Noname manuscript No.
(will be inserted by the editor)

Real-Time Behavior-Based Robot Control

Brian G. Woolley · Gilbert L. Peterson · Jared T. Kresge

Received: date / Accepted: date

Abstract Behavior-based systems form the basis of autonomous control for many robots, but there is a need to ensure these systems respond in a timly manner. Unexpected latency can adversely affect the quality of an autonomous system's operations, which in turn can affect lives and property in the real-world. A robots ability to detect and handle external events is paramount to providing safe and dependable operation. This paper presents a concurrent version of a behavior-based system called the Real-Time Unified Behavior Framework, which establishes a responsive basis of behavior-based control that does not bind the system developer to any single behavior hierarchy. The concurrent design of the framework is based on modern software engineering principles and only specifies a functional interface for components, leaving the implementation details to the developers. In addition, the individual behaviors are executed by a real-time scheduler, guaranteeing the responsiveness of routines that are critical to the autonomous system's safe operation. Experimental results demonstrate the ability of this approach to provide predictable temporal operation, independent of fluctuations in high-level computational loads.

Keywords General Purpose Real-Time Operating Systems · Behavior-Based Robotics · Reactive Behavior Hierarchies

B.G. Woolley · G.L. Peterson · J.T. Kresge
Air Force Institute of Technology, Department of Electrical and Computer Engineering, 2950 Hobson Way, Bldg 640, Wright-Patterson AFB, OH 45433, USA
Tel.: 937-255-3636, Fax: 937-656-7061
E-mail: brian.woolley@ieee.org, [gilbert.peterson, jared.kresge]@afit.edu

1 Introduction

Autonomous system's operating in the real-world have an inherent requirement to be both robust and responsive to sudden and unpredictable changes in the environment. Typically, reactive behavior-based routines are tasked with maintaining the safe operation of the system. And as demonstrated by YARA [8], the ability of these low-level routines to run at periodic intervals is crucial to the safety and reliability of the robot's operation. The need to make some processes "more important" than others is becoming common in applications where responsiveness is measured in milliseconds. This paper expands on the concepts of behavior-based systems [3], transitioning the sequential execution of the behavior logic in a hierarchy into a multithreaded domain that supports the periodic execution of individual behavior components at independent intervals.

The development of robotic system's that attempt to balance their ability to be both deliberative and responsive in dynamic and changing environments face a difficult problem because simple processes that execute at frequent intervals are interleaved with computationally intensive planning and optimization algorithms that run for relatively long periods. Such situations introduce unpredictable delays where high-priority control routines are forced to wait until lower priority planning elements yield or are preempted by the operating system. This delay can cause detrimental problems when an autonomous system and/or the environment is highly dynamic [2,20], or where processing capability is limited, such as in embedded applications [17,9]. Ideally, deliberative processes execute between the periodic execution of the low-level control routines. Another alternative separates low-level and deliberative tasks between different computers [22], this however be-

comes difficult as operational platforms become smaller and smaller. Further, the amount of computational time available for these high-level processes fluctuates in response to the amount of change in the environment. In chaotic environments a system may operate exclusively under reactive control to maintain a safe operating envelope. In quieter environments reactive control is needed less frequently, allowing deliberative calculations to be performed with the remaining processor time []. The root of the problem is that the schedulers used by modern operating systems do not guarantee that the highest priority process will be the running process, only that the highest priority process will run next.

This paper presents the Real-Time Unified Behavior Framework (RT-UBF) which supports the ability to execute reactive control elements inside a real-time domain. This approach allows the system's behaviors to be scheduled as real-time tasks that can preempt the execution of high-level processes. The RT-UBF is implemented on a Pioneer P2-AT8 robot using RTAI [18] on a standard Linux [26] installation running a three layer robot control architecture. Results comparing real-time and non-real-time execution latency show that the real-time control elements are able to maintain a stable basis of reactive-control with time-critical tasks responding deterministically regardless of the system's computational load. Thus, the execution of individual behaviors are more reliable, as the real-time scheduler ensures each runs in an exact periodic manner.

The following section discusses how real-time scheduling approaches satisfy the needs of the system. As well as presents previous real-time behavior-based systems, from which the design of the RT-UBF builds upon.

2 Background

The responsiveness of a behavior-based controller depends heavily on the deterministic scheduling of the controller components. Several real-time patches and extensions exist to meet these time-critical thresholds, and several robot system exist which employ various real-time capabilities.

2.1 Concurrent Programming

To maintain the responsiveness of a computer, modern operating systems use concurrency to give the appearance that multiple tasks are being handled simultaneously. Concurrency models parallelism by giving each thread a time slice in which to perform a task before being preempted or forced to yield to the next process that is ready to run. This approach attempts to maximize "average" performance [30] and provide the appearance of multitasking. In some cases the overall performance can appear poor because the scheduling algorithm gives no assurance about when a process will run, or that the highest priority task will always be active [28].

Since threads operate within a shared memory space, they are required to synchronize at critical junctions. If not implemented properly, concurrent programs introduce the potential for new problems that do not exist in their sequential counterparts, such as deadlock, interference, and starvation [28].

Currently, modern operating systems and programming languages allow threads to be interrupted synchronously. This approach simplifies the control and synchronization requirements, but is inherently weak because it does not provide a guarantee that the interrupted thread will yield within a known period of time.

2.2 Real-Time Systems

Real-time systems are used to control critical systems where an untimely response to an event in the real-world is either too late or incorrect and risks the safety of the public, personnel, or the system itself [28]. A system is said to be real-time if the correctness of an operation depends not only upon its logical correctness, but also upon the time at which it is performed. Such systems provide control facilities that enable a programmer to specify times at which actions are to be performed or times at which actions are to be completed, as well as the ability to respond or dynamically reschedule tasks when a timing requirement cannot be met. It is also common to distinguish between hard and soft real-time systems. Hard real-time systems typically have a strict schedule in which processes must complete their task, or forfeit the integrity of the system. This approach is typically implemented as an embedded system and guarantees response times less than the maximum stated latency. Approaches that can tolerate some lateness are referred to as soft real-time and are typically responsive but can not assert their maximum latency. The violation of timing constraints in soft real-time systems results in degraded quality, but does not necessarily lead to a failure state.

The advent of the POSIX-1003.1b real-time extensions [16] led to two efforts to develop Linux into a general purpose real-time operating system, RTLinux [30], and RTAI (Real-Time Application Interface) [29].

RTLinux uses an approach known as preemption improvement to shorten interrupt servicing latencies down to levels that support real-time applications [4]. In the preemption improvement approach, the Linux kernel is modified to reduce the length of non-preemptible code in order to minimize the latency of interrupt handling routines or real-time task scheduling in the system [5].

Use of the preemption improvement approach creates several drawbacks. The first is that any guarantee of maximum latency is effectively unverifiable. Although the kernel is generally more preemptible, such a guarantee is limited unless every possible code path in the kernel is examined. Another limitation is that future maintenance is difficult. The preemption improvement approach requires substantial modifications throughout the Linux kernel, which poses the risk of introducing new bugs and makes it unsupportable by the main Linux community [6].

Real-Time Application Interface (RTAI), in contrast to RTLinux, uses an approach known as interrupt abstraction to reduce interrupt latency for real-time applications. Instead of making incremental changes to the kernel to improve its preemptibility, RTAI uses the ADEOS hardware abstraction layer to create a real-time nano-kernel which has higher priority than the Linux kernel [23]. The Linux kernel runs as RTAI's idle process, only running when there are no real-time tasks to run and the kernel is preempted whenever a real-time task needs to run [4,30]. Because a separate hardware handling layer intercepts and manages the actual hardware interrupts, any missed hardware inputs are simulated, making the Linux kernel mostly unaware that it is being subverted by RTAI [6].

The interrupt abstraction approach leaves the Linux kernel largely untouched, avoiding many of the software maintenance problems faced by RT-Linux. Additionally, the RTAI scheduler and hardware abstraction layer total 64 kilobytes, which no longer makes verification of the latency guarantees prohibitive [6]. Additionally, the current version of RTAI provides inter-process communication methods with priority inheritance and a symmetrical API that allows POSIX threads created inside the Linux user-space to be scheduled as real-time tasks, allowing an application to operate using a mixture of real-time and non-real-time tasks [23].

2.3 Real-Time Robot Architectures

The three-layer architecture presents a system that is both deliberative and reactive [13]. However, mobile robots exist in the real world where time and events occur continuously and not in discrete time steps. Despite the concurrent execution of each layer, there are no real-time guarantees that the reactive elements providing for the safe operation of the robot will execute as scheduled. The following discusses current robot architectures that use real-time approaches to enhance responsiveness and ensure safety.

OpenR - Developed as an open architecture (or multi-vendor system) for autonomous robot systems, OpenR is based on Aperios [31], an object-oriented, distributed operating system which allows physical and software components to be defined uniformly as objects. Because everything is referenced as an object, OpenR advocates a common interface for various components like sensors and actuators. Expanding on this approach, the design is a layered model consisting of: a hardware abstraction layer (HAL), a system service layer (SSL), and an application layer (APL) [11]. This approach is intended to allow developers to use well defined interfaces and introduce new programs without affecting adjacent layers. Its major weakness is that the HAL layer provides designated services which are not sufficiently modular, and thus not easily enhanced. Another weakness is that OpenR uses message passing to communicate, causing it to potentially suffer long delays that result from messages setting off a cascade effect resulting in long service periods prior to a task being achieved. Though lauded as a real-time system, this approach fails to enforce real-time constraints on process execution and provides no guarantee that a higher priority process will be given access in a timely manner.

Miro - A CORBA-based robot programming framework [10], Miro is intended to allow for the development of reliable and safe robotic software on heterogeneous computer networks and supports the use of several programming languages. The decision to use CORBA supports a common interface wrapper that allows for distributed processing and platform independent code reuse. However, the overhead that occurs while using CORBA wrappers is not conducive to maintaining the responsiveness required by low-level robot control elements. Although their robot implementation was able to accept and schedule tasks from multiple remote workstations it is unclear how the internal robot control was implemented or how that implementation affects responsiveness of the low-level control elements.

SmartSoft and OROCOS - The goal of the SmartSoft [25] and OROCOS [24] projects is to establish robot control frameworks that are both modular and responsive to events in real-time. The central approach to responsiveness is based on the observer pattern [12] which allows a collection of interested components to be immediately notified of an external event. This ap-

proach achieves good results overall, but it does not limit the length of the code path triggered by an event, and subsequently cannot guarantee that the system remains predictably responsive.

YARA - The YARA architecture [8], which stands for "yet another robot architecture," is unique in that it uses dynamic priority assignment of its active threads to achieve a responsive basis of control in a changing environment. To improve the dependability of the system and ensure a fast response, the priority of each thread is adjusted using an earliest deadline first approach, this helps to achieve a better coexistence of reactive and deliberative components. The soft real-time process scheduler available with Linux versions 2.6 or later gives the next time slice to the highest priority thread waiting to run. This approach demonstrates the ability of a general purpose operating system to provide inter-process communication with an average response time of 175 μs under optimal conditions. The ability of the YARA architecture to remain stable and predictable under an increasing computational load is demonstrated by implementing two edge following behaviors, one in YARA and another in SmartSoft [25], and is capable of producing 786 μs response times between processes. A major problem exposed by this experiment is that execution failures went undetected because the SmartSoft architecture had no internal monitoring mechanism to detect processes that failed to execute as scheduled [8]. By dynamically adjusting the priority of the active processes, the improved Linux scheduler ran the highest priority process in the next time slice, but no guarantees can be made about responsiveness, except that the system has the ability to preempt the running process [1]. The YARA paper closes by suggesting that hard real-time approaches be explored to improve responsiveness and provide guarantees of fine-grained timing intervals.

This paper expands on the goals of the YARA project by presenting a responsive behavior-based controller design that operates as a collection of periodic tasks managed by a hard real-time scheduler.

3 Implementation

This section describes a real-time behavior-based system which can make hard time guarantees for real-world response. First, the high-level component design of the system is presented, followed by an overview of the Unified Behavior Framework (UBF). Next, the Real-Time UBF (RT-UBF) and the modifications made to the robot control architecture to meet hard real-time constraints are discussed.

3.1 High-Level Design

A block diagram of the high-level design is presented in Fig. 1 and shows how RTAI resides directly above the hardware and that the behavior-based controller elements are able to bypass the Linux kernel and be treated as real-time tasks by the RTAI scheduler. The ability to schedule the robot's low-level control routines as periodic real-time tasks is provided by hooks into the RTAI nano-kernel. The RTAI nano-kernel is also responsible for scheduling the Linux kernel which handles the scheduling of all non-time-critical tasks (right hand execution path in Fig. 1).

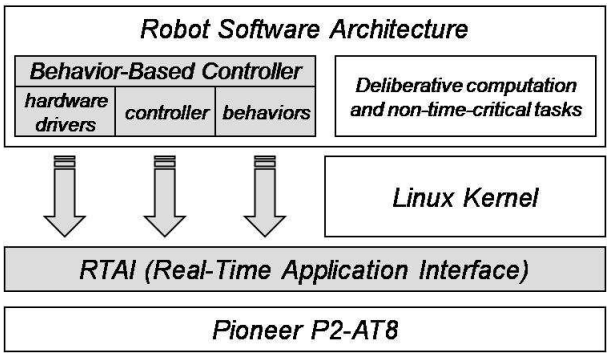

Fig. 1 Real-time tasks bypass Linux and run on the RTAI scheduler.

In this implementation HAMR (Hybrid Architecture for Multiple Robots) [15], a non-real-time three layer robot control architecture, is modified to allow the behavior-based controller's sub-components to be established as hard real-time tasks which have their execution controlled by the RTAI scheduler. By developing the behavior-based controller in this way, the ability to provide a responsive basis of reactive control independent of fluctuations in the system's computational load is accomplished.

The *hardware drivers* and *behaviors* subcomponents of the behavior-based controller shown in Fig. 1 are a library of available drivers and behaviors, while the *controller* is implemented using the RT-UBF.

3.2 Unified Behavior Framework (UBF)

The concept of the UBF is to improve the software development of behavior-based systems though the use of a modular design which attempts to simplify development and testing, promote the reuse of code, support designs that scale easily into large hierarchies while restricting code complexity to base behaviors, and allows

the behavior developer the freedom to use the behavior system they are most familiar with.

In the UBF, all incoming sensor information is stored in the *State*. The central *State* object is a representation of the current environment and includes decoupled sensor data, positional information, goals, and the current operational parameters. The *State* class is implemented using a singleton design pattern to ensure one, and only one instance is created. During execution, a reference to the *State* is passed into the *genAction* method of the abstract *Behavior* class which encapsulates all of the behavioral logic and returns an *Action* object which consists of hardware commands to be applied to the robot (Fig. 2).

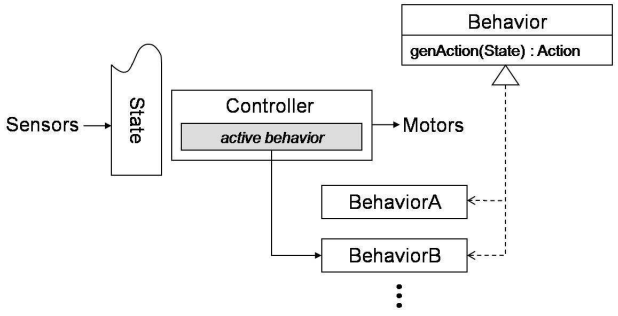

Fig. 2 Strong encapsulation of reactive behaviors allows the controller to change its active behavior during execution.

The UBF uses the strategy pattern [12] to establish an abstract *Behavior* interface. The strategy pattern provides the *Controller* with the proper interfaces to *State* and *Action*, and hides the implementation details of each behavior while allowing the controller to use all of the available behaviors in a uniform manner, making them fully interchangeable. Such an approach frees the low-level controller from being bound to any single behavior architecture. In fact it provides the ability to seamlessly switch between distinct architectures and hierarchies during execution. The class diagram for the UBF is shown in Fig. 3.

In building behavior-based systems with multiple behaviors, the results of the behaviors are fused under a *Composite* behavior, modeled on the composite pattern [12], the *Composite* extends the abstract *Behavior* class with the addition of an *Arbiter*. Through arbitration, the UBF supports reuse by allowing behaviors to be joined in many locations. The *Composite* and *Arbiter* provide the UBF the ability to form hierarchical structures of behavior collections. Thus, a developer is free to reuse the functionality of an existing behavior and incorporate it as part of a new structure. This is of interest because the implementation of substructures

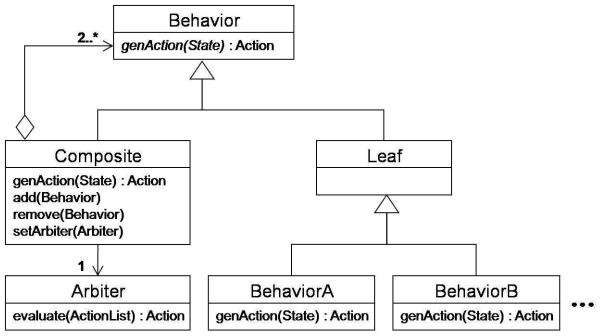

Fig. 3 Class diagram for the Unified Behavior Framework.

used to create a new behavior is not important. This provides a means for hierarchies of disparate architectures to be used in the formation of a new coherent behavior. The Leaf behaviors then perform all of the reactive behavior work.

Following this structure, it is easy to create a variety of behavior heirarchies which execute sequentially to form a desired robot action. A simple behavior hierarchy, shown in Fig. 4, establishes a control structure that includes a goal-seeking behavior and a reactive behavior joined by a highest activation arbiter. The *GoalSeeking* behavior commands the robot to a goal specified in the shared *State*. The *Reactive* behavior provides a means of responding to unexpected changes in the environment. The *HighestActivation* arbiter allows the goal-seeking behavior to yield to the reactive behavior for a period of time in order to respond to the environment. After the period of time expires, the goal-seeking behavior out votes the reactive behavior to make another attempt at the goal.

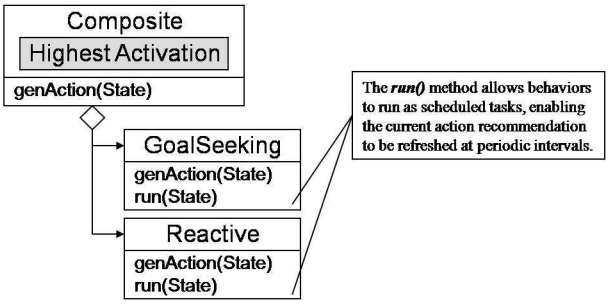

Fig. 4 Behavior structure formed from a goal-seeking element and a reactive element joined by a highest activation arbiter.

At its highest level, a goal directed action recommendation is available via the *genAction* method. When an action recommendation is requested, the composite node sequentially builds a set of action recommendations by calling *genAction* on each of its sub-behaviors.

The set of actions is then evaluated by the arbiter to form a single action that is subsequently returned as the current recommendation.

This sequential action generation approach is modified into a multithreaded one which leverages the advantages provided by deterministic real-time execution.

3.3 Real-Time UBF (RT-UBF)

As demonstrated by YARA [26], the ability of low-level control behaviors to reliably run at periodic intervals is crucial to the safety and reliability of robot's operating in continuous domains that are both dynamic and unpredictable.

This section presents the RT-UBF, an asynchronous implementation of the UBF intended to be employed within robot architectures that leverage concurrent and real-time scheduling. The fundamental change over a sequential approach is that the computational logic of the Leaf behaviors is moved out of the genAction method and into separate threads of execution that run periodically via a real-time scheduler. With each Leaf behavior scheduled to evaluate the environment at appropriate periodic intervals, the current action recommendation can be obtained repeatedly via the genAction method. This approach treats the set of base behaviors as a pool of independent worker threads that execute as concurrent and potentially simultaneous processes.

This change does not affect how a controller requests an action recommendation from its active behavior. In fact, this implementation makes the call to genAction quite fast because it need only traverse the behavior hierarchy to collect and arbitrate the current action recommendations down to a single recommendation. Additionally, this approach divorces the rate at which a controller requests action recommendations from the rate at which each Leaf behavior evaluates the environment.

Using this implementation approach, the structure of a Leaf behavior has two major parts: the genAction method and the run method. The genAction method is a requirement of the abstract behavior class and provides asynchronous access to the behavior's current action recommendation. The run method, called periodically by a scheduling process, evaluates the current environment and updates the current action recommendation.

An asynchronous implementation naturally raises the question, "How frequently should a controller poll its active behavior?" Unfortunately, there is no best answer, but all solutions must consider the level of uncertainty in the current environment. One approach is to request an action recommendation at twice the rate of the fastest periodic evaluation of the environment. This solution is based on the principle of the Nyquist sampling rate [21] and assumes that the periodic schedules of the base behaviors are adjusted at runtime to match the environment's current level of change. In rapidly changing environments, this approach allows low-level processes to execute at shorter periodic intervals, increasing the computational time used by reactive control routines. In more stable environments, the scheduler can set low-level processes to execute less frequently, making computational time available to higher-level planning processes.

By converting the UBF into disparate asynchronous behaviors, the ability to sample the environment and recommend actions in a deterministic periodic manner creates a highly reactive basis of control for behavior-based architectures.

3.3.1 RT-UBF Addition Into HAMR

The HAMR robot control architecture [15] (Fig. 5) interfaces to the physical robot hardware via the Player robot control suite [14]. To make HAMR's behavior-based controller operate in real-time, the hardware drivers required by the controller (*Sensors* and *Robot Command*) are divorced from the existing Player [14] implementation and modified into priority-based hard real-time tasks. This allows them to preempt the Linux kernel when they enter the run state and register with the RTAI hardware abstraction layer to interface with the hardware components in real-time. Second, the hardware manager, a subsystem of HAMR's *Sequencer* is modified to interface to the modified RTAI schedulable hardware drivers. Third, the Leaf behavior classes in the UBF are modified for real-time execution, this converts all behaviors into real-time tasks. Fourth, the *Controller*, which is responsible for arbitrating the behavior hierarchy to generate a final action is written for real-time periodic execution.

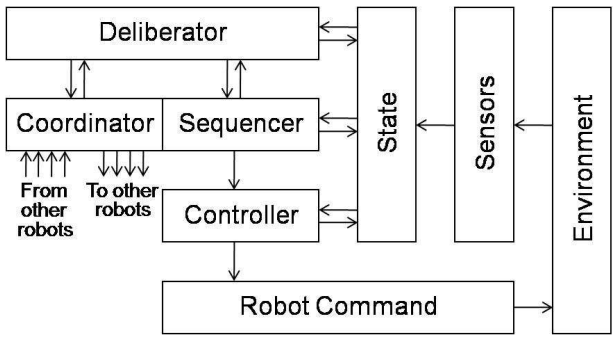

Fig. 5 The deliberative subsystem of HAMR will be preempted by the low-level controller and sensor subsystem.

These changes allow the RT-UBF to operate it's control loop within the HAMR architecture. The behavior-based low-level control loop consists of the three-step process presented in Fig. 6. First, the *State* is updated by the *Robot Driver* to represent the current conditions of the environment. Second, the *Controller* pulls the final action from the *Active Behavior*. Finally, the *Controller* gives the proposed *Action* the authority to issue motor commands to the *Robot Driver*, closing the sense/act low-level control loop.

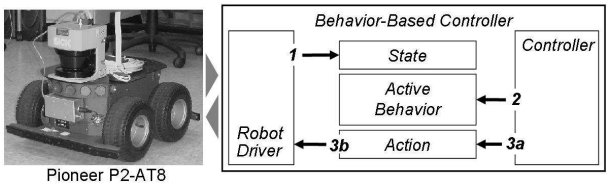

Fig. 6 The RT-UBF low-level control loop is established as a three step process (1) update the state; (2) generate an action recommendation; (3) authorize the action to enact motor commands on the robot.

Under this RT-UBF controller design, the set of behaviors assumes that the *State* is representative of the current environment. This assumption places the responsibility of keeping the system in sync with the real-world on the *Robot Driver*, because it updates the *State* with the current sensor data. The ability to establish a driver as a real-time task allows its routines to execute at predictable intervals driven by the sensors update rate, which in turn ensures that the central *State* object is updated at regular intervals. Each piece of hardware (i.e. robot, laser, and camera) operates as a separate real-time task, and each updates its associated *State* data members individually.

The next responsibility of the behavior-based controller is for each behavior to generate an action recommendation based on the current conditions of the environment. Each behavior does so by using the sensor data contained in the *State*, and assumes that it contains an accurate representation of the current environment. The interval at which each individual behavior generates an action depends on how time-critical the behavior is perceived to be, and on how long the behavior algorithm takes to execute.

The *Controller*, like the behaviors and the sensor drivers, is established as a real-time task, allowing a final action to be generated at predictable intervals. The *Controller* keeps an active behavior without knowing about its implementation. Rather than executing this three-step process sequentially, the *State* and the behaviors are updated asynchronously. Thus, the *Controller* enacts a two-step periodic process that first requests an action recommendation from the *Active Behavior*, which is typically a hierarchy of behaviors, and then authorizes the final action to enact the recommended motor commands on the robot.

The State and Action classes are introduced as generic interfaces to the *Robot Driver*. Explicitly missing from the *State* are methods that access the *Robot Driver*'s motor command interface. This capability is embedded in the execute method of the Action class, and requires a reference to the *Robot Driver*. This requirement ensures the coordinated operation of the robot by allowing the *Controller* to enact the final action recommendation returned by the *Active Behavior* on the robot. The bifurcation of the *Robot Driver* into two interfaces allows the RT-UBF to make information about the robot's current state widely available and protect against behaviors hat may act unilaterally on the robot.

The ability to regularly update and evaluate the environment allows the robot to operate in a safe and dependable manner by remaining responsive to changes in the environment.

4 Experiment

The latency experienced by time-critical tasks, as scheduled by, the native Linux kernel, and the RTAI real-time nano-kernel, both in user-space, and while operating a Pioneer P2-AT8 robot are compared. The robot, a four-wheeled robotic platform equipped with 16 sonars, odometry, a SICK LMS200 laser scanner, and a firewire camera. The robots task during this test was to be led around a hallway by following an orange cone, while avoiding stationary and moving obstacles.

There are seven periodic real-time tasks, given in Table 1. The *rt-ubf* controller is setup to arbitrate between two behaviors. The first, the *follow* behavior, follows an orange cone by using blob data provided by feeding camera images through a blob detection algorithm. The second, the *obs_avoid* behavior, uses laser scan data to steer the robot around detected obstacles. Inputs to the *State* are provided by the *robot*, *laser*, and *camera* hardware drivers. The *blobfinder* pulls the latest image the *camera* driver wrote to the *State*, processes it using it's blob detection algorithm, and writes the blob results to the *State*.

Since each task is scheduled periodically, harmonics exist that require some tasks to run at exactly the same time. To reduce unnecessary latency due to scheduling collisions, the initial execution of each thread is staggered in time by some offset which interleaves their execution.

The hardware drivers provide a central service by ensuring that the perceived *State* correctly represents

Table 1 Periodic Real-Time Task Scheduling Configuration

Task	Period	Duration	Offset	Priority
robot	25 ms	50 μs	0 ms	1
laser	25 ms	50 μs	1 ms	1
camera	100 ms	1 ms	2 ms	1
blobfinder	100 ms	3 ms	4 ms	1
follow	250 ms	10 μs	8 ms	2
obs_avoid	100 ms	50 μs	9 ms	2
rt-ubf	50 ms	10 μs	10 ms	3
Linux	Idle			9999

the current environment. If the *State* falls out of sync with the real-world, the remaining controller components become ineffective, consequently the hardware drivers hold the highest priority. The elemental behaviors are given the next highest priority because their evaluations of the *State* form the basis for what actions the controller's active behavior will recommend at any given time. The controller holds the third highest priority to ensure the final arbitrated action gets executed by the *robot* driver.

The Linux user-space computational load for this experiment is provided by the deliberative part of HAMR as well as a SLAM algorithm. The deliberative part of HAMR is composed of numerous threads that make up the deliberator, sequencer, and coordinator subcomponents [15]. The SLAM processing uses a FastSLAM [19] algorithm running 1000 particles, which provides a substantial processing burden to the robot's Intel Pentium M 1.6 GHz CPU.

The seven routines that form the behavior-based controller are instrumented to capture the current time, and calculate the latency experienced per execution period. Latency is measured as the time between when a periodic task is scheduled to execute, and when it actually begins executing. For example, if a task is scheduled to execute every 20 ms and the difference between the previous start time and the current start time is 22 ms, the reported latency is 2 ms. Process jitter is determined by evaluating the difference in latency measurements over time.

The real-time tasks are set to run in periodic mode, as opposed to one-shot mode. In periodic mode, the 8254 timer is used to create a real-time clock period which generates interrupts to signal the RTAI scheduler to run the next available task. This limits what periods the various tasks can be set to since each period must be a multiple of the real-time clock period. However, it provides consistent and reliable thread scheduling, where one-shot mode uses the microprocessor's clock to schedule tasks at any period, but with much less reliability.

5 Results

The results of this experiment demonstrate that regardless of the computational load, a responsive basis of control is attained by implementing time-critical routines as real-time tasks. The latency measurements achieved by this experiment far exceeds the 100 μs hard real-time guarantee provided by the RTAI documentation. The empirical results of this experiment indicate that the periodic scheduler executes tasks exactly on time, ± 1 count of the 8254 timer, as long as a few considerations are followed. First, each thread period must be set at a multiple of the 8254 base period, i.e. if a 20 ms period is desired, but the 8254 base period is 438 μs then the closest available period is 19.80 ms or 20.28 ms. Second, if multiple periodic threads are to be executed, careful planning of how frequently each runs, how much processing time each requires, and when each task is started will help to avoid scheduling collisions. If scheduling is not planned carefully, a significant amount of latency could be introduced into the real-time tasks, which in turn could affect responsiveness.

Fig. 7 compares the latency over a 10 second period of time for non-real-time Linux user-space threads, and real-time RTAI scheduled *laser* driver, *obs_avoid* behavior, *rt-ubf* controller, and *robot* driver tasks. The four trends show the processes required to detect and respond to an obstacle. First, the *laser* driver must load the latest scan data into the *State*. Second, the *obs_avoid* behavior must use the laser scan to generate an action recommendation that out votes the *follow* behavior. Third, the *rt-ubf* controller must arbitrate the behaviors and generate a final action. Fourth, the *robot* driver must respond to the action by issuing motor commands.

The latency measurements taken for each task indicate that time-critical routines can be scheduled to execute at predictable intervals by removing them from the context of the Linux environment and running them as real-time tasks. However, the robot used for this experiment was only capable of low dynamics, therefore the measured latency was never found to be detrimental to operation. But, the ability to guarantee reactive control through the use of the deterministic scheduling provided by the RT-UBF is especially important for autonomous system's where throwing more CPUs at the problem is not feasible. Such as in embedded system's, where power and size are limited, which are becoming prevalent in micro unmanned vehicles.

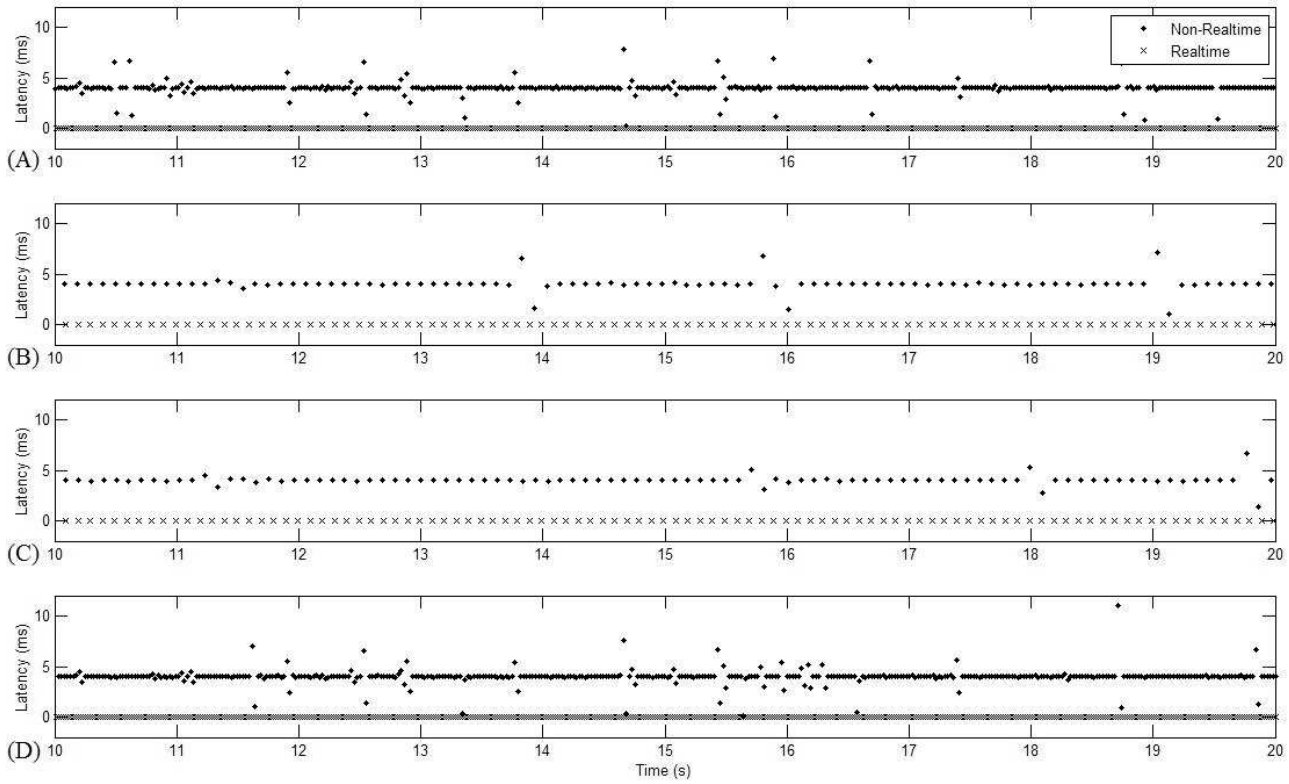

Fig. 7 Observed latency for (a) the *laser* driver; (b) the *obs_avoid* behavior; (c) the *rt-ubf* controller; and (d) the *robot* driver.

6 Conclusions

Mobile robot architectures are a mixture of interconnected processes working to achieve specific results. As the dynamics of the application platform increases, or as the complexity of the operational environment increases, the responsiveness of behavior-based system's need to keep up. Also, as autonomous system's have evolved into the micro domain, where processing ability is limited, there is a need to ensure the responsiveness of behavior-based systems.

This paper presents the Real-Time Unified Behavior Framework (RT-UBF) behavior-based controller, and compares the responsiveness of a non-real-time versus a real-time version using a hardware-based experiment. This experiment demonstrates that the ability to establish low-level control routines as real-time tasks is an effective approach to ensuring that a mobile robot can remain responsive to sudden and unpredictable changes in the environment. RTAI provides the services to make some processes "more important" by moving time-critical routines out of the Linux environment and into an environment managed by a real-time scheduler. The real-time UBF is built on top of these services to ensure that the robot's reactive controller remains responsive to changes in the environment.

The next logical question is, "How many real-time tasks can be supported by this approach?" Like YARA [26], this experiment focuses on allowing low-level control routines to remain predictably responsive to changes in the environment while sharing a single processing resource with computationally intensive routines. Although isolated from the effects of unpredictable fluctuations in a system's computational load, the ability of a system to remain predictably responsive requires that the real-time domain behaviors identified as time-critical be managed as real-time components and do not jeopardize the system's operational requirements.

Acknowledgements This work was supported in part through Lab Task #08RY10COR from the Air Force Office of Scientific Research, USAF. The views expressed in this article are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U.S. Government.

References

1. J. Aas. "Understanding the Linux 2.6. 8.1 CPU Scheduler." *Retrieved Oct*, vol. 16, 2005.
2. J.S. Albus. "Features of Intelligence Required by Unmanned Ground Vehicles." *Proc. of Performance Metrics for Intelligent Systems Workshop*, 2000 PerMIS Workshop, 2000.

3. R.C. Arkin. *Behavior-Based Robotics*. Cambridge, MA: MIT Press, 1998.
4. M. Barabanov. "A Linux-Based Real-Time Operating System." New Mexico Institute of Mining and Technology, 1997.
5. M. Barabanov and V. Yodaiken. "Introducing Real-Time Linux." *Linux Journal*, vol. 34, pp. 19-23, 1997.
6. T. Bird. "Comparing Two Approaches to Real-Time Linux." *CTO of Lineo*, 2000.
7. J. Bloch. *Effective Java: Programming Language Guide*. Boston, MA: Addison-Wesley, 2001.
8. S. Caselli, F. Monica, and M. Reggiani. "YARA: A Software Framework Enhancing Service Robot Dependability." *Robotics and Automation*, 2005. ICRA 2005. *Proceedings of the 2005 IEEE International Conference*, pp. 1970-1976, 2005.
9. M. Dong, B.M. Chen, G. Cai, and K. Peng. "Development of a Real-time Onboard and Ground Station Software System for a UAV Helicopter." *AIAA J. Aerosp. Comput., Inf., Commun*, vol. 4, pp. 933-955, 2007.
10. S. Enderle, H. Utz, S. Sablatnog, G. Kraetzschmar, and G. Palm. "Miro: Middleware for Autonomous Mobile Robots." *Robotics and Automation*, *IEEE Transactions*, vol. 18, pp. 493-497, 2002.
11. M. Fujita and K. Kageyama. "An Open Architecture for Robot Entertainment." *Proceedings from the First International Conference on Autonomous Agents*, pp. 435-442, 1997.
12. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Boston, MA: Addison-Wesley, 1994.
13. E. Gat. Three-layer architectures. In: D. Kortenkamp, R.P. Bonasso, and R. Murphy (Eds.), *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*, AAAI Press, Menlo Park, CA/MIT Press, Cambridge, MA, pp. 195210, 1998.
14. B. Gerkey, R.T. Vaughan, and A. Howard. "The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems." *Proceedings of the 11th International Conference on Advanced Robotics*, pp. 317323, 2003.
15. D. Hooper and G.L. Peterson. "HAMR: A Hybrid Multi-Robot Control Architecture." *The 22nd International FLAIRS Conference*, Sanibel Island, FL, pp. 139-140, May 2009.
16. IEEE. *Portable Operating System Interface (POSIX)*: IEEE/ANSI Std 1003.1, 1996.
17. A. Kurdila, M. Nechyba, R. Prazenica, W. Dahmen, P. Binev, R. DeVore, and R. Sharpley. "Vision-Based Control of Micro-Air-Vehicles: Progress and Problems in Estimation." *43rd IEEE Conference on Decision and Control*, Paradise Island, Bahamas, December 2004.
18. P. Mantegazza, E.L. Dozio, and S. Papacharalambous. "RTAI: Real-Time Application Interface." *Linux Journal*, vol. 2000, 2000.
19. M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit. "FastSLAM: A Factored Solution to the Simultaneous Localization and Mapping Problem." In *Proceedings of the AAAI*, pp. 593-598, 2002.
20. W.S. Newman. "Team Case and the 2007 DARPA Urban Challenge." *http://urbanchallenge.case.edu*, June 2007.
21. H. Nyquist. "Certain Topics in Telegraph Transmission Theory." *Proceedings of the IEEE*, vol. 90, 2002.
22. M. Quigley, E. Berger, and A. Y. Ng. "STAIR: Hardware and Software Architecture." In AAAI 2007 Robotics Workshop, Vancouver, B.C, August, 2007.
23. P. Sarolahti. "Real-Time Application Interface." Technical Report, University of Helsinki, Dept. of Comp. Sci., 2001.
24. C. Schlegel. "Communicaions Patterns for OROCOS. Hints, Remarks, Specifications." Technical Report, Research Institute for Applied Knowledge Processing (FAW), 2002.
25. C. Schlegel and R. Worz. "The Software Framework SMARTSOFT for Implementing Sensorimotor Systems." *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'99)*, pp. 1610-1616, 1999.
26. M. Shuttleworth. "Ubuntu: Linux for human beings." *http://www.ubuntu.com/*, 2006.
27. W. Stallings. *Operating Systems*. Third ed., Upper Saddle River, New Jersey 07458: Prentice Hall, 1998.
28. A.J. Wellings. *Concurrent and Real-Time Programming in Java*. West Sussex PO19 8SQ, England: John Wiley & Sons, Ltd, 2004.
29. K. Yaghmour. "The Real-Time Application Interface." *Proceedings of the Linux Symposium*, July, 2001.
30. V. Yodaiken and M. Barabanov. "A Real-Time Linux." *Proceedings of the Linux Applications Development and Deployment Conference (USELINUX)*, January 1997.
31. Y. Yokote. "The Apertos Reflective Operating System: The Concept and Its Implementation." *ACM SIGPLAN Notices*, vol. 27, pp. 414-434, 1992.