Hindawi Publishing Corporation
Journal of Robotics
Volume 2011, Article ID 525724, 11 pages
doi:10.1155/2011/525724

Research Article

Using Sun's Java Real-Time System to Manage Behavior-Based Mobile Robot Controllers

Andrew McKenzie,[1] Shameka Dawson,[2] Fei Hu,[1] and Monica Anderson[2]

[1] *Department of Electrical and Computer Engineering, The University of Alabama, Tuscaloosa, AL 35487, USA*
[2] *Department of Computer Science, The University of Alabama, Tuscaloosa, AL 35487, USA*

Correspondence should be addressed to Andrew McKenzie, awmckenzie@crimson.ua.edu

Received 7 May 2011; Revised 16 September 2011; Accepted 16 November 2011

Academic Editor: Yuan Zheng

Implementing a robot controller that can effectively manage limited resources in a deterministic, real-time manner is challenging. Behavior-based architectures that decompose autonomy into levels of intelligence are popular due to their robustness but do not provide real-time features that enforce timing constraints or support determinism. We propose an architecture and approach for using the real-time features of the Real-Time Specification for Java (RTSJ) in a behavior-based mobile robot controller to show that timing constraints affect performance. This is accomplished by extending a real-time aware architecture that explicitly enumerates timing requirements for each behavior. It is not enough to reduce latency. The usefulness of this approach is demonstrated via an implementation on Solaris 10 and the Sun Java Real-Time System (Java RTS). Experimental results are obtained using a K-team Koala robot performing path following with four composite behaviors. Experiments were conducted using several task period sets in three cases: real-time threads with the real-time garbage collector, real-time threads with the non- real-time garbage collector, and non-real-time threads with the non-real-time garbage collector. Results show that even if latency and determinism are improved, the timing of each individual behavior significantly affects task performance.

1. Motivation

Developing a mobile robot controller can be a complex process. Robots must simultaneously perform safety checks (such as obstacle avoidance) while accomplishing high-level goals (mapping and path following). Behavior-based controllers [1] use prioritized finite state machines to manage layers of complexity. Existing robot architectures either enable this approach [2, 3] or are purposefully unstructured to accommodate behaviors [4].

Although behavior-based paradigms provide robust controllers in the face of complex conditions, real-time issues are not addressed as part of these architectures. Tasks may need to be executed and completed within a time constraint to prevent undesired results. Since timing and determinism matters [5], should robot controllers be programmed using real-time control system standards, allowing tasks to execute in actual real time?

Currently, real-time features and specific frequency requirements are not included in mobile robot architectures [6]. Timing constraints, determinism, and low latency are important factors to ensure proper operation of the robot and can result in better resource utilization. By not having an actual real-time controller that explicitly addresses these issues, developers may get erratic results causing inconsistent and inefficient robot behavior.

If the laws of control theory [7] were to be applied to a behavior-based system, the frequency of each individual robot behavior would be based on input availability. The definition of individual frequency requirements for subsystems within a control system is part of the process for traditional real-time system development [8]. Frequencies define the period and the deadline for processing sensor input. If a task misses its deadline, the system is not responsive to changes in input. However, it is not enough to run all behaviors "as fast as possible." In fact, some resource intensive behaviors

may take resources and/or preempt more time-sensitive behaviors. Slowing down behaviors that do not receive input frequently may free resources for more time-sensitive tasks.

In order to address this timing constraint issue, we examine adding the Real-Time Specification for Java (RTSJ) [9] standard to an existing mobile robot architecture [10], using Sun's Java Real-Time System (Java RTS), and running it on a real-time operating system (RTOS). Java, a language known for its portability and safe object-oriented features, has become an option for developers of real-time systems [11]. While languages such as C and C++ pose an issue for many real-time developers because of the memory issues associated with them, they are the most common choice because of their performance and ease of use [12]. As an alternative, Java RTS is designed to be a system that has deterministic, real-time features [13] without forcing developers to manage memory usage. This makes Java an appropriate choice for programming a real-time robot architecture. However, Java is timing sensitive. Particularly the traditional garbage collector may cause nondeterministic behavior in a real-time system. The real-time garbage collector (RTGC) implemented in Java RTS provides a deterministic approach to dealing with the garbage collector.

The robot architecture that we propose has several fundamental purposes. First, real-time issues pertaining to robot controllers will be made explicit. Secondly, an effective way to use real-time Java within a behavior-based mobile robotics framework will be demonstrated. By also allowing developers to declare frequency requirements, tasks can be run only as fast as needed. Allowing frequencies to be declared can ultimately provide the developer with a better understanding of the frequencies and their relationship to code features. This includes which tasks depend on specific input devices as well as the dependencies between different tasks.

It is possible that such an architecture will help move embedded systems students up the learning curve faster. In [14], a real-time systems undergraduate laboratory that was created for Java programmers is described. It was found that students taking the real-time software engineering module performed better and enjoyed it more than other modules. Also, the majority of the students who took this module found work in the real-time and embedded systems industry.

The results of this work widely impact both current and future behavior-based robot development environments. Robot architectures need to manage real-time constraints in order to effectively behave in a deterministic manner. Experimental results show that timing of individual behaviors do affect the robot's high-level task performance.

In this paper, we map a behavior-based robot architecture to Java RTS in order to show that the absence of or incorrect timing constraints affect performance. The architecture allows developers to declare frequency requirements as part of the client interface. Java RTS features are also added to help better manage resources. The main contribution of this paper is to create an off-the-shelf system that allows real-time control of autonomous robots. To our knowledge our architecture was the first one made for fully autonomous mobile robots doing complete tasks, using off-the-shelf OSâ€s and languages. In Section 2, background and previous work is presented. Control theory and its relevance to our approach is presented in Section 3. Section 4 describes Java RTS and its relevance to the interface. Section 5 presents the implementation of the architecture and the experimental setup. The experimental results and the analysis are presented in Sections 6 and 7, respectively. Conclusion and future directions are in Section 8.

2. Background

2.1. Real-Time in Mobile Robotics. Several robot development platforms are available for mobile robots. However, some of these architectures do not provide real-time features that enforce timing constraints. Of the ones that do provide real-time features, they do not allow developers to provide their own frequency requirements. Buttazzo et al. [15] conducted research in real-time control of robots where they designed and implemented a software architecture used for designing robotic applications with real-time constraints. The architecture was a hierarchical programming environment consisting of four parts: an action layer, control layer, communication layer, and a hard real-time kernel (HARTIK) to control the three layers. All of the layers were written in C and encapsulated with a set of library functions. They concluded that their architecture provided a flexible framework for the development of robotic applications. However, they did not provide any experimental results.

Brega et al. [16] conducted a case study of the XO/2 operating system controlling a Pygmalion Robot in real time using the Oberon-2 programming language. They discussed the needs and requirements for real-time control of robots in research, education, and real-world use. The XO/2 operating system was designed by Brega et al. as a real-time operating system. They concluded that they could not quantify the need for self-contained autonomy, real-time, and safety in mobile robots. They also noted that the increasing complexity of mobile robots set higher requirements for the robots' hardware, software, and indirectly the operating system. Finally, they noted that "safe composition of software modules, type-safety, deadline-driven scheduling and automatic memory reclamation mechanisms can relieve the application programmer from many time-consuming implementation issues, while raising the safety-bar."

Auerbach et al. [17] implemented a real-time Java application that used Exotasks, a novel Java programming construct that achieved deterministic timing, on a real-time virtual machine. They built a helicopter, the JAviator, for testing. They showed that their implementation achieved time portability on different hardware platforms. However, all of their data was processed off-board and they only observed one behavior (hovering of the helicopter).

2.2. Real-Time in Non-Mobile Robotics. Stewart et al. [18] designed a real-time architecture that is based on port-based objects. Port-based objects are fully contained software components that contain both state and methods as well as provide input, output, and resource ports. Input ports

describe data needed by an object, and output ports describe data produced by an object. An input port can only be connected to exactly one output port. Outputs of the same type must be merged into one unambiguous output. Each object is self-contained and has its own frequency. Task scheduling is determined by the developer and can be modified to either increase system stability or reduce computational resource consumption.

Bruyninckx et al. [19] implemented a 1 kHz position, velocity, and hybrid force control of a KUKA 361 six-DOF robot using the hard real-time motion control core of the OROCOS project (Open Robot Control Software) [20]. They bypassed the KUKA legacy controller and controlled it using only the OROCOS code. The code runs on Linux (without real-time performance) and on RTAI (RealTime Application Interface for Linux) in hard real time.

Robertz et al. [21] implemented a motion control system for an industrial robot using Java RTS on an RTOS. They used real-time threads (RTTs) for critical tasks and non-RTTs for noncritical tasks. In their approach, the RTTs were run at the highest priority and the non-RTTs were run at a lower priority, even lower than the real-time garbage collector. Although they discuss time measurements of network delays, they do not present results showing the effect of these real-time features on the overall system.

2.3. Behavior-Based Architectures. Subsumption-based robotic controllers build autonomy out of layers of behaviors [22]. Basic layers are complete processing units that take input and provide appropriate output. We are particularly interested in properties that enable temporal decompositions. Aria [2] is a robotic architecture that provides support for behavioral decompositions. Aria programs are composed of individual behaviors that are each given a priority that is applied to the importance of actuator output. Priority does not affect resource assignment or utilization. Because the target hardware architecture only processes commands every 100 ms, all behaviors are run every 100 ms. Other threads can be added by the developer outside of the framework for processing tasks that have different frequency requirements.

Carmen [23] is an open source modular architecture for mobile robot control. It features a three-tier architecture, where the first layer is the hardware interface, the second layer contains the basic robot tasks, and the final layer is the user-defined application. It uses interprocess communication to provide functionality for mobile robots and supports various platforms.

The OROCOS project [20] is an open source modular framework for general-purpose robot and machine control. The OROCOS project supports four C++ libraries: the Real-Time Toolkit, the Kinematics and Dynamics Library, the Bayesian Filtering Library, and the OROCOS Component Library.

Although not a behavior-based architecture, Player [4] is an open source robot architecture designed to operate with a wide range of hardware components. The Player architecture specifically avoids defining decompositions and therefore does not provide a framework for adding frequency for behaviors.

3. Control Systems

Control systems are devices or systems that control another device or system. Closed loop control systems use the systems output (feedback) to modify the input to get the desired output. Today most control systems use computers to make all of the controller decisions, called digital control. Since digital control uses computers, which make decisions every clock cycle, the signal is discrete.

On the basic level, autonomous robots use computer-run algorithms to control its movement, based on input from sensors such as sonar, infrared, GPS, and laser range finders. Therefore, an autonomous robot qualifies as a control system because it is a device (a computer) that controls other devices such as motors. So autonomous robots should be treated as control systems and be required to follow the same rules that digital control systems follow.

Since digital control is a digital (discrete) time system, problems can occur if input data is not being sampled fast enough. This happens if the input signal changes between samplings. For example, if an input has two values A and B, when value A occurs, the signal will be transmitting data for a minimum of 80 ms, and B for 50 ms. The input is sampled at 10 Hz (once every 100 ms). The system samples the input signal at times 100 ms, 200 ms, 300 ms, and so on. The input signal for A is active from time 0 to 130 ms, changes to B at time 130 ms, then changes back to A at 180 ms. Since the sampling rate occurs every 100 ms, the digital control system sees A at time step 1 and 2. So it does not see B, which could cause problems. The simple fix is to make sure that your sampling time is fast enough.

In control theory, the sampling rate should be at least twice the fastest input signal [7]. In the above system, the fastest input signal B occurs a minimum of 50 ms (20 Hz). So the system needs to have a minimum sampling rate of 40 Hz. This problem is easy to fix by making sure that the system can get an input at a rate of 40 Hz, which can be done by software or upgrading to hardware that can sample a signal at 40 Hz. This example directly relates to mobile robotics. Simply consider any digital sensor, such as a laser range finder, as the source for the input signal. In the case of a Hokuyo PBS-03 laser range finder, which produces laser readings every 180 ms [24], if the system (i.e., controller) does not run at least every 90 ms, then laser readings will be missed. If a reading is missed, it is possible for the system not to see an object which could produce an incorrect map or, the worst case, cause the robot to collide with the missed object. According to [7], in order to improve the performance of a control system, the advanced rules from control theory should be applied. Similarly, a robot controller using only the basic design principles will still work. However, if more advanced control techniques are applied, performance will improve.

4. Java Real-Time System

In order for Java to be considered a programming language for real-time systems, several criteria are required [11]. First of all, it must provide determinism. When programming

a robot controller in Java, it is essential that the robot behaves in a predictable manner and that all tasks meet their deadlines. Low latency is another important criteria. Latency is the difference between when an event happened and when it is seen to have happened. In essence, there should be a bounded and known maximum delay between the two to ensure that a task completes before the end of the period in which it is executed. Also, the timing of the overall system is a huge factor. Robot behaviors should be executed in a certain timeframe and run only as fast as needed.

We extended the Java client to include features available for Java RTS. It provides developers with a platform where they can accurately determine the temporal behavior of their software in a real-time fashion. This allows programmers to determine how often tasks will execute in their code, therefore making it possible to schedule tasks so they can be completed before they miss their deadline. In order to use the features of Java RTS, the *javax.realtime.RealTimeThread* class is used which extends *java.lang.thread*. Java RTS enhances the Java specification by adding additional features, which include real-time threads, scheduling, memory management, and a real-time garbage collector. Java RTS includes a built-in scheduler, which follows a fixed-priority preemptive policy.

4.1. Real-Time Threads. There are no guarantees for thread scheduling and thread priorities when using the regular Java specification. Java RTS provides real-time threads which offer more accurate scheduling than regular Java threads. The Java RTS also provides thread priority which is strictly enforced. In a real-time system, if there are higher-priority tasks that need to be executed, then lower priority tasks will be sacrificed. Therefore, higher-priority tasks are guaranteed to execute so they are less likely to miss their deadline.

4.2. Memory Management. With the traditional Java virtual machine (JVM), some tasks may contend with the memory management costs of other tasks. However, memory management is automatic using Java RTS. The NoHeapRealtimeThread (NHRT) class, a subclass of the RTT class, uses scoped and immortal memory so that memory can be allocated and reclaimed in a more predictable manner. It does not use the heap memory which the garbage collector is responsible for.

Immortal memory is the area in memory that is not garbage collected. This form of memory is used to avoid dynamic allocation. All the memory is statically allocated ahead of time, and it remains in use until the JVM terminates. Scoped memory is available to RTTs and NHRTs only. This type of memory has a defined lifetime. Objects allocated from scoped memory will stay allocated until the scope is no longer active. Through scoped memory, Java RTS provides developers with an environment that does not suffer interference from the garbage collector.

In contrast, memory management is done manually in C and C++. C and C++ have three kinds of memory: static memory, automatic memory, and free store [25]. The structure of a program, which is written by the developer, determines when memory is allocated and deallocated by using the malloc and free functions. Therefore, the lifetime of the program is determined by the logic of the program. However, in Java the developer does not have control over releasing the memory. This is all done by the garbage collector.

4.3. Garbage Collector. There is no guarantee within the JVM that garbage collection will not interfere with the timing of code since the garbage collector preempts the execution of all threads while it is running. A real-time machine must find ways to deal with this issue. Java RTS uses an RTGC that is based on Henriksson's garbage collector [26] that deals with hard real-time scheduling. The garbage collector runs as an RTT. These threads run at a priority that is lower than all the other RTTs; therefore, their execution is deferred until the critical tasks finish. This works well as long as the critical threads do not run out of memory during the threads' execution. If so, the RTGC will be forced to run when the program's memory usage reaches a certain level. This is implemented by increasing the RTGC's priority when the level is reached. The priority goes back to the initial setting when the usage levels go below the defined threshold. Therefore, the overhead of the RTGC is reduced and determinism of the threads is not affected.

5. Approach

5.1. Implementation. In [10], a real-time aware architecture was proposed, tested, and had promising results. The architecture used Player to interface to the robot hardware and Java for the control code. It was made real-time aware by creating a Monitor module that scheduled tasks and reported missed deadlines. However, it was not an actual real-time system. In order to be real-time, it would need to be run on an RTOS in a real-time compatible language. Player, which is written in C, does not support real-time capabilities [6]. Therefore, our robot architecture was rewritten entirely in Java with the addition of Java RTS. The architecture used soft real-time constraints. Soft real-time constraints are time constraints that will not necessarily cause the system to suffer a critical failure if the constraints are not met. The architecture applies time constraints to all tasks. The client program was interfaced to various hardware platforms such as the URG laser and the robot via serial connections. The drivers for the architecture translated generic commands to specific hardware interfaces.

The structure of the new architecture that we propose is shown in Figure 1. The framework is implemented as part of the client program. Behaviors are created by subclassing Tasks. The Tasks superclass contains functionality needed for initialization and registration of data store use. Tasks provides methods for registering production or consumption of data with the DataManager. During execution, the Tasks superclass method replicates consumed data to a local store. All data is created by the RobotController when the Khepera and Laser interfaces are created and sent to all tasks. All tasks were treated as time-critical tasks because they directly or indirectly controlled the robots

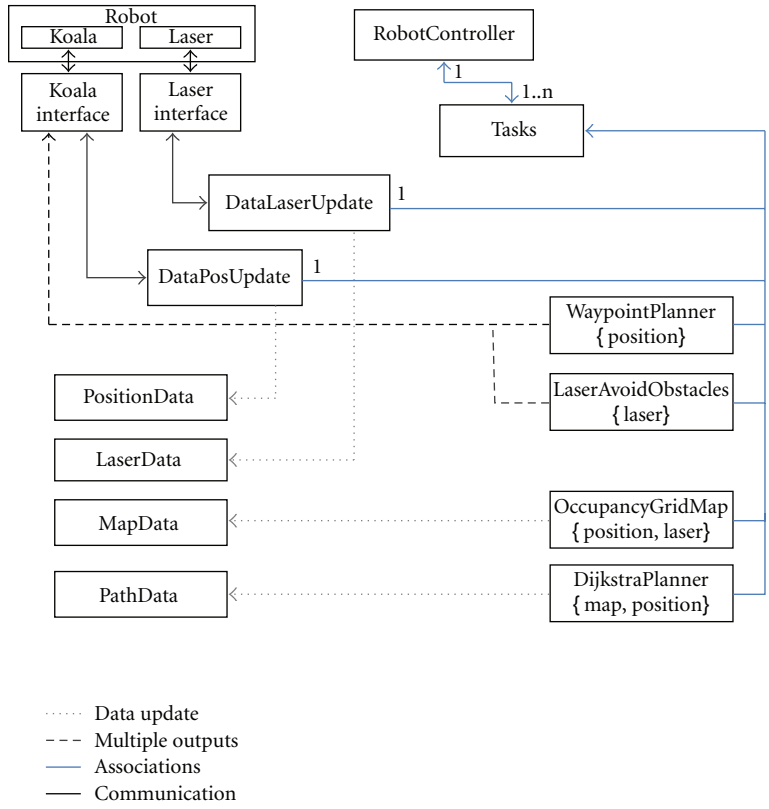


Figure 1: Framework for the robot architecture.

actuators. The scheduling is done through the Java RTS scheduler using RTTs. All of the tasks run in separate RTTs. Timed tasks are executed according to their specified period. The RobotController initializes and starts the threads. Once the RTTs signal that a task has run too long, each task's ranTooLong() method reports deadline misses. A class diagram depicting the central components of the architecture is shown in Figure 2.

5.2. Experimental Setup. Experiments were performed to measure the usefulness in applying different frequency requirements to behaviors. These frequencies were supplied by the user via the command line. A waypoint navigation controller was used to measure the effects of frequency on task performance.

The program structure has seven key components, a DataManager and six task threads. The six task threads are summarized in Table 1. Other important modules include the Khepera class. It takes movement requests from LaserAvoidObstacles and WaypointPlanner and determines what commands are ultimately sent to the robot. Motor control is ordered so that safety-related activities, such as obstacle avoid and stall recovery, get the opportunity to control the motors first. If both LaserAvoidObstacles and WaypointPlanner request control of the motors, LaserAvoidObstacles commands are executed since it is a safety-related task. WaypointPlanner only gets control of the motors when LaserAvoidObstacles does not want control.

Physical experiments employed a K-team Koala robot (Figure 3). The robot was equipped with an Acces I/O ETX-Nano computer which has an Intel Core Duo 1.66 GHz processor, 2 GB of RAM, and an 8 GB compact flash card for storage. It is running Sun Solaris 10 RTOS, which requires a dual-core processor [29]. Instead of using the onboard IR proximity sensors, the Koala is augmented with a Hokuyo URG laser range finder. The Java program handles the interface to the robot and its devices via serial connection through *javax.comm* (the Java Communication API). All Java code, including the logic and control code, is executed onboard the robot. Because of the high power requirements of the Acces I/O ETX-Nano, the robot was tethered with a power cord and an ethernet cat5 cable for remote communication. The testing environment was a 9.6 m $\times$ 6.2 m room (Figure 4).

Three different experiments were conducted: those using RTTs with the RTGC, RTTs with the non-RTGC, and regular threads with the non-RTGC. In these experiments, all trials were given the same four waypoints to reach (shown in Figure 4). From the start position, the robot would travel to each waypoint until it arrived at the finish position. The waypoints were chosen such that all tasks needed to be employed to reach the ending position.

Each task was given an interval value that defined both the task's deadline and period. For comparison, the robot was run with different interval values for the task threads. Each

Table 1: Task threads used in the robot architecture.

Tasks	Description	Provides	Requires
DataPosUpdate	Sends movement commands to the robot and receives updated position information.	PositionData	
DataLaserUpdate	Gets the laser readings from the URG laser.	LaserData	
WaypointPlanner	A high-level behavior that moves the robot through a series of predefined waypoints using a path created by the DijkstraPlanner.		PositionData, LaserData, PathData
LaserAvoidObstacles	Responds to obstacles sensed by the laser, via DataLaserUpdate, by slowing down the robot and turning away from the obstacle, via DataPosUpdate.		LaserData
OccupancyGridMap [27]	Uses the position and laser data reported by DataPosUpdate and DataLaserUpdate and maintains an occupancy grid map. The map tracks unknown, open, and occupied space.	MapData	LaserData PositionData
DijkstraPlanner [28]	Takes goal position requests and maintains a path from the current position to the closest goal. Multiple goals may be added and the requester must remove the goal when it is no longer valid (has been reached).	PathData	MapData

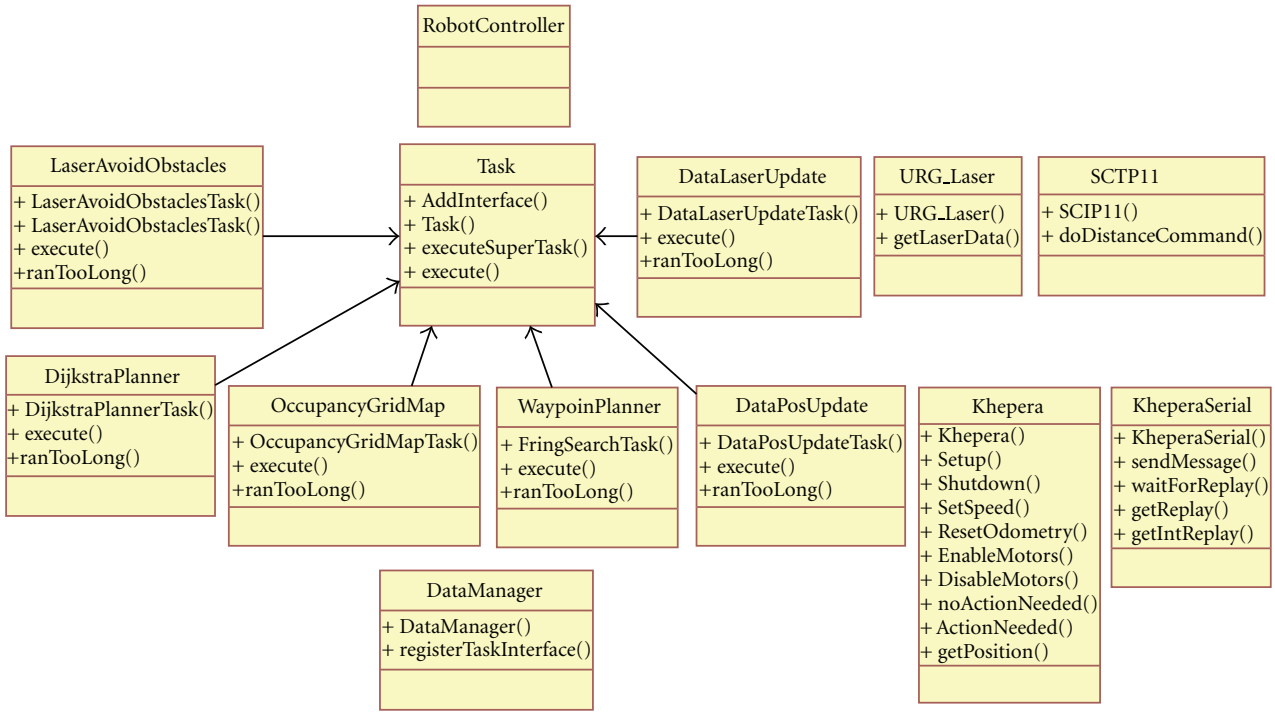


Figure 2: Class diagram of the essential parts of the interface.

set of experiments had a total of five runs. The initial period values were based on the frequency of the URG laser, which is 100 ms [30]. The period for each set thereafter was chosen based on observation. In three of the five experiment sets, all tasks were given the same period. In the remaining two sets, the period of the LaserAvoidObstacles was varied to see its effect on the overall performance of the experiment. We chose to lower this period because of safety issues of both the robot and environment. The period of DataPosUpdate and DataLaserUpdate was set at 75 ms and 100 ms, respectively.

It was hypothesized that these experiments would help demonstrate the effect of task frequency and determinism on the overall system. Trials were considered complete if they circled the obstacles approaching and passing each waypoint.

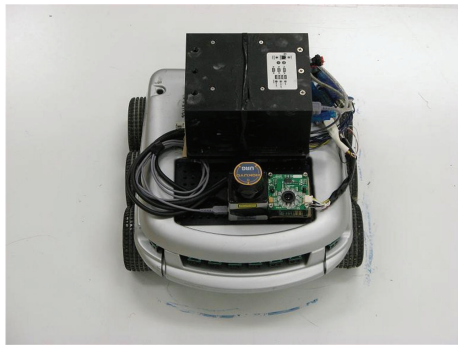


Figure 3: K-team Koala Robot equipped with an ETX-Nano computer for control and a Hokuyo URG laser ranger.

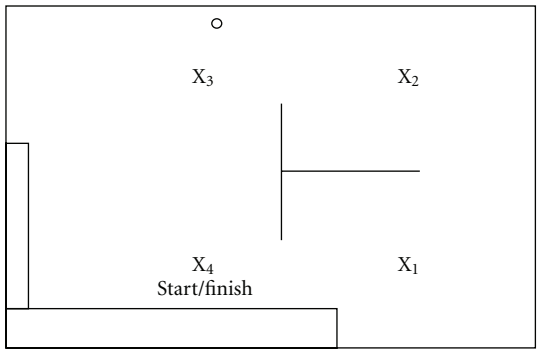


Figure 4: The experiment room (9.6 m × 6.2 m) with four waypoints was equipped with ground truth positioning sensors. The path was chosen such that all behavior tasks could be employed.

If the robot either got stuck on an obstacle or did not reach all waypoints within six minutes, the trial was considered incomplete. Missed deadlines and completeness for each trial was recorded.

6. Experimental Results

Each of the three trials were run using the different behavior periods (shown in Table 2) to determine the effect of timing and determinism on the program's performance. Good performance was based on the following categories: run time, task completion, and distance to the ending position.

Table 3 shows the average number of times the garbage collector ran and the standard deviation for each set using RTTs with the RTGC (RTT/RTGC), RTTs with the non-RTGC (RTT/RegGC), and regular threads with the non-RTGC (RegT/RegGC).

The experiments were broken into two groups for analysis. Group 1 contains the results from the experiments with all behaviors running at the same period. Group 2 contains the results from the experiments where LaserAvoidObstacles was adjusted as well as the set with all tasks running at a rate of 100 ms. Running LaserAvoidObstacles at a shorter period is sufficient since it handles the time consuming safety function of moving around the obstacle.

Table 2: Behavior periods for each experiment set.

Set	Behavior periods (ms)			
	Waypoint Planner (W)	LaserAvoid Obstacles (L)	Occupancy GridMap (Oc)	Dijkstra Planner (P)
1	10	10	10	10
2	100	10	100	100
3	50	50	50	50
4	100	50	100	100
5	100	100	100	100

Table 3: Experimental results showing the average number of times the garbage collector executed along with the standard deviation.

	RTT/RTGC		RTT/RegGC		RegT/RegGC	
Set	Avg	SD	Avg	SD	Avg	SD
1	24.6	0.548	1721	24.515	2770	50.710
2	5	0	514.4	9.940	531	12.247
3	6	0	617.8	12.697	536.4	14.082
4	4.2	0.447	430.2	11.256	368.8	13.971
5	4	0	422.6	8.295	376	40.050

Figures 5 and 6 show the average time that it took the robot to finish the course. Figures 7 and 8 show the average distance that the robot came to the expected finish position. Figures 9 and 10 give the average time it took the robot to travel from waypoint 1 to waypoint 2 (where obstacle avoidance occurred). Figure 11 shows the percentage of missed deadlines for the four behavior tasks for each experiment set. Figure 11(a) shows an overview of all five sets. Since the first set of experiments had such a large percentage of missed deadlines as compared to the others, that set was removed in Figure 11(b) to show more details of the other four sets.

7. Analysis

7.1. Real-Time Threads versus Non-Real-Time Threads. The results show that experiments using RTTs perform similarly or better than experiments that use the regular threads when measuring time to complete the course (Figures 5 and 6). These experiments completed the course faster than those using the regular threads in most cases.

7.2. Comparison of Garbage Collector. We measured the effect that the garbage collector had on the performance in order to analyze determinism. The garbage collector ran less often in the RTT/RTGC experiments (see Table 3). When comparing the trials using only the RTTs, the experiments that utilized the RTGC completed the course faster than those using the regular garbage collector (see Figures 5 and 6). Since the RTGC is deterministic and does not affect the behavior tasks as they execute, we believe that it may have an effect on the average course time.

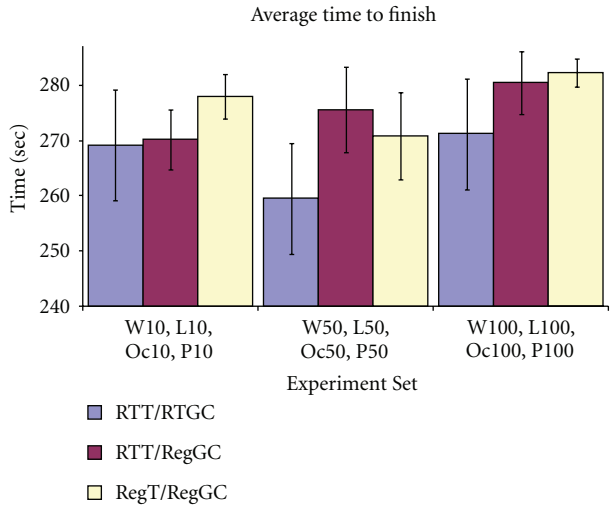


Figure 5: The average time it took the robot to complete the course for group 1.

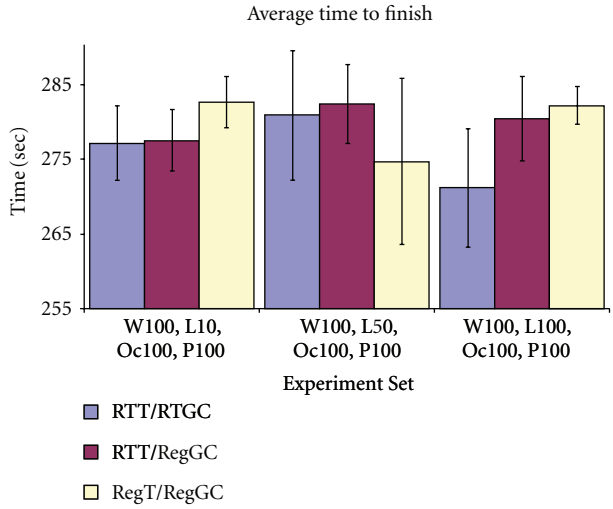


Figure 6: The average time it took the robot to complete the course for group 2.

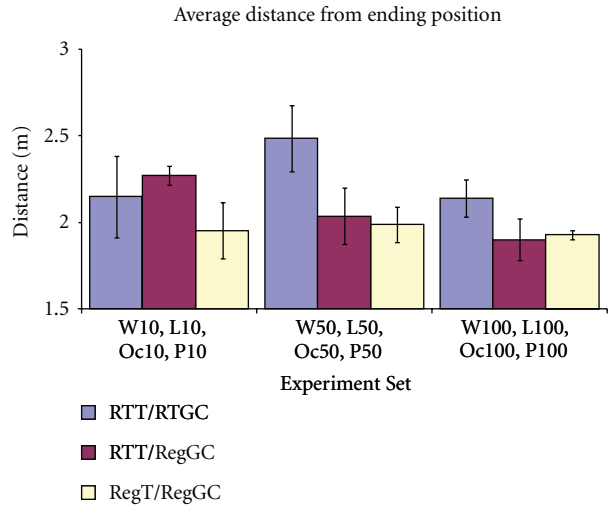


Figure 7: The average distance the robot came to the ending position for group 1.

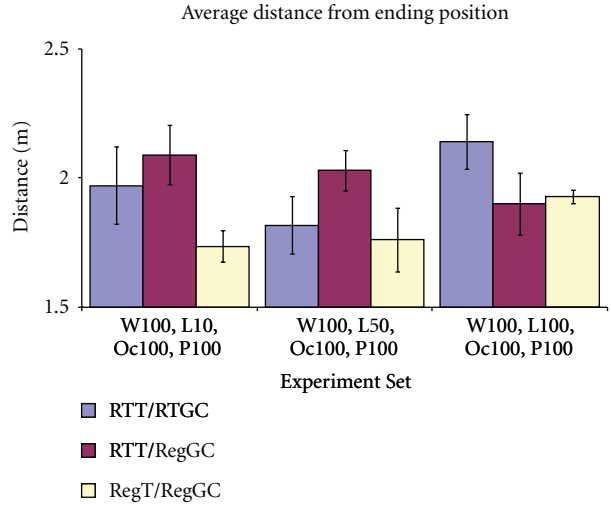


Figure 8: The average distance the robot came to the ending position for group 2.

7.3. Effect of Behavior Frequency. There were statistical differences with the behavior periods when comparing the average distance when the robot came to the ending position for the RTT/RTGC experiments. For instance, in group 1 (Figure 7), running the behaviors at a frequency of 10 ms is statistically different from running at 50 ms. Also, running at a frequency of 100 ms is statistically different from running at 50 ms. In group 2 (Figure 8), running LaserAvoidObstacles at a period of 50 ms and all other behaviors at 100 ms is statistically different from running at a period of 100 ms. These results show that even changes to the frequency of a single behavior can affect performance. This also shows that, if tasks are permitted to run too fast or too slow, it can adversely affect overall performance in form of task completion and accuracy.

Also, to measure the effect that an individual task has on performance, we observed how obstacle avoidance affected the time it took the robot to travel from waypoint 1 to waypoint 2 (see Figures 9 and 10). In all but one case the RTT/RTGC experiments reached waypoint 2 faster than the RTT/RegGC experiments.

7.4. Effect of Latency. One indicator of latency is missed deadlines. The system reports when tasks do not complete within their specified period; however, the tasks are not killed when using the RTTs. Therefore, tasks that take too long to complete may cause other tasks not to have enough resources.

We hypothesized that latency would not be an issue with the lower frequencies because they provide more time for each behavior to complete. From Figure 11(a), it is

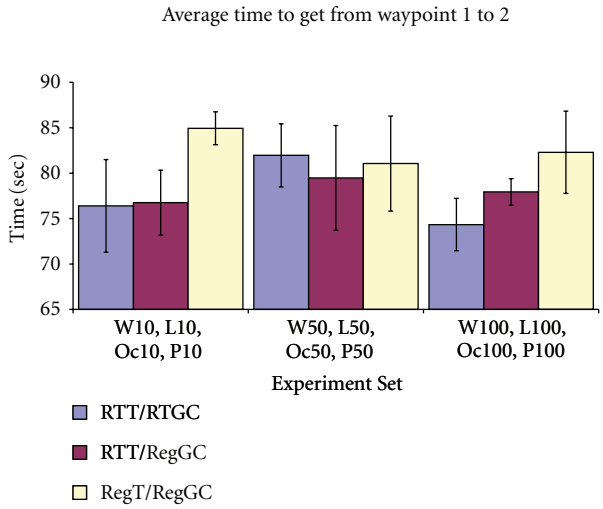


FIGURE 9: The average time it took the robot to travel from waypoint 1 to waypoint 2 for group 1.

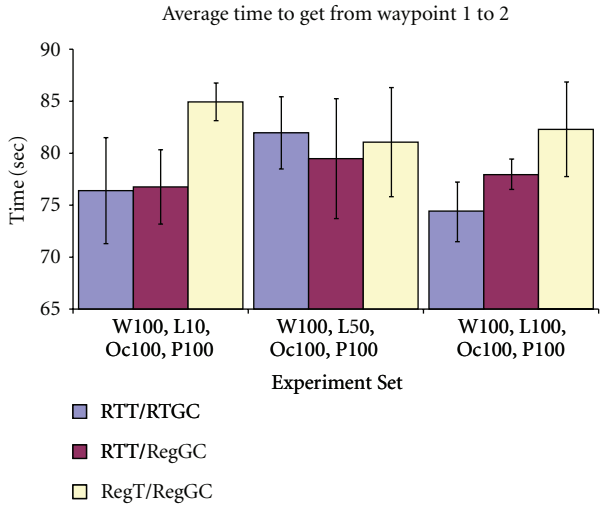
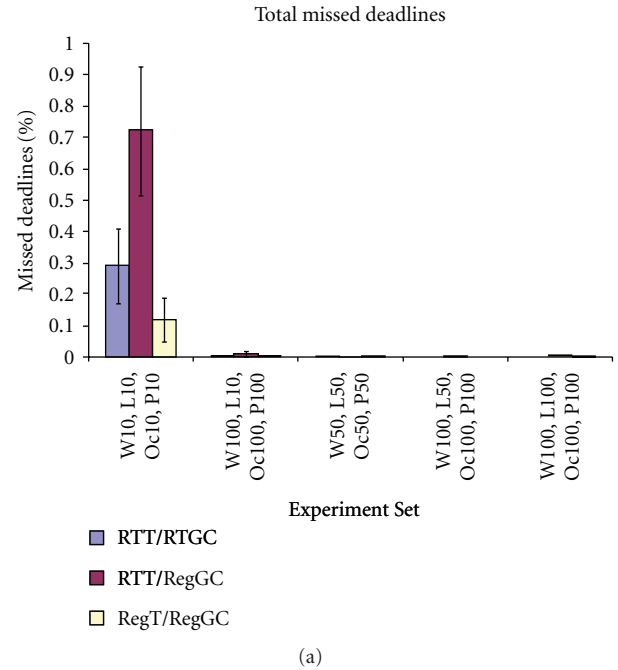
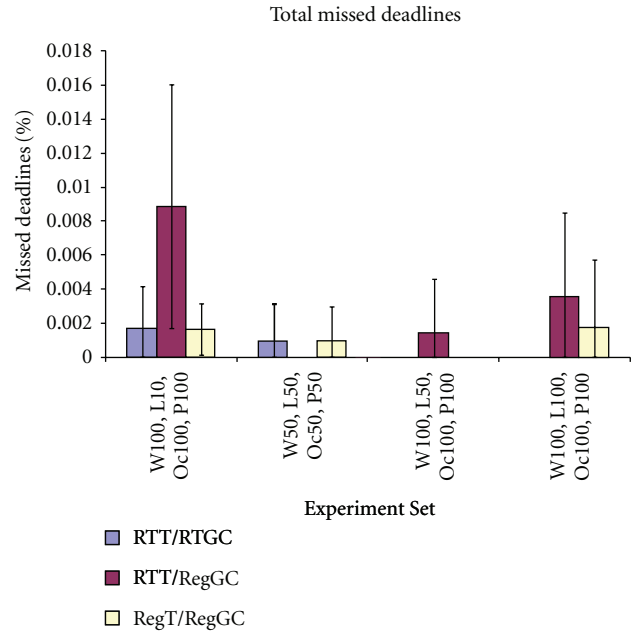


FIGURE 10: The average time it took the robot to travel from waypoint 1 to waypoint 2 for group 2.



(a)



(b)

FIGURE 11: Percentage of missed deadlines (a) for all five experiment sets and (b) for the four experiment sets with a lower missed deadline percentage (rescaled to show more details).

shown that the set with all behaviors running at a period of 10 ms resulted in more missed deadlines. The experiments in this set utilizing RTTs had more missed deadlines than those using regular threads. However, this is not the case when running the behaviors at a period of 100 ms (see Figure 11(b)). In this case, the RTT/RTGC experiments had no missed deadlines.

However, (with the exception of experiment set 3) the RTT/RegGC experiments have a higher percentage of missed deadlines than the RegT/RegGC experiments. Since the regular garbage collector is less deterministic, it is difficult to determine the effect that it has on the task threads.

7.5. Overall Analysis. Results show that proper timing is important to resource utilization even in an unconstrained (resource rich) system. Although latency and determinism are improved in the RTT/RTGC experiments, there is still a difference in how each experiment set performs at different frequencies. Therefore, the frequency of each behavior should be based on input availability.

Results also show that faster cycle times are not a good approach. However, the prevailing approach is to run behaviors as fast as possible, which is not ideal. Slower cycle times would give behaviors more time to finish, but too slow may

also cause system instability, which is consistent with control theory. The key is to find the optimal window at which tasks do not run too short (depleting system resources) or too long (causing slow response). Therefore, the laws of control theory should be applied to robotics requiring all behaviors and their dependencies to be analyzed. This would give developers the chance to define their own frequency requirements that is appropriate for the system that they are using.

8. Conclusion

Many robot architectures view correct frequency as a function of resource availability. By not properly addressing timing concerns, mobile robot systems may not efficiently utilize system resources. Furthermore, ignorance of timing details can lead a developer to produce unstable systems. This is partially because our intuition is that only the critical tasks of a robot control program need to run fast, which is incorrect. One must apply the laws of Control Theory to robotics, requiring all behaviors and their input dependencies to be analyzed instead of just the critical tasks. These laws should be applied because, consistent with control theory, periods that are too short may cause system instability.

We developed a robot software platform to incorporate temporal awareness. The system, written completely in Java using Java RTS, allows individual program components to be assigned an execution frequency. It also utilizes RTTs, non-RTTs, the RTGC, and the traditional garbage collector. We presented the details of the system and its implementation. Experiments demonstrated how manipulating the timing of subtasks and using RTTs with the RTGC affect overall task performance as measured by time to complete the path and the error associated with the ending position. The results of testing validate the hypothesis and show how important proper timing is to resource utilization in mobile robot systems.

In future work, the architecture will be written in a more common real-time language such as C++ using the ORO-COS Real-Time Toolkit [19]. It will then be compared to the Java RTS version. This will be a good measure of Java RTS's performance. Human studies that evaluate programmer proficiency on both Java RTS and C++ may shed light on features of the architecture that are useful in teaching embedded programming techniques.

Acknowledgments

The authors gratefully acknowledge the following NSF Grants: IIS-0846976 and CCF-0829827 for supporting this work. They would also like to thank Nicholas A. Kraft for his knowledge of Solaris.

References

[1] R. A. Brooks, "A robust layered control system for a mobile robot," *IEEE Journal of Robotics and Automation*, vol. 2, no. 1, pp. 14–23, 1986.

[2] K. Konolige, "Saphira robot control architecture," Technical Report, SRI International, Menlo Park, Calif, USA, 2002.

[3] D. S. Touretzky and E. J. Tira-Thompson, "Tekkotsu: a framework for AIBO cognitive robotics," in *Proceedings of the National Conference on Artificial Intelligence*, vol. 4, pp. 1741–1742, AAAI Press, MIT Press, Pittsburgh, Pa, USA, 2005.

[4] B. P. Gerkey, R. T. Vaughan, K. Støy, A. Howard, G. S. Sukhatme, and M. J. Matarić, "Most valuable player: a robot device server for distributed control," in *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS '01)*, vol. 3, pp. 1226–1231, 2001.

[5] G. Beccari, S. Caselli, M. Reggiani, and F. Zanichelli, "Real-time library for the design of hybrid robot control architectures," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 1145–1150, October 1998.

[6] J. Kramer and M. Scheutz, "Development environments for autonomous mobile robots: a survey," *Autonomous Robots*, vol. 22, no. 2, pp. 101–132, 2007.

[7] J. Dorsey, *Continuous and Discrete Control Systems*, McGraw-Hill College, New York, NY, USA, 2001.

[8] S. Bennett, *Real-Time Computer Control: An Introduction*, Prentice Hall International, New York, NY, USA, 1994.

[9] G. Bollella and J. Gosling, "The real-time specification for Java," *Computer*, vol. 33, no. 6, pp. 47–54, 2000.

[10] A. McKenzie, S. Dawson, Q. Alexander, and M. Anderson, "Using real-time awareness to manage performance of java clients on mobile robots," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS '09)*, pp. 3422–3428, December 2009.

[11] A. Nilsson, T. Ekman, and K. Nilsson, "Real Java for real time—gain and pain," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '02)*, pp. 304–311, October 2002.

[12] D. Arora, A. Raghunathan, S. Ravi, and N. K. Jha, "Architectural support for safe software execution on embedded processors," in *Proceedings of the 4th International Conference on Hardware Software Codesign and System Synthesis*, pp. 106–111, October 2006.

[13] G. Bollella, T. Canham, V. Carson et al., "Programming with non-heap memory in the real time specification for Java," in *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '03)*, pp. 361–369, 2003.

[14] S. Smith, S. W. Lawson, and A. Lawson, "Can real-time software engineering be taught to Java programmers?" in *Proceedings of the 17th Conference on Software Engineering Education and Training*, vol. 17, pp. 124–129, 2004.

[15] G. Buttazzo, F. Conticelli, G. Lamastra, and G. Lipari, "Robot control in hard real-time environment," in *Proceedings of the 4th International Workshop on Real-Time Computing Systems and Applications (RTCSA '97)*, pp. 152–159, October 1997.

[16] R. Brega, N. Tomatis, and K. O. Arras, "The need for autonomy and real-time in mobile robotics: a case study of XO/2 and Pygmalion," in *Proceedings of the IEEE International Conference on Intelligent Robots and Systems*, vol. 2, pp. 1422–1427, 2000.

[17] J. Auerbach, D. F. Bacon, D. T. Iercan et al., "Java takes flight: time-portable real-time programming with exotasks," in *Proceedings of the ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '07)*, pp. 51–62, June 2007.

[18] D. B. Stewart, D. E. Schmitz, and P. K. Khosla, "Chimera II real-time operating system for advanced sensor-based

control applications," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 22, no. 6, pp. 1282–1295, 1992.

[19] H. Bruyninckx, P. Soetens, and B. Koninckx, "The real-time motion control core of the OROCOS project," in *Proceedings of the IEEE International Conference on Robotics and Automation*, vol. 2, pp. 2766–2771, September 2003.

[20] H. Bruyninckx, "Open robot control software: the OROCOS project," in *Proceedings of the IEEE International Conference on Robotics and Automation*, vol. 3, pp. 2523–2528, May 2001.

[21] S. G. Robertz, R. Henriksson, K. Nilsson, A. Blomdell, and I. Tarasov, "Using real-time Java for industrial robot control," in *Proceedings of the 5th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES '07)*, pp. 104–110, 2007.

[22] R. A. Brooks, "Intelligence without representation," *Artificial Intelligence*, vol. 47, no. 1–3, pp. 139–159, 1991.

[23] M. Montemerlo, N. Roy, and S. Thrun, "Perspectives on standardization in mobile robot programming: the carnegie mellon navigation (CARMEN) toolkit," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2436–2441, October 2003.

[24] *Measuring Distance Type Obstacle Detection Sensor PBS-03JN Series Instruction Manual*, Hokuyo Automatic, 2002.

[25] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, Mass, USA, 1986.

[26] R. Henriksson, *Scheduling garbage collection in embedded systems*, Ph.D. dissertation, Lund University, 1998.

[27] A. Elfes, "Using occupancy grids for mobile robot perception and navigation," *Computer*, vol. 22, no. 6, pp. 46–57, 1989.

[28] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[29] *Sun Java Real-Time System Precision Control for the Financial Services Market*, Sun Microsystems, 2008.

[30] *Range-Finder Type Laser Scanner URG 04LX Specifications*, Hokuyo Automatic, 2005.