thread_fnx(StartUpFnx,Data->userarg);
 /* exec periodic */
 if(PeriodicFnx!=NULL){
   state=PERIODIC_EXEC;
   do {
     rt_thread_fnx(PeriodicFnx,Data->userargs);
   } while(state!=TERMINATE);
 }
 /* exec release */
 if(ReleaseFnx!=NULL)
   rt_thread_fnx(ReleaseFnx,Data->userargs);
}
```

A Solaris real-time timer T is enabled to handle time fault occurrences. Then the `StartUpFnx` is executed at the first cycle of the `rt_thread` computation. The `do..while` loop executes the periodic computation of the `PeriodicFnx` until a request from the upper layer switches `state` to `TERMINATE`. Finally the `ReleaseFnx` is invoked.

The `StartUpFnx, PeriodicFnx, ReleaseFnx` procedures are executed by the `rt_thread_fnx` as shown below:

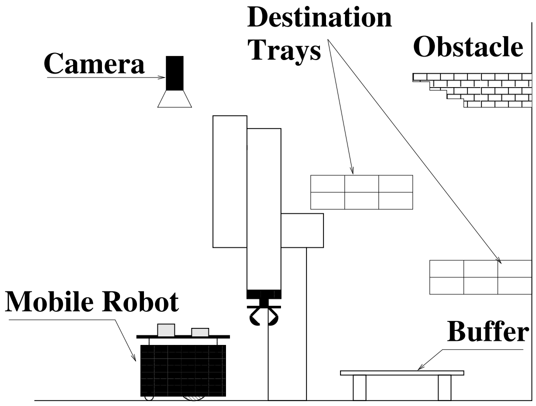

Figure 2: The robot workcell including the mobile robot and the Puma manipulator.

```
void rt_thread_fnx(void (*fnx)(),void *arg)
{
 startR = gethrtime();
 startV = gethrvtime();
 /* set timer to the current thread deadline */
 if( time_notzero(D) )
   timer_settime(ID,Deadline,NULL);
 /* execute fnx */
 fnx( arg );
 /* reset timer */
 timer_settime(ID,0, NULL);
 endV = gethrvtime();
 endR = gethrtime();
 /* get computation time */
 C = (endV-startV);
 /* sleeping */
 if( time_notzero(T) ) {
   rqtp =
   time_diff(T,time_diff(D,(endR-startR)));
   if(rqtp>0)
      nanosleep(&rqtp, &rmtp);
   else
       overrun++; }
}
```

The `gethrtime, gethrvtime` system calls return the real time clock and virtual time values of the current thread. The first value is used to compute the wall-clock execution time of each thread, whereas the second one is exploited to trace the computation time for time monitoring purposes. The timer T is armed at the beginning of every loop (`timer_settime` call) with timeout set to the thread deadline. When the timer expires, a signal is delivered and the proper `TimingFaultFnx` is invoked. `rt-lib` provides default `TRACE` or `IGNORE` policies for these events if no specific handler is supplied by the user [14]. Finally, the `nanosleep` Solaris system call is invoked at the end of each cycle with a sleeping time depending upon the wall-clock execution time and the thread period.

4 The kitting task

We have developed an example domain to evaluate the effectiveness of the proposed real-time library (Figure 2). A sensorized Puma manipulator and a small mobile robot cooperate in performing a kitting task. The goal of the manipulator is to put the blocks carried by the mobile robot in proper slots in trays located in its workspace. Tray positions are known and do not change during execution of the kitting task. The mobile robot is loaded by a human operator with a variable number of blocks. It travels about 1.5 m until it detects the approximate unloading point (marked with black tape) using infrared sensors. Once at the unloading point, the mobile robot waits until its upper platform has been cleared of blocks (detected using a set of top mounted infrared sensors) or for a fixed amount of time, whichever occurs first, and then returns back to the loading position. This cycle is iterated with random arrival time.

The Puma monitors the arrival of the mobile robot using a vision system. There is no direct, explicit communication between the robots: the number, type, position and rotation of the objects carried by the mobile robot are also detected using the vision system. A small intermediate buffer, shown in Figure 2, serves the purpose of partially decoupling the unloading of the mobile robot from the actual delivery of objects to their destination trays. During each unloading task the manipulator actively detects the current position of the intermediate buffer using an infrared sensor mounted near the gripper. Sources of uncertainty in this task include exact position of the mobile robot, number, position, orientation and type of objects, and position of the intermediate buffer. The manipulator must overcome these uncertainties, move the blocks from the mobile robot deck to the intermediate buffer, and plan collision free paths to bring each block to its final tray and slot destination.

This task involves two basic types of motion of the manipulator: point-to-point motions parametrized by the sensor data (to move the objects from the mobile robot to the buffer), and path planned motions (to move the objects to the trays). The latter type of motion involves computation of a free path in a 3D space for a 6 d.o.f. manipulator in a workspace including obstacles. This is done by a planner evolved from RPP, a non-deterministic but probabilistic-complete path planner [1]. Even though RPP is a fairly advanced planner, in this environment path computation is very time consuming (tens of seconds on the μSPARC robot controller CPU).

Since the mobile robot could leave the meeting point at any time, the Puma attempts to unload it as soon as possible with the help of the intermediate buffer, while beginning also to compute a path based on the current data. However, due to the limited size of the buffer this strategy is no longer applicable once the buffer contains three objects. At this time freeing one position in the buffer becomes of prominent importance.[1] If no path is available yet, the control architecture should increase the priority of path computation, so as to minimize the time to path availability. This is done at the expense of other computations which become less important since the robot is idle waiting for a path. In the specific application, this dynamic adaptation of CPU utilization is obtained by reducing the rate of several periodic threads performing perceptual and monitoring activities. Once a path has been computed, the original computation rates are restored.

[1] We assume for simplicity that all blocks must transit through the buffer. This constraint has no impact on the underlying issues.

Agent	Instances	Timing	rt-pri
Supervisor	1	Periodic	–
Motion Planner	n	Aperiodic	1
Motion Executor	2	Aperiodic	2
Timing Monitor	1	Periodic	RM
Sensor Manager	2	Periodic	RM
Guard	n	Periodic	RM
Trajectory Control	1	Periodic	59

Table 1: The agents of the architecture. "Instances" refers to the number of agents for each type. "rt-pri" is the Solaris real-time priority.

5 The hybrid control architecture

A hybrid control architecture for the kitting task has been implemented by means of the real-time library. We denote each structuring element of the architecture as an "agent". The various types of agents are listed in Table 1 along with their category and real-time properties. There may be more than one instance for certain agent types, each implemented as a separate thread.

Straightforward deliberative activities are carried out by the *Supervisor*, where the limited form of planning required by the task takes place. In each phase of the task the *Supervisor* determines which *Motion Executor* agent has to be activated, and when to request a new path to the *Motion Planner* agent. A *Motion Executor* agent pursues the execution of either a parametrized motion (go-above-mobile-robot, grasp-a-block, go-above-buffer, etc.) or a path-planned motion (sequences of robot displacements). The latter ones are computed by the *Motion Planner* agent encapsulating the modified RPP planner, which determines the manipulator path to a specific tray location.

Whenever the buffer becomes full, the *Supervisor* favors path computation by tuning the real time parameters of the reactive layer (thus decreasing its global computational load), in order to reduce the wall clock completion time for the entire kitting task. Overruns and timing fault statistics are managed by the *Timing Monitor* agent, and related information is written in a shared area. *Sensor* agents read data incoming from two external subsystems (a DataLogic DVS1000 industrial vision system and an infrared/ultrasonics control unit) and store them in a shared area. *Guard* agents periodically analyze sensor information and notify the *Supervisor* when the conditions they are evaluating are satisfied. The *Trajectory Loop* agent carries out the trajectory generation algorithm and drives the actual Puma motions within RCCL/RCI [8].

As shown in Table 1, the *Supervisor* is implemented as a non real-time periodic `rt-thread` while *Motion* agents are defined as low priority aperiodic real-time `rt-threads`. All remaining agents, defining the reactive layer, are periodic real-time `rt-threads`. Their priority has been assigned in Rate Monotonic fashion [7] using Solaris RT priorities between 3 and 59. For all threads the `TimingFaultFnx` is set to the default `TRACE` policy.

6 Experimental results

The kitting task has been performed a number of times as described in Section 4. Whenever the *Supervisor* notices that the buffer is full and no path is available, it induces a dynamic adaptation in CPU utilization to favor the *Motion Planner* agent. This is obtained exploiting `rt-lib` features to decrease the rate of periodic real-time threads implementing *Sensor, Monitor* and *Guard* agents.

Due to the high variance of the path planning algorithm and the dynamics of the kitting tasks, in each individual experiment it is somehow difficult to assess the impact of rate adaptation on task performance. A dedicated set of experiments thus has focused on the quantitative evaluation of the beneficial effects that can be obtained in terms of task-level performance by changing the real-time parameters of periodic `rt-threads`. As a measure related to task-level performance we evaluate the effect of rate changes on the wall-clock completion time of the aperiodic *Motion Planner* agent computing an RPP-based path. More specifically, in these experiments the rates of the *Sensor* and *Timing Monitor* agents have been modulated, whereas no perturbation has been introduced on the *Trajectory Loop* to avoid injecting disturbances during trajectory generation. The *Trajectory Loop* is always executed at the highest priority (Solaris RT priority = 59) with a period of 30 ms and an average execution time of about 8 ms. In these dedicated experiments no *Guard* agent is active.

Figure 3 shows both the average wall-clock completion time and the average execution time for the *Motion Planner* agent in a set of seven experimental runs. Each reported value is the average over ten executions. Periods for the `rt-threads` listed in Table 2 have been de-

`rt-thr`	W	T_1	T_2	T_3	T_4	T_5	T_6	T_7
Monitor	1	800	400	200	100	60	40	30
Sonar, IR	1.1	400	200	100	50	30	20	15
Camera	12	400	200	100	50	30	20	15

Table 2: Rate-modulated threads: WCET and periods used in the seven experimental runs (times in ms).

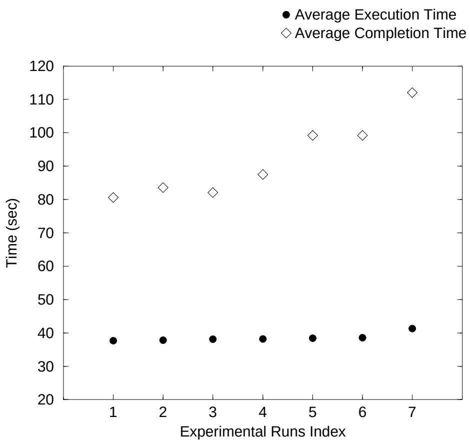

Figure 3: Wall-clock completion times and execution times for the *Motion Planner* in seven experiments with increasing `rt-threads` rate.

creased along the seven experiments, while keeping the rates harmonic. Worst Case Execution Time (WCET) obtained using the `gethvtime` system call has been taken into account for Rate Monotonic analysis. The average wall-clock completion time increases from 80.6 s with the first set of values in Table 2 to 112.0 s with the last one. Even though additional experiments are needed, these data show the tradeoffs that can be pursued by dynamically allocating computational resources between deliberation and reaction.

The lower priority *Timing Monitor* thread reports sporadic deadline misses and overruns in experiment #6 and systematic misses in experiment #7. The *Sensor* threads also report sporadic misses in experiment #7. Experiments #6 and #7, indeed, are executed with no scheduling guarantee and lead to CPU overload because of beyond-capacity requested utilization [7]. As threads are still scheduled with RMS policy, lower priority threads may miss their deadlines and incur in overruns. Due to the overload, the system actually executes the computations at a lower rate than required. Such operating condition is detected but ignored by `rt-lib` timing fault handlers, since this status is intentionally induced in these experiments.

In spite of the high variance of the probabilistic planner, the average execution time in the first six experiments is nearly constant at about 38.1 s and is only marginally affected by rate modulation, as expected. Only in experiment #7 the execution time is somehow influenced by the peculiar overload conditions.

7 Conclusion

The paper has described a real-time library providing facilities useful in the design of robot control architectures. The library supports structured creation of reactive and deliberative modules, dynamic modification of relevant real-time parameters, generation of timing fault handlers, measurement and monitoring of execution times.

As further developments, we plan to increase the complexity and dynamics of our reference kitting task and assess in more general terms the effect on task performance of `rt-threads` rate changes (possibly including the *Trajectory Loop*). We also plan to extend the library by introducing aperiodic server mechanisms (polling, deferrable, sporadic) to effectively handle asynchronous events. Finally, we will investigate architecture level policies and mechanisms to fully exploit on-line rate adaptation of the reactive layer while preserving real-time scheduling guarantees.

References

[1] J. Barraquand, J.C. Latombe, "Robot Motion Planning: A Distributed Representation Approach," Int. J. of Robotics Research, Vol. 10, No. 6, December 1991.

[2] R. Brooks "A Robust Layered Control System For A Mobile Robot", IEEE J. of Robotics and Automation, Vol. RA-2, No. 1, March 1986.

[3] G. Buttazzo, M. Di Natale, "HARTIK: A Hard Real-Time Kernel for Programming Robot Tasks with Explicit Time Contraints and Guaranteed Execution", IEEE Int. Conf. on Robotics and Automation, ICRA'93, Atlanta, GA, May 1993.

[4] R.P. Bonasso, R.J. Firby, E. Gat, D. Kortenkamp, D. Miller, M. Slack, "Experiences with an Architecture for Intelligent, Reactive Agents", J. of Experimental and Theoretical Artificial Intelligence, Vol. 9, No. 1, 1997.

[5] R.J. Firby, "An Investigation into Reactive Planning in Complex Domains", Nat. Conf. on Artificial Intelligence, AAAI'87, 1987.

[6] F.F. Ingrand, R. Chatila, R. Alami, F. Robert, "PRS: A High Level Supervision and Control Language for Autonomous Mobile Robots", IEEE Int. Conf. on Robotics and Automation, ICRA'96, Minneapolis, MN, April 1996.

[7] C.L. Liu, J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", J. of the ACM, Vol. 20, No. 1, January 1973.

[8] J. Lloyd, "RCCL/RCI User's Guide", Computer Science Department, University of British Columbia, Vancouver, Canada, June 25, 1996.

[9] D.M. Lyons and A.J. Hendriks, "Plannig for Reactive Robot Behavior", IEEE Int. Conf. on Robotics and Automation, ICRA'92, Nice, France, May 1992.

[10] D.J. Musliner, E.H. Durfee, K.G. Shin, "CIRCA: A Cooperative Intelligent Real-Time Control Architecture", IEEE Trans. on Systems, Man, and Cybernetics, Vol. 23, No. 6, November/December 1993.

[11] M. Schoppers, "A Software Architecture for Hard Real-Time Execution of Automatically Synthesized Plans or Control Laws", Conf. on Intelligent Robotics in Field, Factory, Service, and Space (CIRFFSS'94), AIAA, March 1994.

[12] R.G. Simmons, "Structured Control for Autonomous Robots", IEEE Trans. on Robotics and Automation, Vol. 10, No. 1, February 1994.

[13] J.A. Stankovic, et al., "Strategic Directions in Real-Time and Embedded Systems", ACM Computing Surveys, Vol. 28, No. 4, December 1996

[14] D.B. Stewart, P.K. Khosla, "Mechanisms for Detecting and Handling Timing Errors", Communications of the ACM, Vol. 40, No. 1, January 1997.

[15] D.B. Stewart, D.E. Schmitz, P.K. Khosla, "The Chimera II Real-Time Operating System for Advanced Sensor-Based Control Applications," IEEE Trans. on Systems, Man, and Cybernetics, Vol. 22, No. 6, November/December 1992.

[16] D.B. Stewart, P.K. Khosla, "Real-Time Scheduling of Dynamically Reconfigurable Systems", IEEE Int. Conf. on Systems Engineering, Dayton, OH, August 1991.

[17] SunSoft, "Multithreaded Programming Guide", Mountain View, CA 94043, USA, August 1994.