AIAA-94-1288-CP

A SOFTWARE ARCHITECTURE FOR HARD REAL-TIME EXECUTION OF AUTOMATICALLY SYNTHESIZED PLANS OR CONTROL LAWS

Marcel Schoppers
Robotics Research Harvesting
166 Springdale Way, Redwood City, CA 94062

Abstract

We present a hard real-time software architecture which enables two kinds of safety within a single system: the safety of flexibility and robustness, and the safety of guaranteed timing. The architecture combines a) online intelligent synthesis of courses of action, with b) precise timing and very high speeds in the performance of such automatically synthesized plans. A fundamental component is a negative feedback loop from available computing resources to computational demands, which ensures both that online decisions are feasibly performable, and that the available computing resources are used to best advantage. This feedback loop allows the architecture to respond to degraded sensor systems by scheduling alternative computations without accidentally impairing the timing of other tasks; allows to respond to degraded computers by scaling back system activities to maintain minimal standards of performance, e.g. safety; and allows to dynamically exploit excess computing power for improved performance.

The architecture, though still subject to limits on the achievable robustness, can nevertheless be expected to out-perform systems in which computer power is allocated off-line. We briefly describe two robotic subsystems which can not be built safely without our architecture.[1]

1 Motivation

1.1 Two Control System Problems

Some NASA applications, such as the intelligent management of hardware faults in Space Stations and spacecraft, and the use of intelligent mechanisms (e.g. robots) about the Space Station and on Mars, require both *flexibility* and *hard real-time* execution. Two cases in point are the intelligent control of "redundant" robots and the dynamic reconfiguration of perception processing, both of which we are currently implementing.

With only 6 degrees of freedom a robot arm can reach any posture in exactly one way, and this simplicity makes mathematical motion analysis easy. The NASA/JSC robot called the Extra-Vehicular Activity Helper/Retriever (EVAHR) has 20 degrees of freedom: 7 in each of the two arms, and 6 in the body. As a result there are an infinite number of ways for the EVAHR to get one hand to a particular posture. The extra ("redundant") degrees of freedom greatly complicate the motion control problem. Our approach to the control of such a "redundant" robot takes dynamical limits into account, validates the robot's expected motion, and also allows it to start moving immediately. This is accomplished as follows.

(1) Our Artificial Intelligence (AI) software uses a qualitative kinematic model to choose a sequence of postures for the robot to achieve [Jung and Badler, 1992]. This sequence of postures deals with robotic "redundancy" in roughly the way humans would: to reach under a bed, one gets down on one's knees, rests on all fours, lowers one's head, and stretches out one arm. Each intermediate posture is then treated as an attractive potential field. The attractive fields, together with their repulsive counterparts (obstacles) drive motion dynamics in the usual way (after [Khatib, 1986]) thus striving to avoid collisions and ensuring that the motions are compable with the robot's force and torque limitations. A super-real-time motion previewer then uses these potentials to preview the resulting motion before it is performed, and so verifies that the motion will avoid both collisions and potential wells (local minima).

(2) For this scheme to work, there must be a careful synchronization between trajectory previewing and the actual motion, lest the actual motion get into space that has not been previewed, with potentially dangerous consequences. Indeed, for given motion speeds and accelerations the previewer must be executed with a frequency lying within a limited band [Schoppers, 1993]: if the previewer is executed too frequently it won't have time to preview a fail-safe trajectory; if it is executed too infrequently the robot can accelerate ahead of it.

The preceding two paragraphs taken together signal trouble. Paragraph (1) says that we need an Artificial Intelligence (AI) program to decide, on-line, how the robot should move, by choosing a sequence of postures. Paragraph (2) says that motion-related computations must

[1]Copyright ©1993 by Marcel Schoppers. Published by the American Institute of Aeronautics and Astronautics Inc. with permission.

be carefully timed. These two requirements have been incompatible for many years.

(3) Our second robotic subsystem provides a new degree of system robustness and survivability for robotic vision. We take advantage of the presence of several types of vision sensors on the EVAHR robot by dynamically selecting available sensors and perception algorithms, thus improving the probability of system survival despite sensor malfunctions.

(4) Robotic vision is part of the pipeline from sensor input through state estimation to motion control. Since data reaches the robot's effectors some time after the data was first sensed, the control system must compensate for the delay by predicting where things *will* be when the intended actions are finally performed or completed. If any part of the sensor-input-to-action-output pipeline is mis-timed, the control system's predictions will be wrong and damage may result.

Here too, the combined implications of paragraphs (3) and (4) are problematic, but for a different reason. Perception can be implemented without resorting to AI algorithms; but because the computing time required to interpret different sensors differs widely, dynamic reconfiguration effectively makes the computing time of the whole perception system unpredictable. Once again, the conflict between system flexibility and real-time execution is holding up an otherwise good idea.

1.2 The Core Problem

It is a very difficult problem to implement systems that are at once *flexible* in the sense of adaptivity and robustness, and *hard real-time* in the sense of timely execution of actions. To show why this combinations of requirements is so difficult, this subsection presents a careful analysis of each requirement.

"Flexibility" in a machine is as apparent as its makeup is subtle. The dark background against which flexibility stands out in sharp relief is the problem of domain complexity. We explain this with reference to automated diagnosis. In principle, diagnosis could be done by compiling a large decision table that associated combinations of symptoms with faults. As the number of possible faults and symptoms grows, however, the number of possible *combinations* of faults (and symptoms) grows astronomically. The programmer is inevitably overwhelmed by the complexity of the domain, and so fails to anticipate some of the possibilities and to properly understand others. The result is software that makes false assumptions and sometimes behaves inappropriately. Flexibility, then, stands out as the ability to go on behaving appropriately even when the number of possible situations grows beyond the reach of human forethought.

The fundamental justification for Artificial Intelligence (AI) algorithms is their flexibility as just defined. Flexibility may be necessary for many reasons, including: the environment may be unpredictable, or subsystems may malfunction, or the environment may be entirely predictable but very complex. In such cases, AI algorithms can do *on-line* the problem solving that could (in principle but not in practice) have been done by a programmer off-line. Not only are AI algorithms more economical in terms of programming effort, they are also safer because the responses devised by the AI software can take dynamically occurring factors into account more thoroughly, and the resulting system is likely to generate appropriate behavior for a much wider variety of the possible situations.

We now turn to the requirement of real-time execution. "Hard real-time" execution is necessary whenever hardware must be controlled safely. In general, a computation is called "real-time" when the correctness or other value of a computation depends not only on what data or action the computation produces, but also on the time at which that data or action is produced. For example, there is a particular moment in time beyond which any computing about collision avoidance is no longer useful because a collision is no longer avoidable. More often than not, the timing of computations is a critical factor in total system safety. The field of hard real-time computing research is working on ways of *guaranteeing* that the timing of computations is correct.

It would be nice if we could build systems that were both safe on account of the flexibility built into them with AI algorithms, and doubly safe because their computations were guaranteed to be completed on time with the available computing power. But this conjunction is a difficult one. The pivotal problem was first clearly described by [Paul et al, 1991] as follows:

> The timing of actions taken by a real-time system must have low variance, so that the effects of those actions on unfolding processes can be predicted with sufficient accuracy. But intelligent software reserves the option of extended searching, which has very high variance.

Because AI algorithms — like people solving difficult problems — generally reach conclusions by making plausible guesses that may turn out to be wrong any number of times ("searching"), their total execution time is highly unpredictable. To make matters even worse, AI software in control applications often dynamically varies the tasks being carried out, thus dynamically changing the structure of the computation. When there are n computations a system could execute, there are 2^n possible subsets, each with its own execution time, so that again the total system's computational load is highly variable. Both sources of unpredictability make it very difficult to be sure that the total system can perform in hard real-time.

Thus it was necessary, until recently, to choose between the safety provided by hard real-time performance, and the safety provided by on-line automated decision making. Systems such as intelligent robots or space stations, which require both, were specifically beyond the state of the art.

1.3 A General Solution

We have undertaken to provide both kinds of safety simultaneously by means of a novel software architecture. Our architecture, which has been worked out in detail but not yet implemented, exploits a specific set of hard-real-time techniques that have become available only in 1992 and 1993.

Previous attempts by others to integrate hard real-time performance and on-line automated decision making were of two kinds. One tried to force AI software into a real-time mold by limiting on-line searching, thus also eliminating most of the flexibility. The other allowed the AI software to compute as long as it liked, and tried to design the real-time part of the system to keep the whole system out of trouble until (indefinitely much later) the AI software communicated a decision to the real-time subsystem. This allowed flexible behavior until the real-time part of the system stumbled into unfamiliar territory with the AI software still thinking about the past.

Our approach is an elaboration of the CIRCA architecture [Musliner et al, 1993]. The AI software doesn't merely send new parameters to a fixed real-time subsystem, it dynamically reprograms the whole real-time subsystem, and *simultaneously* plans total system behavior to ensure that the real-time program will be able to cope with the *chosen* future. This produces such interesting phenomena as robots slowing down to ensure that their real-time programs will not be overloaded — the AI software now can shelter the real-time subsystem and can also buy itself time to think. In short, our approach imposes a negative feedback loop from excessive computational loads to less demanding system behavior. Further, the real-time subsystem is no longer cast in stone but can be made to adapt to its context, and to perform procedures that were automatically constructed.

We were careful to design our software architecture in an application independent way, so that the solution would be suitable for use in everything from robots, to space stations, to interplanetary spacecraft, to Lunar and planetary bases — with a promise of enhanced safety in all cases.

Having resolved the central flexibility/real-time conflict, we intend to demonstrate the great value of our software architecture by having it enable two novel applications, namely the dynamical control of robots with many degrees of freedom, and the dynamic reconfiguration of a multi-sensor fusion subsystem. Our subsystem for reconfigurable multi-sensor fusion will dynamically choose whatever sensors and perception algorithms it likes, and the robot's physical behavior will adjust to the dynamically changing computing load. Our subsystem for intelligent control of the motion dynamics of robots with many degrees of freedom will be a large advance on current robot control technology. It will rely on our general software architecture for proper timing of motion previewing relative to actual motion, and for hard real-time computing in the presence of AI posture planning algorithms. Both of these applications are impossible to implement safely without our software architecture. In enabling feedback from computing load to safe behavior, our software architecture will automatically and dynamically determine the maximum speed at which a robot can safely move, even when some of the robot's computers and/or sensors are malfunctioning or disabled. As a result our architecture will also support sensor processing reconfiguration in a completely safe and general way. Both modules will demonstrate the value of our architecture for improved adaptiveness and survivability of complex control systems.

1.4 Our Approach In Perspective

The Artificial Intelligence (AI) community has come to address the requirements of real-time systems in three ways [Durfee, 1990]. When building a system that must act in real-time as well as reasoning, one can:

- Subject the AI component of the system to hard deadlines (*e.g.* anytime algorithms). Under time pressure, this results in truncation of intelligent function.
- Allow the AI component to think freely, make the real-time subsystem responsible for total system safety, and have the AI component re-parameterize the real-time subsystem with whatever guidance the AI subsystem can produce in time. Under time pressure, this results in intelligent function being left far behind the rush of events.
- Refuse to subject the AI component of the system to hard deadlines, but let the AI component dynamically reprogram the real-time subsystem with a program realizing *a discrete-event control law that preserves closed-loop stability with sufficient robustness for the period of time in which the AI subsystem is deciding what to do next.* This approach remains functional even under time pressure — almost by definition.

We regard the NASREM architecture [Albus et al, 1987] as embodying the first approach, and the architectures of Bonasso [Bonasso, 1991; Bonasso and Slack, 1992] and Gat [Gat, 1992] as embodying the second. We favor the third, in which we have imposed control-theoretic criteria upon the ideas of [Musliner et al, 1993]. The resulting architecture has the following advantages:

1. the AI software can remain intelligent and flexible;
2. the real-time subsystem need not be programmed (at system design time!) to be competent against all possible contingencies;
3. the AI software can reprogram the real-time subsystem to prevent computing overloads before they happen (e.g. with slower motion), thus pro-actively buying itself time to think;
4. the program down-loaded into the real-time subsystem can be as flexible as the AI software can make it (e.g. in a robotic application the maximum safe speed can now be a function of computational load, and so will be higher than the worst-case limit nearly all the time, allowing substantial performance gains);
5. the real-time subsystem can be a small operating system, time-slicing tasks or threads with widely differing frequencies, in contrast to the many mobile robot controllers in which all computations are executed with the same frequency.

2 Architecture for Flexible Real-time Control

2.1 Description of Architecture

Figure 1 shows the architecture we have designed on the preceding foundations; this Figure will serve as a guide for the ensuing discussion.

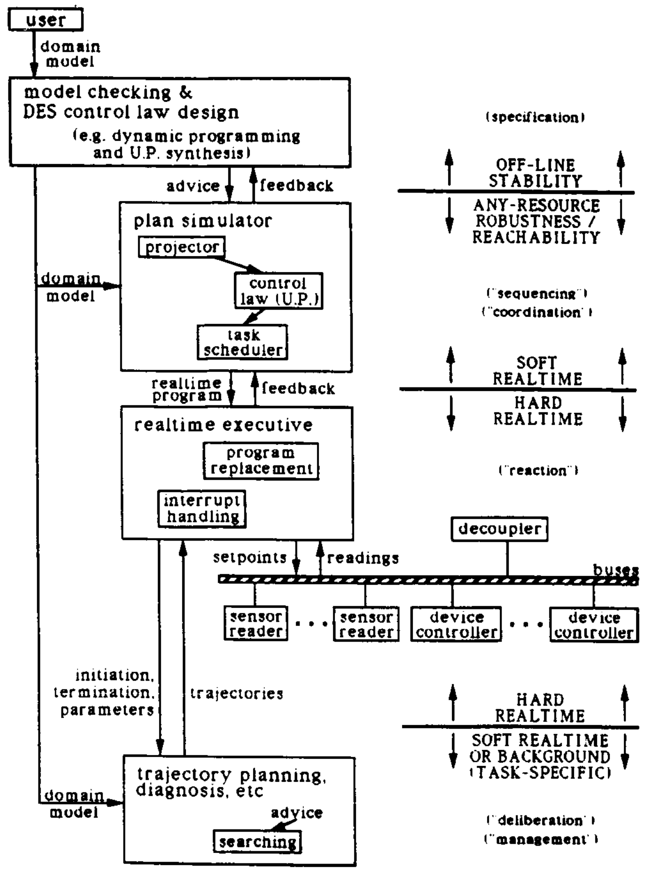

Figure 1: Architecture Diagram.

There is assumed to be a fixed control program which, in our architecture, is split into two parts. One part, usually called the control law or plan, is modified so that its execution or interpretation affects only a simulation model of the controlled system. The other part determines where simulation should begin and end, and is called the "plan simulator".

The basic idea is that the plan simulator chooses a set of initial and goal states, simulates the plan or control law over several possible futures, collects the real actions appropriate to the anticipated states of the controlled system, and so dynamically constructs a small real-time program, which it downloads to the real-time executive. From then on, the real-time executive takes care of communicating with the decoupler, sensor readers, and device controllers, in hard real-time. This mitigates time pressure on the plan simulator, which can proceed with the construction of a new real-time program. This separation of timing concerns is indicated in the architecture diagram by the dividing line between soft real-time and hard real-time.

Each new real-time program specifies: a set of functions implementing actions on the controlled system; a set of functions which test the estimated state of the controlled system; the conditions under which each action is appropriate; and maximum allowed delays between satisfied conditions and appropriate reactions.

In practice, each real-time program is a set of tasks or threads[2] that can be run, suspended, resumed, and so forth, under control of the real-time executive. Threads may be periodic, being executed cyclically and with a definable frequency; or they may require execution only sporadically (e.g. interrupts), in which case there may be a maximum allowable delay between the time the thread becomes relevant and the time it is actually executed.

The real-time executive comes with a "schedulability check" designed specifically for that real-time executive. It is used by the plan simulator to test whether each new real-time program is indeed executable in hard real-time. A successful schedulability check absolutely guarantees that no thread will miss any deadline specified for it. A failed schedulability check tells the plan simulator to design a less demanding real-time program, and if necessary, to choose a less demanding future.

The "actions" performed by the real-time executive do not themselves drive effectors or sensors. Instead, they send on/off signals and parameters to the effectors' and sensors' servo/driver loops, which often run on dedicated microprocessor chips.

Just as servo loops may take extended amounts of time to reach a setpoint, task-specific AI modules may take extended amounts of time to make a decision. Just as the real-time executive is controlling sensori-motor servo loops by updating their parameters and turning them on and off, it also turns task-specific AI modules on and off according to the needs of the moment. That's why Figure 1 shows a typical task-specific AI module *below* the real-time executive, with the device controllers.

The real-time executive, plan simulator, servo loops, and all AI computations are threads which operate in parallel by default. The fraction of computing power (CPU cycles per second) available to each thread is carefully decided by the plan simulator and enforced by the real-time executive. That is, the real-time program constructed by the plan simulator may include actions that adjust (even to 0) the frequency with which certain threads are executed.

As a special case, the real-time program may set even the plan simulator's execution frequency to 0. Even though the real-time executive depends on the now-stopped plan simulator for updated real-time programs, this situation does not mean that the system is permanently stuck with the existing real-time program, since the existing real-time program is conditional, and will have been designed so that new circumstances cause the execution of actions that allow the plan simulator to resume. (If the real-time executive's control over AI modules is reason for Figure 1 to show the latter below the

[2]A thread is a function that has its own piece of program stack, so that execution of the function can be suspended and resumed independently of the execution of other similar threads.

former, perhaps the plan simulator should also be shown below the real-time executive.)

Executed as "background" threads, AI modules can take as long as they like, and provide the system with the full power of on-line intelligent reasoning without endangering the operation of time-critical system elements. At the same time, the plan simulator can ensure, by proper design of the real-time program, that such AI modules are not left behind the rush of events: if the decision of an AI module is crucial to future activities, the plan simulator can program the real-time executive to take actions that delay the on-coming future. This resembles the idea of designing a "reactive subsystem" to ensure total system safety while the AI software thinks (see above), with the important differences that in our case (a) the "reactive subsystem" is not fixed at system design time, and (b) the AI software's thinking may be aborted and redirected by new events.

2.2 Control Theoretic Requirements

In our architecture,

> The automatically constructed real-time program is a partial control law. The real-time executive, in its performance of that control law, functions as a controller.

The hardware devices being controlled will in general have dynamical behavior. Ideally we would like the real-time program to take care of all hard real-time processes, so that the plan simulator itself does *not* have to worry about real-time deadlines. To this end we qualify the CIRCA approach with a stipulation that the real-time program must drive the controlled system into a set of goal states, and then must maintain the controlled system within that set of goal states for an indefinite period of time. Such behavior on the part of a controller is known as "closed-loop stability." For the case of discrete-event control systems, we define it as follows:

> A discrete-event system is *closed-loop stable*, with respect to a set of goal states, if the controller is able to drive the controlled system so that it actually reaches a goal state in finite time, and then (in the absence of disturbances) stays within the set of goal states indefinitely.

(Of course, along the way toward a goal state the real-time program may also decide to do nothing for a while, knowing that a device will do a desirable thing without being forced to do so.) In sum, we require, as a key property of our architecture, that

> Each automatically synthesized real-time program (discrete-event control law) must make the controlled system closed-loop stable.

When the controlled devices are subject to disturbances (i.e. dynamics that are unpredictable) the disturbances may throw the system out of the set of goal states, and we expect the real-time program to drive the controlled system back to a goal state. This too is a familiar notion in control theory:

> A control law is *robust* to the extent that it can keep the controlled system closed-loop stable despite the occurrence of disturbances and unmodelled dynamics.

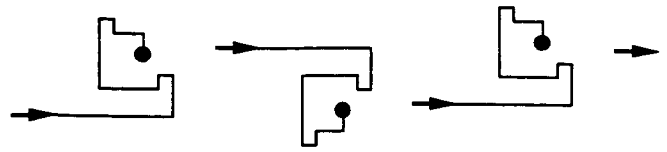
Figure 2: Controlling via Stable Subdomains.

Thus we desire that our automatically synthesized real-time program should be as robust as possible.

Figure 2 depicts the plan simulator's repeated revision of the real-time program, along with the latter's effect upon the controlled system. The closed-loop stability of the controlled system is represented by the "attractor states" into which the system eventually settles. The sequence of attractor states represents the plan simulator's repeated selection of goals. The robustness of each real-time program is represented by the fact that closed-loop stability is achieved despite unpredictable meanderings. The meanderings are rectangular rather than smooth, to emphasize that the real-time program encodes a discrete-event control law, not a continuous-variable control law.

In constructing a real-time program, the plan simulator must choose goals and must anticipate enough possible futures to achieve closed-loop stability and robustness. In practice, in discrete-event control applications, finding a set of goal states that allow closed-loop stability is much easier than in continuous-variable control applications. Cups can be made to stay on tables, doors can be made to stay open, buildings can be made to last, and so on. In AI the standard counter-examples to such stabilities are "spontaneous processes" and the actions of other agents, but remember that for our proposed architecture to work, the real-time program need only be ***closed-loop*** stable. This means that the devices being controlled can act to prevent unwanted interference. Thus the real-time program could prevent other agents from closing doors it wanted kept open, and so forth. Under these conditions, discrete-event closed-loop stability is not hard to find.

Achieving robustness is more difficult, especially for discrete-event control applications. The problem is that unpredictable or unmodelled disturbances may throw the controlled system into so many different states that it may be impossible to synthesize a real-time program that can respond to all of them within the CPU and/or memory resources provided by the real-time executive. Consequently, making a discrete-event control law robust is a fine art. The best that can be done is to include "important" disturbances within the domain model, and further, to include into the model such information as the likelihood, outcome, and severity of a possible disturbance. The availability of that information will allow the plan simulator to decide on-line which disturbances should be anticipated by the real-time control program being constructed, in order to make that program as robust as possible within the available resource limits.

The properties of closed-loop stability and robustness, which must hold of the real-time program being down-loaded to the real-time executive, are shown in the architecture diagram (page 4) as being the responsibility of the plan simulator. There we have referred to "any-resource" robustness to indicate that, once closed-loop stability has been built into the real-time program, the plan simulator may apply whatever resources are left over towards improvements in robustness.

There remains the question of what should happen if, despite the construction of a feasibly robust real-time program, the controlled system makes a low probability transition into a state for which the real-time program has no response. A number of solutions are possible. In the CIRCA approach that problem was passed back to the plan simulator. Since that solution makes system stability dependent on the response speed of the plan simulator, we distrust it. An alternative is to have the plan simulator design the real-time program so as to anticipate all device states that must be controlled within some time horizon, or within hard deadlines; in this way the plan simulator will have a minimum amount of time with which to build a replacement real-time program, and will need to deal only with soft deadlines. This alternative works only if the plan simulator's model of the domain dynamics is correct. Our proposed solution is to have the real-time program include alternative subprograms, along with instructions for their contingent deployment. This solution can work if the real-time executive's execution-time resource is more constraining than its memory space resource — and that is likely.

2.3 Relation to 3-Level Architectures

The architectures of [Firby, 1989], [Bonasso, 1991; Bonasso and Slack, 1992] and [Gat, 1992] specify three "levels", namely

1. a "reactor" containing servo loops, safety reactions, and behaviors;
2. a "sequencer" for deciding which specific activities are needed both now and in the near future;
3. a "deliberator" which contains AI software for planning, diagnosis, metalevel reasoning, and so forth.

These three levels resemble our hard real-time executive, our soft real-time plan simulator, and our non-real-time AI modules, respectively. None of the cited authors took timing seriously, however. In the reactors of their mobile robots, reaction threads for avoiding collisions run every so often, but there is no guarantee whatever that collision avoidance will always be performed in timely fashion. Their programs may work properly for very long times, but ultimately it is impossible even to know whether the robots have ever endured worst-case logjams of threads competing for execution time. Hence we have replaced their reactors with our hard real-time executive, which takes timing very seriously indeed.

A second important difference is we have changed the locus of control. In both [Firby, 1989] and [Bonasso and Slack, 1992] control was hierarchical: the deliberator was on top and in control, deciding what the sequencer should do, and the sequencer decided what the reactor should do. This was changed by [Gat, 1992], who put the sequencer in control and reduced the deliberator to computing task-specific parameters like motion trajectories. The major reasons for this were that (1) reaction should not be kept waiting for deliberation, and (2) old deliberations may be irrelevant to new situations, so the sequencer should get to initiate and terminate deliberations. The locus of control changes again for our architecture: we are moving it all the way down into the reactor (a.k.a. the real-time executive). The reasoning for this is that the results to be computed depend on the situation at the time; the amount of computing power to devote to hard real-time, soft real-time, and background threads depends on what needs to be computed, and hence on the situation at the time; but the situation can change very rapidly, hence the allocations of computing power and the threads being executed may also have to change very rapidly; and so the changing allocations and the initiation and termination of all threads (including deliberations) must be handled by the real-time executive. ([Kaelbling and Rosenschein, 1990] also have a fast reactive system determining which deliberations are relevant and for how long.)

Third, our real-time executive is a real-time microkernel running threads at multiple frequencies, not a single loop that performs a fixed list of functions all at the same frequency.

3 Implementation of the Real-time Executive

Any real-time subsystem adopted as the basis of our architecture should provide the following minimum capabilities:

R1: insulating the processing time allocated to hard real-time threads from the cycle-stealing desires of all other threads.

R2: testing, as part of program execution and without modifying the set of currently executing hard real-time threads, whether a proposed new set of hard real-time threads is such that all its deadlines can be met with the available processing time.

R3: conditionally switching thread subsets on and off, so that the real-time executive is effectively executing a conditional real-time program.

R4: allowing dynamic modification of the hard real-time program (the set of hard real-time threads and the switching logic) while still meeting all deadlines before, after, and during the modification.

R5: ability to do **R1-R4** when the set of hard real-time threads contains both cyclic (periodic) and sporadic (aperiodic) threads.

These requirements can be met in a variety of ways. The alternatives are

1. multiprocessor schedulers, which are the most powerful but expensive in proportion;
2. earliest-deadline-first schedulers, which are the most user-friendly;
3. rate monotonic schedulers, which are the easiest to build.

We believe that our architecture could be implemented equally well on top of any of these options. However, the budgetary constraints on our work incline us away from multiprocessor schedulers (while yet we also believe that for such complex systems as space stations or lunar bases, multiprocessor schedulers are by far the best choice). The best fit with our work on the EVAHR robot is an earliest-deadline-first scheduler, since such schedulers facilitate dynamically changing thread frequencies (for synchronizing computing with robotic motion) and conditional schedules. Our intended implementation is similar to that of [Ramamritham and Stankovic, 1984] with the schedulability check of [Jeffay, 1992].

4 Discussion

4.1 Benefits of the Architecture

The final justification for our architecture consists of three words: safety, generality, and economy. Safety: because the timing of program execution is taken so seriously that even intermittent timing problems can not survive, and because the architecture establishes a feedback loop from computing load to graceful degradation of task performance. Generality: because the system can contain on-line intelligent reasoning, can enact the results of such reasoning in hard real-time, can dynamically replace the system's hard real-time reactions, and can judiciously control its own use of computing power. Economy: because a fixed amount of computing hardware can be time-shared and carefully allotted. Here we elaborate on just a few of these benefits; the complete list appears in [Schoppers, 1993].

- Spatial synchronization. The ability to dynamically modify the frequency of thread execution allows to synchronize computing with motion through space. With collision checks being made at regular distance intervals, slower motions require less calculation. Knowing such facts about itself, an intelligent real-time system can reduce the speed of its motions *for the sake of* reducing the processing power devoted to motion-related threads.
- Graceful degradation. The ability to scale back its operations as necessary to ensure timeliness eliminates the need to design to an imaginary worst case scenario, because there is no longer a sharp performance cliff that the system can fall off in unpredictably disastrous ways.
- Reconfigurability. A survivable system must have several ways of achieving the same result. When the sensor normally used to deliver a given datum malfunctions, another can be used. Since the computing time required to interpret different sensors differs, such result-compatible reconfiguration is only safe in systems (like ours) that can plan their behavior to match their planned computing load.

4.2 Limits of Robustness

Despite our concern for hard real-time and for dynamically achieved robustness, some kinds of mishaps can still happen (of course). If enough sensors malfunction, a robot will be unable to see new dangers approaching, so cannot be held responsible for avoiding them. Similarly a robot may, for the sake of getting its job done, have to place itself in situations that would be dangerous if the robot's computers suddenly died. In all other cases, however, including robotic inability to go on sensing objects it already knew about, as well as computer failures, our software will be aware of the potential mishaps and will continuously and intelligently redesign the robot's behavior specifically to optimize first the safety, then the performance, of the robot in its surroundings.

4.3 When Is Hard Real-time Important?

The development of our architecture was driven primarily by concern for hard real-time response despite the presence of AI software. It is unclear to many people why timing should be taken so seriously. The most common objections are (1) Couldn't we hand-code a fixed layer of real-time reactive behaviors that take care of everything (e.g. collision avoidance) while the AI software is thinking, and (2) Couldn't we make sure that the real-time software works, by means of a test-debug cycle? Sometimes yes, but also sometimes no.

Objection (1) is usually raised by people who have programmed wheeled mobile robots on earth. For such robots there is a small repertoire of actions that can ensure robotic safety, e.g. slamming on the brakes or moving away from impending collisions. However, as soon as either the robot or its environment becomes more complex, a fixed "reactive safety layer" no longer suffices. One example is the Adaptive Suspension Vehicle (ASV) [Payton and Bihari, 1991], which was the size of a bus, with six legs that were each 6 feet high at the hip. Maintaining stable balance while keeping the legs away from each other and while switching between gaits required a super-real-time motion planner. For the ASV, even stopping was so complicated that no predetermined set of "reactions" could have sufficed. Alternatively, more complex environments also prevent a hand-written safety layer, since such a layer must assume that its actions are easily reversible and will not themselves lead to new dangers. On orbit, however, a free-flying robot's action to avoid a collision will move the robot into a new orbit from which it may be both time-consuming and fuel-consuming to return, and on which it is still flying at approximately 20,000 miles per hour. In general it is not true that all robots can be kept safe forever with a fixed set of hand-coded reactions.

Objection (2) cannot be sustained if a thread's missed deadline can lead to loss of human life. Since experienced programmers know better than to claim that they've found "the last bug", and since concurrent software is worse than most, the test-debug approach may well yield life-threatening software [Stankovic, 1988]. The risks can be diminished by applying existing hard real-time scheduling research. Objection (2) is also rebutted if a system's worst-case computational load is several times higher than the average load. For example, setting a robot's top allowable speed to avoid timing problems under a rare scenario will also limit the robot's performance at all other times. Our architecture allows to adapt the robot's top speed to current computational loading. Here too it helps to take hard real-time seriously.

Acknowledgements

We thank Peter Bonasso for stimulating debates. The architecture described herein was elaborated with the support of NASA JSC under SBIR Phase 1 contract NAS.9-18861.

References

[Albus et al, 1987] J. Albus, H. McCain and R. Lumia. *NASA/NBS standard reference model for telerobot control system architecture.* Technical Report 1235, National Bureau of Standards, 1987.

[Bonasso, 1991] R.P. Bonasso. Coordinating perception and action with an underwater robot in a shallow water environment. *Proceedings of the SPIE Conference on Sensor Fusion* IV, SPIE volume 1611, pages 320–330, 1991.

[Bonasso and Slack, 1992] R.P. Bonasso and M. Slack. Ideas on a system design for end-user robots. *Proceedings of the SPIE Conference on Cooperative Intelligent Robotics in Space* III, SPIE volume 1829, pages 352–358, 1992.

[Durfee, 1990] E. Durfee. A cooperative approach to planning for real-time control. In *Proc DARPA Workshop on Innovative Approaches to Planning, Scheduling and Control*, pages 277–283. Morgan Kauffman, 1990.

[Firby, 1989] R.J. Firby. *Adaptive Execution in Complex Dynamic Worlds.* PhD thesis, report RR-672, Dept of Computer Science, Yale University (1989).

[Gat, 1992] E. Gat. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. *Proc AAAI National Conf*, pages 810–815, 1992.

[Jeffay, 1992] K. Jeffay. Scheduling sporadic tasks with shared resources in hard real-time systems. *Proc IEEE Real-Time Systems Symposium*, pages 89–99, 1992.

[Jung and Badler, 1992] M. Jung and N. Badler. Human-like agents with posture planning ability. In *Cooperative Intelligent Robotics In Space III*, SPIE's OE/Technology '92. Boston MA, November 1992.

[Kaelbling and Rosenschein, 1990] L. Kaelbling and S. Rosenschein. Action and planning in embedded agents. In P. Maes (ed), *Designing Autonomous Agents*, pages 35–47, 1990. MIT/Elsevier.

[Khatib, 1986] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *Internat'l Journal of Robotics Research* 5:1 pages 90–98, (1986).

[Musliner et al, 1993] D. Musliner, E. Durfee and K. Shin. CIRCA: a cooperative intelligent real-time control architecture. *IEEE Trans SMC* 23:6, 1993 (to appear).

[Paul et al, 1991] C. Paul, A. Acharya, B. Black and J. Strosnider. Reducing problem solving variance to improve predictability. *Communications of the ACM* 34:8, 1991.

[Payton and Bihari, 1991] D. Payton and T. Bihari. Intelligent real-time control of robotic vehicles. *Communications of the ACM* 34:8, pages 48–63, 1991.

[Ramamritham and Stankovic, 1984] K. Ramamritham and J. Stankovic. Dynamic task scheduling in distributed hard real-time systems. *IEEE Software* 1:3, 1984.

[Schoppers, 1993] M. Schoppers et al. *Robotic Whole-Body Dexterity and a Software Architecture for Robotic Task Performance in Uncontrolled Environments.* Phase 1 SBIR Final Report under NASA solicitation 92-1, contract NAS 9-18861, from NASA Johnson Space Center, Houston TX, 1993.

[Stankovic, 1988] J. Stankovic. Misconceptions about real-time computing. *IEEE Computer* 21:10, 1988.