Platform-Based Embedded Software Design and System Integration for Autonomous Vehicles

BENJAMIN HOROWITZ, MEMBER, IEEE, JUDITH LIEBMAN, CEDRIC MA,
T. JOHN KOO, ALBERTO SANGIOVANNI-VINCENTELLI, FELLOW, IEEE, AND
S. SHANKAR SASTRY, FELLOW, IEEE

Invited Paper

Automatic control systems typically incorporate legacy code and components that were originally designed to operate independently. Furthermore, they operate under stringent safety and timing constraints. Current design strategies deal with these requirements and characteristics with ad hoc approaches. In particular, when designing control laws, implementation constraints are often ignored or cursorily estimated. Indeed, costly redesigns are needed after a prototype of the control system is built because of missed timing constraints and subtle transient errors. In this paper, we use the concepts of platform-based design to develop a methodology for the design of automatic control systems that builds in modularity and correct-by-construction procedures. We illustrate our strategy by describing the (successful) application of the methodology to the design of a time-based control system for a helicopter-based uninhabited aerial vehicle.

Keywords*—*Aerospace simulation, control systems, helicopters, mobile robots, real-time systems.

I. Introduction

Automation of traditionally human-controlled domains has long been a driving force within the controls research community. From industrial plants to vehicles, from airplanes to home appliances, the application of embedded controllers has become pervasive, aided by the relentless increase in capabilities of integrated circuit technology and the advances in control theory. The design process has been, in most cases, a staged process where control laws were first chosen and then an appropriate implementation selected. Initially, the implementation of the control laws required the use of *ad hoc* hardware. Lately, the increase in computing power of microprocessors has led to the implementation of control laws in software. Owing to cost and safety considerations, the microprocessors of choice were often not the top of the line in terms of speed. For this reason, and because most control applications require real-time responses, control laws were often "cheap" heuristics that were validated in an empirical way. In addition, software designers for these applications used techniques that were at best unsound (e.g., communication among tasks implemented by common variables, home-grown primitive operating systems). As long as the complexity of the systems to control was low, this design methodology could yield working implementations. However, recent incidents where incorrect software caused severe problems in very expensive systems, such as the Mars Polar Lander and the Ariane rocket, point out the risks associated with using an outdated approach when developing embedded controllers. A great deal of concern about the control of vehicles, such as airplanes and cars, and about the control of armaments, has led to a strong push for novel design methods that take full advantage of control theory, formal verification, integrated circuit technology, and software practices.

One of the most serious problems in controller design is the current disregard for the interaction of the control laws with their implementation. When a control law is designed, the computational power of the implementation platform is only grossly estimated. This neglect leads to long redesign cycles when the timing requirements of the applications are not met. This situation has its origin in the difficulty of mixing implementation and functional design, and in the

Manuscript received December 20, 2001; revised August 31, 2002. This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) under Contract no. F33615-98-C-3614, Software Enabled Control, administered by the Air Force Research Laboratory, Dayton, OH; and in part by the Microelectronics Advanced Research Corporation (MARCO)/DARPA Gigascale Silicon Research Center.

B. Horowitz, J. Liebman, T. J. Koo, A. Sangiovanni-Vincentelli, and S. S. Sastry are with the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Berkeley, CA 94720-1776 USA (e-mail: bhorowit@cs.berkeley.edu; judithl@eecs.berkeley.edu; koo@eecs.berkeley.edu; alberto@eecs.berkeley.edu; sastry@eecs.berkeley.edu).

C. Ma is with the Northrup Grumman Corporation, El Segundo, CA 90245 USA (e-mail: cedric.ma@northropgrumman.com).

Digital Object Identifier 10.1109/JPROC.2002.805827

difficulty of evaluating quickly an alternative control law or implementation architecture.

Another problem in current controller design is the deficiency in component reusability. Reusing components that are specific to function or implementation decreases time to market and validation time. Assuming several components are fully validated, the difficulty is in composing these objects to work properly. Components may be full subsystems such as engine controllers and antilock braking systems for cars, or sensors such as global positioning systems (GPSs) for airplanes, or software modules such as device drivers, operating systems, and algorithms.

This paper proposes a methodology where the dichotomy between functional design and implementation is bridged and issues related to component reuse are addressed. One main goal of our design strategy is to build in modularity in order to make code reuse and substitutions of subsystems simple. The other main goal is to guarantee performance without exhaustive testing. To achieve these goals we draw on the principles of *platform-based design* [1]. A platform, in this context, is a layer of abstraction that hides the unnecessary details of the underlying implementation and yet carries enough information about the layers below to prevent design iterations. We achieve the goal of guaranteed performance without extensive testing by using a time-based controller. In particular, the choice of a specific software platform to guarantee correct timing performance for the control laws is of interest. Here, we focus on the Giotto software platform, and we show how this platform substantially aids the development of correct embedded controller software in comparison with other approaches. To quantitatively compare the resulting designs, we also present a hardware-in-the-loop (HIL) simulation framework.

We present the platform-based design methodology for embedded controller design by means of a challenging example of automatic control: a helicopter-based uninhabited aerial vehicle (UAV). The difficulty and complexity of the application serve well the purpose of underlining the features of the design method and demonstrating its power. The choices of design solutions are somewhat application dependent, but the overall scheme is not. In this way, the example provides general guidelines for the application of our method. In summary, the methodology we propose works particularly to integrate systems with the following key traits.

1) They contain a sizable amount of real-time embedded software.
2) They often integrate subsystems that were designed to work independently—for example, sensors from different vendors.
3) Their proper operation is important to ensure human safety.
4) They often need to reuse existing code in the form of applications or device drivers or controllers.

The structure of this paper is as follows. In Section II, we lay the groundwork for the helicopter example. Next, in Section III, we introduce the reader to the principles of platform-based design. In Section IV, we describe a software platform for programming time-based controller applications. Finally, in Section V, we present alternative helicopter-based UAV designs that use the concepts of the previous three sections, and we discuss how to compare these designs using simulation.

II. Background for a Model Helicopter

In this section, we introduce the Berkeley Aerial Robot (BEAR) helicopters, and motivate the redesign of their embedded software. We begin with a brief description of the BEAR helicopters and of why autonomous flight is difficult (see Section II-A). We next discuss the first-generation flight control system (see Section II-B), and describe some of its limitations (see Section II-C). Finally, we describe what is needed for a second-generation system (see Section II-D) to overcome these limitations.

A. *The BEAR Helicopters*

The first goal of the BEAR project was to build a flight control system for small, remotely controlled helicopters. The aim was to fly autonomously and to provide a base for research in other areas such as vision. Basic autonomous flight capabilities include hovering, forward flight, turning at a fixed point, and so on. More advanced maneuvers include formation flying and obstacle avoidance. However, it is difficult to achieve even basic autonomous flight, for the following reasons.

1) The helicopter is unstable during hover. It will tip over within a few seconds if left alone. Therefore, the flight control system needs to take an active role in the stabilization of the helicopter.
2) A crash is very dangerous, even at low speeds.
3) The helicopter is an intricate machine, whose mechanical and electronic systems must operate harmoniously under harsh conditions, such as physical vibration and electromagnetic interference.

Moreover, it is difficult to obtain an accurate dynamic model of the helicopter, for the following reasons.

1) The helicopter controls are often coupled. For example, changing the collective pitch affects the amount of power available to the tail rotor, which temporarily affects the yaw characteristics.
2) The behaviors of the helicopter are dissimilar in different flight regimes, such as hover versus forward flight.
3) The airflow surrounding the helicopter body is chaotic, especially near the tail rotor. In addition, the helicopter is affected by wind, and its aerodynamic behavior changes when it hovers near the ground.

In spite of the challenges, the BEAR team managed to build a working flight control system that makes autonomous flight possible. One of the autonomous helicopters, the Yamaha R-50, is shown in flight in Fig. 1. In Section II-B we introduce the structure of this system in more detail.

Fig. 1 A Yamaha R-50 Berkeley autonomous helicopter.

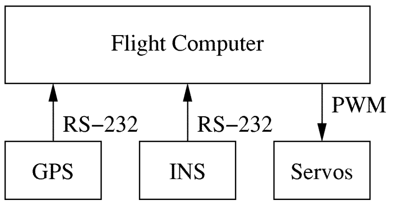


Fig. 2 Structure of the first-generation flight control system.

B. The Flight Control System

As illustrated in Fig. 2, the first-generation flight control system includes a flight computer, a suite of sensors, and actuators. The flight control system interacts with the vehicle dynamics through the sensors and actuators. The major functions of the flight computer are to collect the sensor measurements, to compute control commands based on an underlying control law, and to generate the control commands for the actuators.

The actuators consist of servomotors controlling the main rotor collective pitch θ_M, longitudinal cyclic pitch B, lateral cyclic pitch A, and tail rotor collective pitch θ_T to generate forces and torques applied to the helicopter. We assume the use of an engine governor to regulate the main rotor RPM so the throttle is not directly controlled by our flight control system.

The primary sensors of the flight control system are as follows.

1) Inertial Navigation System (INS). The INS consists of accelerometers and gyroscopes that provide frequent measurements of angular rates and linear accelerations. With proper initialization, the INS can also provide frequent estimations of the helicopter's position, velocity, and orientation. Although this estimate is provided at a high rate—roughly 100 Hz—the error in estimate could grow unbounded over time, owing to sensor bias and limits in sensor accuracy.
2) GPS. The GPS provides position measurement and velocity estimation. The measurement is provided infrequently—roughly 5 Hz—but the error of the position measurement is small—on the order of 1 cm—and is bounded over time.

The INS drift problem is solved by properly integrating the GPS data with the INS data. A Kalman filter is constructed to perform the data integration by taking the dynamic relations of the data into consideration. The Kalman filter provides a frequent and relatively accurate estimation of the helicopter states. In our first-generation system, the Kalman filter is run by the flight computer at 100 Hz; a control law is computed at 50 Hz, and the control commands are received by the actuators at 50 Hz. The derivation of control laws and the selection of the rates are strongly related to the helicopter dynamics.

The helicopter is a nonlinear dynamical system; its equations of motion can be derived from the Newton–Euler equation for a rigid body subject to body forces $f \in \Re^3$ and torques $\tau \in \Re^3$ applied at the center of mass [2]. Let $P \in \Re^3$ and $V = \dot{P} \in \Re^3$ be the position and velocity, respectively, of the center of mass expressed in terms of the inertial frame in north-east-down orientation. The orientation $R \in SO(3)$ of the helicopter can be parameterized relative to the inertial frame by ZYX (or "roll, pitch, yaw"). Euler angles are denoted by $\Theta = [\phi, \theta, \psi]^T$. Let $\omega \in \Re^3$ and $a \in \Re^3$ be the body angular velocity vector and the body linear acceleration vector, respectively. The equations of motion of the helicopter model can be written as

$$\begin{aligned} \dot{P}(t) &= V(t) \\ \dot{V}(t) &= \frac{1}{m} R(\Theta(t)) f(u(t)) \\ \dot{\Theta}(t) &= \Psi(\Theta(t))\omega(t) \\ \dot{\omega}(t) &= I^{-1}(\tau(u(t)) - \omega(t) \times \omega(t)) \end{aligned}$$

where m is the body mass, $I \in \Re^{3\times 3}$ is the inertial matrix, $\Psi : \Re^3 \rightarrow \Re^{3\times 3}$ maps the body rotational velocity to Euler angle velocity, $x = [P^T, V^T, \Theta^T, \omega^T]^T$ is the state vector, and $u = [\theta_M, \theta_T, B, A]^T$ is the input vector. In [3], the previously described system is characterized to be nonminimum phase, i.e., it has unstable internal dynamics. Furthermore, since input u affects both body forces and torques, the linear and rotational dynamics are tightly coupled. Therefore, controlling a helicopter is an extremely difficult task.

Experimental system identification was used to obtain the dynamic model parameters of the helicopter for control design [4]. Given multiobjective design specifications, a specific set of output tracking controllers of satisfactory performance are designed [3]–[6]. Each controller has the static feedback form $u(t) = K_i(x(t), r(t))$ associated with an output $y_i(t) = h_i(x_i(t))$ such that $y_i(t)$ shall track a set point $r_i(t)$ where $y_i, r_i \in \Re^4, h_i : \Re^{12} \rightarrow \Re^4, k_i : \Re^{12} \times \Re^4 \rightarrow \Re^4$ for each $i \in \{1, \ldots, N\}$. By appropriately switching between the controllers a variety of tasks, such as waypoint navigation and high-altitude takeoff, can be accomplished [7].

C. Limitations of the First-Generation System

With basic autonomous flight successfully demonstrated, the BEAR team then set off to equip a number of helicopters with a similar flight control system. Over time, two new and unfamiliar challenges emerged.

1) The first challenge resulted from a widening choice of devices: as the fleet of helicopters became more diverse, so did the selection of sensors, actuation schemes, and computing hardware. Each device provided or received data at different speeds, used different data formats, communicated using different protocols, and so on. To take just one example, the actuators of the first-generation helicopter expected pulse-width modulation signals as input,

whereas the actuators of later helicopters had a serial interface.

The first-generation flight control system reflected the desire to demonstrate the feasibility of autonomous flight, rather than elegance of design. Consequently, the design of the initial system emphasized fast flight computer reaction, achieved by means of tightly coupled sensors, actuators, computer hardware, and embedded software. The tightly integrated flight control system was not prepared to handle the diverse assortment of new devices. Inevitably, any change to the original system required an extensive software rewrite followed by an extended verification process.

In short, the original embedded software was not written with modularity in mind. Yet it would be prohibitively expensive to rewrite all of the software for each particular combination of devices. Instead, we would like to develop embedded software that is simple to configure, so that new components can be added or substituted with relative ease.

2) The second challenge resulted from the event-based nature of the first-generation flight control computer. To ensure the fastest possible response, the computer was set up to process the incoming sensor data as soon as it arrived and to immediately send the control output to the actuators.

As an example of the problems that arose in this event-based system, consider the following first-generation setup. The GPS and INS were synchronized with each other but not with the control computer. The GPS sent readings to the control computer at 5 Hz. The INS sent readings at 100 Hz. The control computer ran the control task at 50 Hz. Because of the lack of synchronization, the sensor data seen by the control computer ranged from 0 ms to 10 ms out of date. Owing to clock drift, this amount of time was nondeterministic. Similarly, the servos were triggered by a clock whose rate was independent of the control computer's clock. Since the servos were triggered at 23.78 Hz, by the time the actuators used the control data, these data could be 42 ms out of date.

Unfortunately, the different rates of the sensors, actuators, and computer resulted in a system whose timing behavior was not particularly easy to analyze. Consequently, the physical behavior of the helicopter could vary greatly from the simulation results.

We have presented several limitations of the first-generation helicopter system. In the next section, we discuss properties that the second generation should have in order to lessen these limitations.

D. A Second-Generation System

We would like a helicopter system whose overall physical behavior can be analyzed and predicted. To this end, we need a unified approach to the timing behavior of the elements—sensors, actuators, and computer—of the control system. We believe the key to this unified approach lies in a time-based, modular design.

1) A time-based design: The system should be time-based in order to allow easy analysis of its closed loop behavior. However, the system must maintain compatibility with existing devices, such as sensors, that are not time-based. A clear boundary between the system's synchronous and asynchronous elements must be drawn, and provisions must be made to bridge the gap.
2) A modular design: The new system must allow the designer to choose from a diverse mix of sensors, actuation schemes, and controllers. The new system must allow a configuration of the same software to run on different helicopters, which may have very different physical dynamics and devices.

III. Platform-Based Design Methods

Automatic control systems, such as the BEAR helicopters, can be designed with legacy code reuse and safety guarantees, and without deficiencies in subsystem integration. This section presents the building blocks that will later be used to design such a system.

The building blocks we use are those of platform-based design. The main tenet of platform-based design is that systems should employ precisely defined layers of abstraction through which only relevant information is allowed to pass. These layers are called *platforms*. For example, a device driver provides a layer of abstraction between an operating system and a device. This layer hides most of the intricacies of the device, but still allows the operating system to configure, read from, and write to the device. Designs built on top of platforms are isolated from irrelevant subsystem details. A good platform provides enough useful information so that many applications can be built on top of it. For example, the C programming language, despite its flaws, provides an abstraction of instruction set architectures that is versatile enough to allow many applications to be written in C.

A system can often be usefully presented as the combination of a top level view, a bottom level view, and a set of tools and methods to map between the views. On the bottom, as depicted in Fig. 3, is the architecture space. This space includes all of the options available for implementing the physical system. For example, a PC can be made from a CPU from Intel or AMD, motherboards from a variety of vendors, etc. On the very top is the application space, which includes high-level applications for the system and leaves space for future applications. These two views of the system, the upper and the lower, should be decoupled. Instead of interacting directly, the two design spaces meet at a clearly defined interface, which is displayed as the shared vertex of the two triangles in Fig. 3. The thin waist of this diagram conveys the key idea that the platform exposes only the necessary information to the space above. The entire figure, including the top view, the bottom view, and the vertex, is called the *system platform stack*.

The platform-based design process is a "meet-in-the-middle" approach, rather than being top-down or bottom-up.

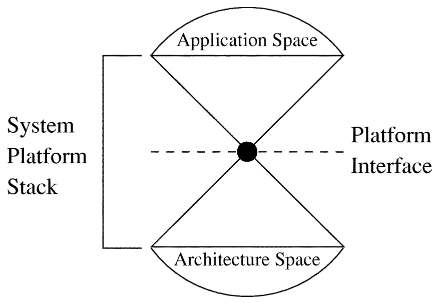


Fig. 3 The system platform stack.

Top-down design often results in unimplementable requirements, and bottom-up design often results in a mess. In platform-based design, a successive refinement process is used to determine the abstraction layer. In this process, an initial application design helps to define a provisional platform interface. This platform interface in turn suggests what the architecture implementation needs to provide. The architecture space can then be explored to find an implementation that comes closest to satisfying both the platform interface and the preset physical requirements. The platform interface may need modification, and the application design may need some rethinking. This process repeats until an appropriate platform interface has been defined. At this point the platform interface is a reasonable and well-specified point of contact between the application and architecture spaces. As a result, new applications may be developed to use the same platform, and new architectures may be explored for future support of the same platform interface. As we have seen, the focus of platform-based design is the correct definition of the platform interface, a process that may involve feedback loops.

IV. A Time-Based Control Platform: Giotto

Control laws for autonomous vehicles are typically implemented on architectural platforms consisting of programmable components (e.g., microprocessors, digital signal processors), memory (e.g., flash, RAM, and ROM), sensors, and actuators. The control laws are almost always implemented as software stored on ROM or flash memory running on the programmable components. There are difficulties in mapping the control laws onto these kinds of architectural platforms.

1) In most cases, the controller must react in real time. A software implementation is intrinsically slower than hardware. In addition, the computing part of the platform is, most of the time, a standard single processor. Thus, the concurrency of the function to implement is lost: concurrent tasks have to be sequentialized. These platforms are equipped with a real-time operating system (RTOS), i.e., a lightweight, low-overhead operating system that schedules tasks to be executed on the processor. There are many scheduling algorithms available to optimize the processor utilization while maintaining deadlines for task execution. The most efficient ones in terms of processor utilization are dynamic. However, it is *very* difficult, if not impossible, to guarantee that deadlines are met with these scheduling algorithms. Static scheduling algorithms are much less efficient in terms of processor utilization, but the analysis is much easier. However, owing to the inefficiencies in the use of the resources, meeting deadlines with this scheme implies an overdesign that is hardly affordable. The most popular RTOSs support preemption and dynamic scheduling.
2) The sensor measurements and the commands given to actuators must be carefully analyzed for errors and malfunctions. The implementation of the control laws must be aware of these nonidealities. However, if the software implementation of the control laws directly includes information about the peripherals of the platform, then reusing the software with different platforms is virtually impossible. An efficient solution for this problem is to use device-geared software processes to isolate the control software and the algorithms from the physical characteristics of the devices.

To ameliorate these difficulties, we introduce a new abstraction layer that sits between the RTOS and the functional description of the control laws. This abstraction layer provides the control designer with a more relevant and very simple method for programming the control laws to meet real-time constraints. However, the control designer must adhere to the simple guidelines that this abstraction allows. In this way, the abstraction layer restricts the design space available to develop the control laws, but significantly shortens the time to market and increases the correctness of the design.

To illustrate this idea using the hourglass platform-based design figure, we place the possible control laws in the application space on the top, and the RTOS in the architecture space on the bottom, as shown in Fig. 4. The proposed abstraction layer makes up the interface between these two views. Ideally, this platform interface should pass the timing constraints of the application downwards, and should pass the performance capabilities of the architecture instance upwards. On the basis of these constraints, the platform's tools should be able to determine if the timing requirements of the application can be fulfilled. In this section, we discuss in detail an abstraction layer between the RTOS and the real-time control laws that is chosen for the helicopter embedded software. This abstraction layer is the Giotto programming language.

Giotto consists of a formal semantics and a retargetable compiler [8]. Giotto has already been used to reimplement the control system onboard a small autonomous helicopter developed at ETH Zürich [9]. In this section, we first present a comparison between Giotto and other tools that may be used to build the abstraction layer we desire (Section IV-A). We then present a brief introduction to Giotto (Section IV-B). Finally, we discuss the tools that may be used to map a Giotto application to its possible implementations (Section IV-C). A more detailed introduction to Giotto is presented in [10].

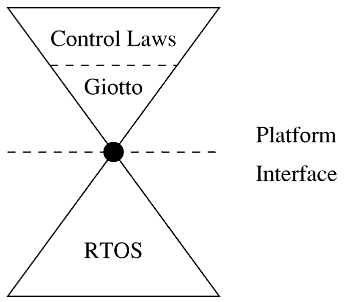


Fig. 4 Platform-based design illustration of control-law implementation on an RTOS.

A. Giotto Compared with Related Technologies

The benefits of introducing a platform always trade off with a decrease in efficiency. Therefore, we would like the chosen platform layer between the control laws and the RTOS to be as efficient and well suited to our needs as possible. In this section, we demonstrate why Giotto is the best fit for our platform implementation. The most popular alternative tools that might be used to build such a platform are the synchronous programming languages, the services provided by the RTOS itself, and the time-triggered architecture (TTA).

The synchronous programming languages are a family of programming languages that have been under development since the 1980s. Esterel and Lustre are the exemplars of this family of languages [11], [12]. Esterel is appropriate for applications in which control flow is of primary importance, and Lustre is appropriate for applications in which data processing is of primary importance. Both Giotto and synchronous languages try to reduce the unpredictable effects of concurrency. The same general approach is taken by both: all activities of the program may be dated on a single timeline. Informally, a *timeline* is a sequential ordering of the activities of a program. In a traditional multithreaded application, each thread has its own timeline. These timelines may be interleaved in many possible ways depending on the operating system scheduler and the inputs from the environment. In contrast, a synchronous program or a Giotto program specifies exactly one way to interleave the timelines of the program's components. Thus, programs written in Giotto and the synchronous languages are more deterministic than those written for traditional multithreaded operating systems. The main problem with the implementation of synchronous programming languages is that they do not make efficient use of the multitasking capabilities provided by a standard RTOS. The synchronous programming community has often avoided the use of multitasking features, since without care these may lead to nondeterministic behavior. However, without the use of multiple tasks, the CPU of a control system may go underutilized. In contrast, since Giotto programs are multithreaded, they can incorporate preemption, and thus fuller CPU utilization.

As will be further discussed in Section IV-C, Giotto tasks are transformed by the Giotto compiler to operating system threads. At runtime, these threads are scheduled by an RTOS. Thus, the Giotto programmer's abstraction could be viewed as similar to the abstraction provided by an RTOS. However, standard RTOSs do not provide integrated schedulability analysis. It is up to the programmer to perform such analysis on her own. In addition, RTOSs commonly provide many styles of intertask communication. Some of these (e.g., shared memory) can be tricky to program. In contrast, Giotto provides only a single communication semantics, but automates its implementation.

The TTA is a hardware and software system that provides fault-tolerant time-based services [13]. It consists of specialized boards that communicate using its own time-based communication protocols. In contrast, Giotto concentrates on providing an *abstract* programmer's interface. The TTA's time-based nature makes it particularly suitable for running Giotto. However, Giotto can also be run in other hardware and software environments. For example, Giotto is run on custom hardware and software designed at ETH Zürich, on the Motorola MPC 555 processor running the RTOS OSEKWorks, and on Linux (without real-time guarantees).

B. The Giotto Programmer's Abstraction

In this section, we discuss the abstraction that Giotto presents to the programmer. Control applications often have periodic, concurrent tasks. For example, the helicopter control application runs a measurement fusion task at a frequency of 100 Hz, and a control computation at 50 Hz. Typically, the periodic tasks communicate with each other. Control applications also need a means to input from and output data to their physical environment. Finally, control applications often have distinct *modes* of behavior; in two different modes, different sets of concurrent tasks may need to run, or the same set of tasks may need to run but at different rates. For example, a robot on a discovery mission may first need to run one set of tasks to navigate to a location; once that location is found, the robot may need to run a different set of tasks to query its surroundings. Giotto provides the programmer a way to specify applications with periodic, concurrent, communicating tasks. Giotto also provides a means for I/O interaction with the physical environment, and for mode switching between different sets of tasks.

Consider the example program of Fig. 5. The concurrent tasks—Fusion and Control—are shown as rectangles with rounded corners. Each task has a logical execution interval. In our example, Fusion logically executes from 0 ms to 10 ms, from 10 ms to 20 ms, etc., whereas Control logically executes from 0 ms to 20 ms, from 20 ms to 40 ms, and so on. Each task has *input ports* and *output ports*, shown as black circles. A task's input ports are set at the beginning of its logical execution interval. During its execution, the task computes some function, and the results are written to its output ports at end of its logical execution interval. For example, the input ports of Fusion are set at 0 ms; between 0 ms and 10 ms, Fusion computes its function; at 10 ms, the result of this function is written to Fusion's output ports.

A Giotto program may also contain *sensors* and *actuators*, both of which are depicted as white circles. Rather than being actual devices, sensors and actuators are programming lan-

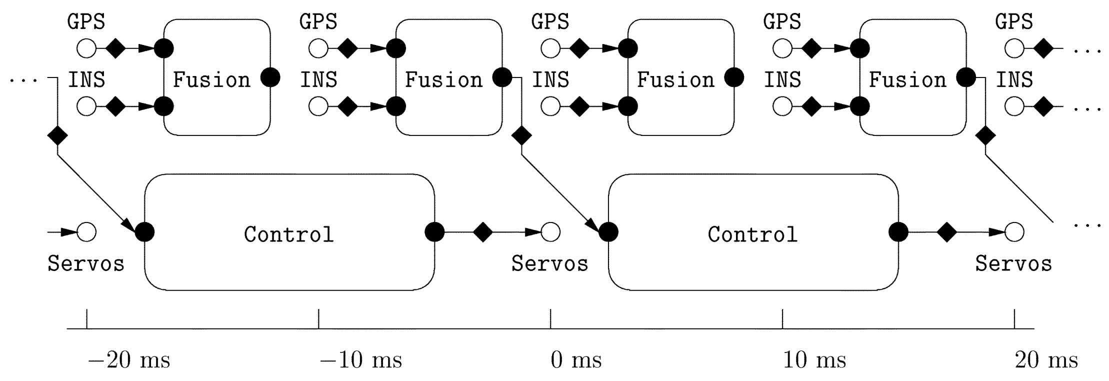


Fig. 5 An example Giotto program for helicopter control.

guage constructs that let the programmer define how to input data to and output data from a Giotto program. Logically, sensors and actuators are passive: they are *polled* at times specified in the Giotto program, and cannot push data into the program at their own times. Our example program has two sensors, GPS and INS, and one actuator, Servos. The sensors are read at 0 ms, 10 ms, 20 ms, etc., and the actuator is written at 0 ms, 20 ms, and so on.

Tasks communicate with each other, and with sensors and actuators, by means of *drivers*, which are shown as diamonds. In Fig. 5, the drivers connect the GPS and INS sensors to the input ports of the Fusion task. They also connect the output port of Fusion to the input port of Control, and the output of Control to the Servos actuator. Thus, the Fusion task that executes between 0 and 10 ms receives its inputs from the GPS and INS readings at 0 ms. Similarly, the Control task that starts at 0 ms receives its inputs from the Fusion task that finishes at 0 ms, and writes its outputs to the Servos actuator at 20 ms.

In this section, we have described the abstraction that Giotto presents to the programmer. In the next section, we will discuss the Giotto compiler, which transforms Giotto programs into RTOS applications.

C. Tools to Implement Giotto

The platform-based design methodology advocates the use of tools to map from high-level abstractions to the underlying architecture. Here, the Giotto language is the abstraction, and RTOSs constitute the architecture. This section describes the Giotto compiler, which maps Giotto programs to RTOS executables. Just as a conventional C compiler transforms C programs into object files for an instruction set architecture, the Giotto compiler transforms Giotto programs into executables for an RTOS.

The input to the Giotto compiler is a Giotto program, together with code to implement the tasks, drivers, sensors, and actuators. These other pieces of code may be written in a conventional programming language such as C. These pieces of code are annotated with worst-case execution times. In effect, these annotations allow constraints to pass upwards from the architecture to the platform. The Giotto program also specifies timing constraints that pass downwards toward the architecture. Using both sets of constraints, the compiler performs schedulability analysis, which ensures that all deadlines in the executable it produces will be met [14]. The compiler then generates an object file that can be run on any RTOS. This object file contains instructions for the Embedded Machine, which is an RTOS-independent virtual machine [15]. At runtime the Embedded Machine sequences and schedules the tasks, drivers, sensors, and actuators of the Giotto program.

An RTOS typically supports applications with multiple threads of control, whether they are called threads, processes, or tasks. In addition, an RTOS usually provides a means for scheduling these threads, whether by priorities, deadlines, or round robin. The Giotto compiler aims to make efficient use of these RTOS services. The Giotto compiler currently uses heuristics for developing a preruntime schedule: drivers, sensors, and actuators are executed at the fixed times given by the Giotto program, whereas tasks are scheduled using earliest deadline first. For example, in the program of Fig. 5, GPS and INS are executed at 0 ms, 10 ms, and so on, and the deadline of Fusion is always 10 ms after its start time.

V. Case Study: End to End Design of Helicopter-Based UAV

In this section, we discuss strategies for building a helicopter-based UAV, with two main goals in mind.

1) The first goal is to incorporate both asynchronous input devices and a time-based controller. In Section II, we saw that the sensors send data at their own, possibly drifting, rates. We also presented the advantages of using a time-based controller. However, we also saw in Section IV that our chosen time-based controller reads from input devices at its own fixed times. Thus, combining these components gives rise to a mismatch in timing behavior that needs to be addressed.
2) The second goal is to build a system that is modular enough to allow one suite of devices (e.g., a sensor suite) to be replaced by another.

To achieve these two goals, we will use the principles of platform-based design presented in Section III. We will show how the insertion of a layer of abstraction between the devices and the controller can be used to bridge the timing mismatch and allow for the inclusion of different sensor suites.

In Section V-A, the platform-based design principles are used to specify a functional description of the helicopter-based UAV. In Section V-B, we describe the process of implementing the functional description. Finally, in Section V-C, we discuss how to compare implementation alternatives.

A. Building Functional Description Using Platform-Based Design

In Section III, we explained how to begin the platform-based design process by separating the system into two views: the application and the architecture. Here we apply this separation to our helicopter-based UAV, which is naturally seen from two views. From the top, a designer sees the time-based control application. From the bottom, a designer sees the available physical devices, such as the helicopter, the sensors, and the actuators. Fig. 6 situates these two views in the context of platform-based design: the time-based control application sits in the application space, while the physical devices make up the architecture space. Following the *meet-in-the-middle* approach of platform-based design, we include an intermediate abstraction layer, the UAV platform, whose top view is suitable for time-based control and whose bottom view is implementable using the available devices.

We next describe the functionality of the UAV platform.

1) Interaction with devices: The UAV platform should be able to receive transmissions from the sensors at their own rates and without loss of data. Similarly, the platform should be able to send commands to the actuators in the correct formats. The platform will also need to initialize the devices. Furthermore, the platform should be able to carry out these interactions with a variety of different sensor and actuator suites.
2) Interaction with control application: The UAV platform should provide measurement data to the control application in the format and at the frequency dictated by the controller. Similarly, the platform should receive the commands from the controller at times dictated by the controller, and immediately send them on to the actuators. The platform should also be able to support a variety of controllers.

One natural conclusion is that the platform should buffer incoming data from the sensors, convert sensor data into formats usable by controller applications, and convert control commands into formats usable by actuators. In Section V-B we describe in detail two ways to implement the functions of the platform.

B. Implementing Functional Description Using Platform-Based Design

While the platform-based design methodology is a meet-in-the-middle approach, it suggests implementing the application first. In this section we begin by discussing the realization of the controller application. This implementation, as discussed in Section V-A, places constraints on the platform. Platform implementations that meet these constraints are presented next. Though the platform is constructed to work with a variety of available devices, we work with only one such architecture instance. Comparing the efficacy of alternate sensors and actuators is beyond the scope of this paper.

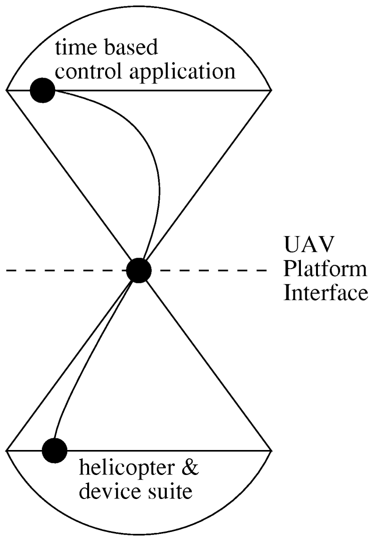


Fig. 6 Platform-based design of helicopter-based UAV.

1) Implementing the Controller Application: To attain the benefits of time-based control, presented in Section II-D, the controller application is realized using the Giotto programming language. Section IV-B presented a rough sketch of the Giotto implementation in Fig. 5. The two essential tasks are Fusion and Control. Fusion combines the INS and GPS data using a Kalman filter and is run at a frequency of 100 Hz. Control uses the output from Fusion to compute the control law shown in Section II-B at a frequency of 50 Hz. The frequencies of these two tasks are chosen based on the expectations of the control law and on the limitations of the devices. Each task is written as a separate C function. These C functions are referenced inside of the Giotto program, which schedules and runs them as described in Section IV-C.

Unfortunately, the advantages of using such a time-based controller application—in particular, reduced jitter—trade off with the disadvantage of increased latency. For example, in Fig. 5, consider the Control instance that executes from 0 to 20 ms. The incoming sensor data for this instance was sampled at -10 ms. At 0 ms the data has been transformed by Fusion and is ready for use by Control. The output of Control is not written to the actuator until 20 ms, resulting in a total latency of 30 ms. This is unfortunate, since a new output of Fusion is available at 10 ms. In fact, the actual execution time of the Control task is much less than 10 ms, so Control should ideally be scheduled *after* Fusion has made a new output available at 10 ms.

One way to reduce the latency of the program of Fig. 5 is to increase the frequency of Control to 200 Hz, so that its deadline reduces to 5 ms. However, this results in Control being executed unnecessarily often. Instead, we wish to execute Control only once per 20 ms interval, but to retain the 5 ms deadline. Achieving this result in Giotto is possible, with

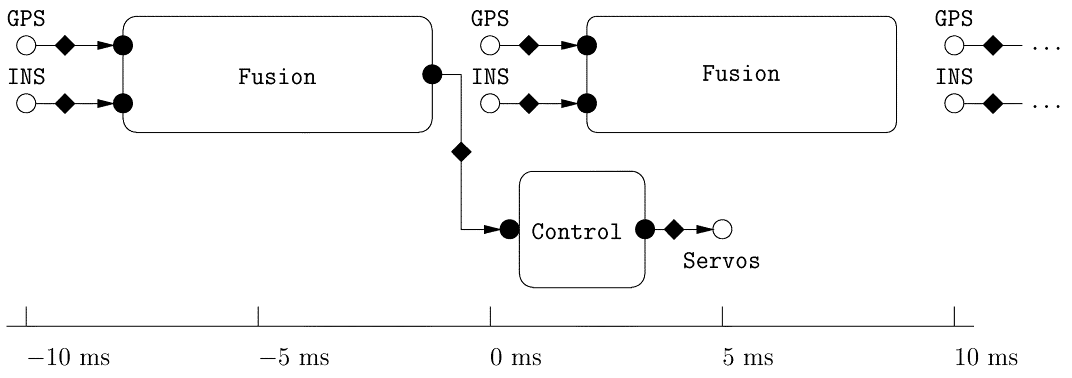


Fig. 7 Refined Giotto program.

a little extra effort. We first note that each Giotto driver is equipped with a *guard*, which is a condition on the driver's input ports. If the guard of a task driver evaluates to true, the task is executed, but if it evaluates to false, the task is not executed. To fix our problem, we add a counter that is incremented every 5 ms, and we add a guard to the driver of Control that evaluates to true when the counter equals $0 \bmod 4$. Control thus executes from 0 to 5 ms, from 20 to 25 ms, and so on. The refined Giotto implementation with reduced latency is displayed in Fig. 7.

2) Implementing the UAV Platform: Having considered a realization of the time-based controller, we now turn to the UAV platform. In Section V-A, we discussed the requirements that our UAV platform needs to fulfill. We now present two possible implementations of the UAV platform, both of which fulfill these requirements. The first implementation uses one computer, effectively implementing in software the buffer discussed in Section V-A. The second uses two computers, and implements the buffer in hardware.

a) First Implementation: One Computer: The single-computer implementation has three main elements, which are depicted in Fig. 8.

1) Data processor. The data processor is an independent process, similar to a standard interrupt handler. In the sensing case, it responds to the new sensor data sent by the devices, and saves this data to a shared memory space with the sensor-specific data format intact. In the actuating case, the data processor passes on to the servos the messages sent by the controller application.
2) Shared memory. The shared memory contains recent sensor readings, and is implemented as circular buffers. Data are placed into the circular buffers by the data processor, and can be accessed by the controller application. In this way the controller application can grab the sensor data without worrying about the timing capabilities of each sensor.
3) Data-formatting library. Within the controller application, the sensor-specific data format must be transferred to the format that the control computation expects. In the sensing case, the controller application uses the data-formatting library to transform the buffered sensor readings. In the actuating case, the controller application uses the library to convert actuation commands into the format expected by the servos.

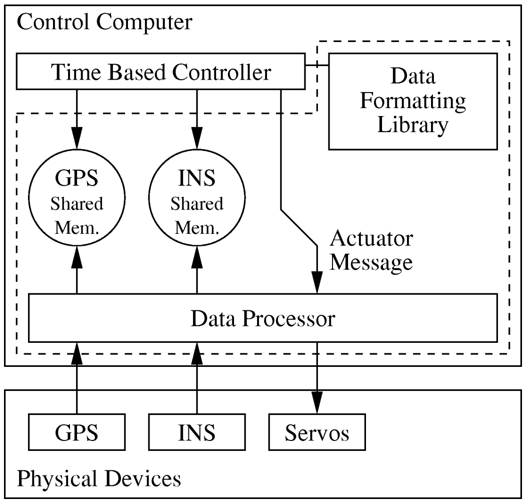


Fig. 8 First implementation of UAV platform. The dashed line encloses the platform implementation.

Recall from Section IV-C that the controller application comes with guarantees about the deadlines of its own internal tasks. These guarantees, however, do not take into account the time that may be needed by other processes or interrupt handlers. If more than a "negligible" amount of time is spent in the other processes, then the timing guarantees of the controller application may cease to be valid. For this reason, the previously shown design keeps the time needed by the data processor to a bare minimum. The data transformations necessary are instead written into the data-formatting library and called from within the control tasks. The benefit of this approach is that the timing guarantees of the controller application are preserved, as much as possible.

b) Second Implementation: Two Computers: Though the single-computer implementation results from a platform-based design methodology, one might well argue that it does not adhere to a strict separation of the control from the sensor details. This problem results from the fact that the format conversion functions are run from within the controller. We have argued that this is needed to preserve the guarantees on the timing of the controller application. In a second implementation, both the timing guarantees and the separation of

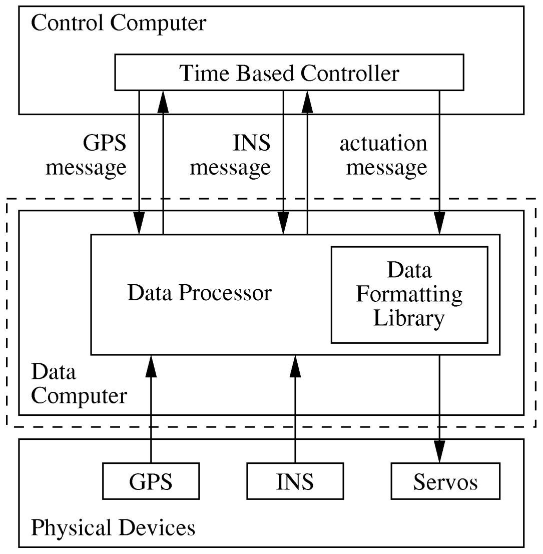


Fig. 9 Second implementation of UAV platform. The dashed line encloses the platform implementation.

control from sensor details are maintained by including two computers on the helicopter. This alternative is depicted in Fig. 9. The two computers perform distinct functions.

1) The control computer runs the controller application. When the application needs the most recent sensor reading, it sends a request to the data computer. The application also forward actuator commands to the data computer.
2) The data computer performs the same functions as the data processor and data-formatting library from the first implementation. It receives readings from the sensors. When the control computer requests the most recent reading, the data computer replies with this reading in the correct format. When the control computer sends an actuator command, the data computer relays the command to the servos in the correct format.

In this implementation, the separation of control from sensor details is strictly followed, and the timing guarantees of the controller application are maintained. However, there is a tradeoff. The downside to this second approach is an added amount of latency that is introduced between the time the sensor readings are taken and the time the control laws use the measurements. This latency is introduced by the communication between the control computer and the data computer. This increase in the staleness of the data is a common tradeoff with more structured designs. In the next section, we discuss methods for a quantitative comparison of the two designs.

C. Comparison of Implementation Alternatives

Now that we have two platform implementations the next question is natural: which one is the best? Ideally, carefully controlled tests could be performed on the physical system. However the fact that we are working with an automatic control system makes that a difficult proposition.

1) Testing is expensive and potentially dangerous. For the helicopter, a safety pilot must be on hand for every test run in case a takeover is necessary.
2) Tests are difficult to standardize. For example, the winds and GPS signal strength cannot be controlled.

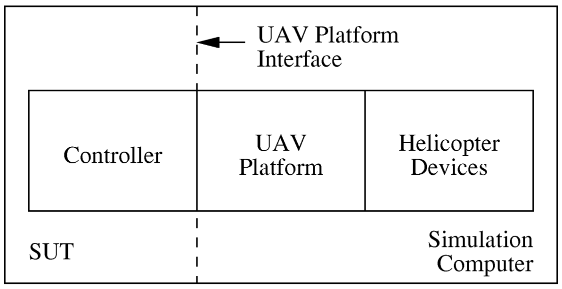
To ameliorate this problem, we propose the use of a HIL simulator, which allows for the direct testing of the entire control system [16], [17]. Instead of mounting the control system onto the helicopter, the controller (often called the *system under test*) is connected to a simulation computer. The simulation computer uses a dynamic model to mimic the exact inputs and outputs of the sensors and actuators on the helicopter.

HIL simulators are well suited to take advantage of the abstraction layers provided by platform-based design. The suitability arises from the capacity to slide back and forth the dividing line between the simulation computer and the system under test, as shown in Fig. 10. To compare the controller applications, the simulator should act as the platform interface, and the controller applications should act as the system under test. To compare platform implementations, the simulator inputs and outputs should closely approximate those of the actual devices, and the controller application and platform implementation should be part of the system under test.

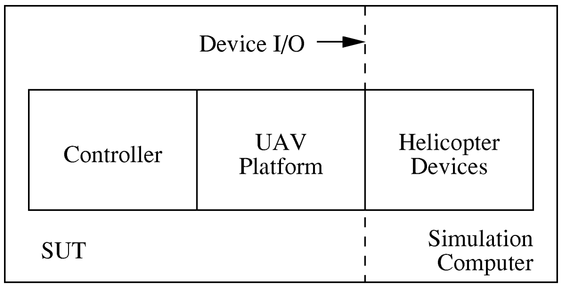
The Giotto controller utilizes the first-generation helicopter controller algorithm of [4] that was used to demonstrate autonomous flight on the Yamaha R-50. This controller application, along with the realization of the first platform, runs on VxWorks, an RTOS, and communicates by serial with the simulator computer. The simulator uses the dynamic model for the Yamaha R-50, whose origin was also presented in Section II-B. The simulator solves the model's differential equations using a numerical recipes package and also runs on VxWorks. To properly view the working combination of the Giotto controller, platform realization, and simulation process, the motion of the simulated helicopter was communicated by Ethernet to a graphical flight display program where a three-dimensional helicopter performed on screen. Fig. 11 displays a screen shot of this GUI.

Because the simulation computer must imitate a physical system, the simulator must meet two additional constraints.

1) The simulator must run in real time. This greatly limits the choice of operating systems available to run the simulator. It also mandates a careful selection of the numerical methods used for solving the model's differential equations [17].
2) The simulated helicopter should faithfully duplicate the dynamics of the real-world helicopter. The parameters of the simulator should be set to values that have been measured on the helicopter. To check that the simulator software mathematically implements the behavior of the physical models, we propose the use of system identification techniques. The parameters of the mathematical model should be compared with those obtained using system identification on the I/O behavior of the HIL simulator.



(a)



(b)

Fig. 10 HIL simulation.

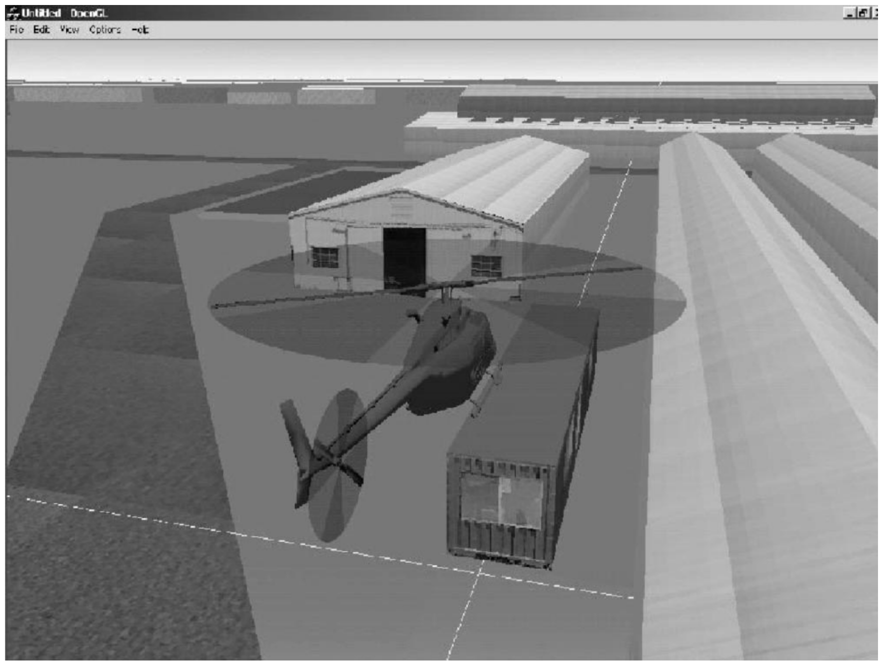


Fig. 11 Graphical flight display.

The proposed simulation framework, in combination with platform-based design, allow for the development of automatic control systems that are modular and have guaranteed performance.

In this project, we have thus far obtained results through the combination of a controller application running in Giotto, the first platform implementation option, and a realization of the HIL simulator.

A further research effort is to realize the second platform implementation option. On the completion of this step, the HIL simulator can be used to examine the comparative performance of the second platform implementation as well as alternative control algorithms. In addition, the chosen Giotto controller and platform realization that reside on one or more flight computers can be flown on the physical helicopter without any alterations.

VI. Conclusion

In this paper, we presented a methodology for the design of embedded controllers. Our methodology is predicated on the

Fig. 12 Flight capable hardware with RTOS.

principles of platform-based design, which uses layers of abstraction to isolate applications from low-level system details and yet provides enough information about the important parameters of the lower layers of abstraction to prevent costly redesigns. The platform-based design approach also provides a framework to pass constraints from higher levels of abstraction to lower ones. Thus, platform-based design provides a basis for successive refinement and correct-by-construction design. In addition, by providing the appropriate layers of abstraction, the methodology allows for the integration of legacy code and "foreign" subsystems. An essential layer of abstraction in our methodology is the software platform provided by the programming language Giotto, which allows a clean implementation of a time-based controller application. To present how our design methodology can be applied, we have discussed two redesigns of the control system of a helicopter-based UAV. These designs go a long way toward meeting the goals for our second-generation helicopter control system.

1) The use of platform-based design allows us to build a bridge between the time-based controller application and the non-time-based sensors and actuators.
2) A time-based controller eliminates the timing irregularities present in the first-generation systems. Further, the Giotto compiler ensures that the controller application meets its timing requirements.
3) Our platform-based design achieves a high degree of modularity. For example, to substitute a different sensor suite in our first redesign requires only changes to the data processor and the data-formatting library. The data processor would require a different sensor initialization routine and a new circular buffer; the formatting library would need a new format conversion routine. However, no part of the controller application would need to be changed.

Though our case study contains many details that are specific to our helicopter system, our methodology is widely applicable. We believe that the combination of time-based control and platform-based design can be generally applied to automatic control systems, for which legacy software, independently engineered subsystems, and strict reliability and timing requirements all play a crucial role.

REFERENCES

[1] A. Sangiovanni-Vincentelli. (2002, Feb.) Defining platform-based design. *EEDesign* [Online] Available: http://www.eedesign.com/story/OEG20020204S0062 and http://www.gigascale.org/pubs/"

[2] R. Murray, Z. Li, and S. S. Sastry, *A Mathematical Introduction to Robotic Manipulation*. Boca Raton, FL: CRC, 1994.

[3] T. Koo and S. Sastry, "Output tracking control design of a helicopter model based on approximate linearization," in *Proc. 37th Conf. Decision Contr.*, 1998, pp. 3635–3640.

[4] D. Shim, "Hierarchical flight control system synthesis for rotorcraft-based UAV's," Ph.D. dissertation, Univ. California, Berkeley, 2000.

[5] D. Shim, T. Koo, F. Hoffmann, and S. Sastry, "A comprehensive study of control design for an autonomous helicopter," in *Proc. 37th Conf. Decision Contr.*, 1998, pp. 3653–3658.

[6] T. J. Koo, "Hybrid system design and embedded controller synthesis for multi-modal control," Ph.D. dissertation, Univ. California, Berkeley, 2000.

[7] T. J. Koo, G. J. Pappas, and S. Sastry, "Mode switching synthesis for reachability specifications," in *Hybrid Systems: Computation and Control*. Heidelberg, Germany: Springer-Verlag, 2001, pp. 331–346.

[8] T. Henzinger, B. Horowitz, and C. Kirsch, "Giotto: A time-triggered language for embedded programming," in *Proc. 1st Int. Workshop Embedded Software*, 2001, pp. 166–184.

[9] C. Kirsch, M. Sanvido, T. Henzinger, and W. Pree, "A Giotto-based helicopter control system," presented at the 2nd Int. Workshop Embedded Software, Grenoble, France, 2002.

[10] T. Henzinger, B. Horowitz, and C. Kirsch, "Embedded control systems development with Giotto," in *Proc. Int. Workshop Lang., Compilers, Tools Embedded Syst.*, 2001, pp. 64–72.

[11] G. Berry and G. Gonthier, "The Esterel synchronous programming language: Design, semantics, implementation," *Sci. Comput. Program.*, vol. 19, no. 2, pp. 87–152, Nov. 1992.

[12] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language Lustre," *Proc. IEEE*, vol. 7, pp. 1305–1320, Sept. 1991.
[13] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Norwell, MA: Kluwer, 1997.
[14] T. Henzinger, C. Kirsch, R. Majumdar, and S. Matic, "Time safety checking for embedded programs," presented at the 2nd Int. Workshop Embedded Software, Grenoble, France, 2002.
[15] C. Kirsch and T. Henzinger, "The embedded machine: Predictable, portable real-time code," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implementation*, 2002, pp. 315–326.
[16] M. Sanvido and W. Schaufelberger, "Design of a framework for hardware-in-the-loop simulation and its application to a model helicopter," presented at the 4th Int. Eurosim Cong., Delft, the Netherlands, 2001.
[17] J. Ledin, "Hardware-in-the-loop simulation," *Embedded Syst. Program.*, vol. 12, no. 2, pp. 42–60, Feb. 1999.

Benjamin Horowitz (Member, IEEE) received the B.A. degree with highest honors in philosophy from Wesleyan University, Middletown, CT, in 1994. From 1995 to 1997, he studied computer science at the University of Massachusetts, Amherst. He is currently a Ph.D. degree candidate in computer science at the University of California, Berkeley.

His research interests include real-time programming languages, scheduling theory, and the design of embedded systems.

Mr. Horowitz is a member of Phi Beta Kappa. He received the Alumni Association Outstanding Achievment Award from the University of Massachusetts, Amherst.

Judith Liebman received the B.S. degree with distinction in electrical engineering from Stanford University, Stanford, CA, in 2000, and the M.S. degree from the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley, in 2002.

Her research interests include embedded software design for safety-critical systems, and her work has been applied to the control system of the Berkeley helicopter uninhabited aerial vehicle.

Ms. Liebman is a member of the engineering honor society Tau Beta Pi. In 2001–2002, she was the president of Women in Computer Science and Electrical Engineering, University of California, Berkeley.

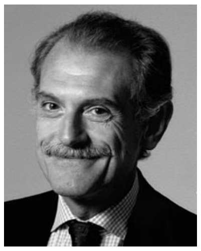
Cedric Ma received the B.S. and M.S. degrees in electrical engineering and computer sciences from the University of California, Berkeley, in 2000 and 2002, respectively.

He is currently an Engineer, Vehicle Systems group, Air Combat Systems business area, Integrated Systems sector, Northrop Grumman Corporation, El Segundo, CA. He works with a core group of control system and fault detection, fault isolation, and recovery experts on various research and development projects contracted from DARPA, other DoD customers, as well as internal customers. As a student of Shankar Sastry, his graduate research interests were the embedded control system for an autonomous helicopter.

T. John Koo received the B.Eng. degree in electronic engineering and the M.Phil. in information engineering from the Chinese University of Hong Kong, Hong Kong, China, in 1992 and 1994, respectively, and the Ph.D. degree in electrical engineering from the University of California, Berkeley, in 2000.

In 1994, he was a Graduate Research Fellow in the Signal and Image Processing Institute of the University of Southern California, Los Angeles. From 1995 to 2002, he was Project Leader in the Berkeley Aerial Robot project, University of California, Berkeley. In 1998, he was a Consultant at the Stanford Research Institute International, Menlo Park, CA. He was a Postdoctoral Fellow in the Department of Electrical Engineering, University of Pennsylvania, Philadelphia, in 2000. In 2001, he was a Research Specialist at the Electronics Research Laboratory, University of California, Berkeley. He is currently a Visiting Faculty Member in the Department of Electrical Engineering and Computer Sciences Department, University of California, Berkeley. His research interests include hybrid systems, embedded software, nonlinear control theory, and soft computing with applications to robotics, power electronics, and networks of autonomous vehicles.

Dr. Koo received the Distinguished M.Phil. Thesis Award of the Faculty of Engineering, the Chinese University of Hong Kong in 1994.



Alberto Sangiovanni-Vincentelli (Fellow, IEEE) received the *Dottore in Ingegneria* degree *summa cum laude* from the Politecnico di Milano, Milano, Italy, in 1971.

From 1980 to 1981, he was a Visiting Scientist at the Mathematical Sciences Department of the IBM T. J. Watson Research Center, Yorktown Heights, NY. In 1987, he was Visiting Professor at the Massachusetts Institute of Technology, Cambridge. From 1976 to the present, he has been on the Faculty of the University of California, Berkeley,where he holds the Edgar L. and Harold H. Buttner Chair of Electrical Engineering and Computer Sciences and the Vice-Chair position for Industrial Relations.

He was a Cofounder of Cadence Design Systems, Inc., San Jose, CA, and Synopsys Inc., Mountain View, CA, the two leading companies in the area of electronic design automation. He is the Chief Technology Adviser of Cadence Design Systems. He is a member of the Board of Directors of Cadence, Softface, Sonics Inc., and Accent, a ST Microelectronics-Cadence joint venture. He is a member of the HP Strategic Technology Advisory Board. He has consulted for a number of U.S. companies including IBM, Intel, ATT, GTE, GE, Harris, DEC, HP; Japanese companies including Kawasaki Steel, where he holds the title of Chief Technology Advisor, Fujitsu, Sony, and Hitachi; and European companies including ST Microelectronics, Alcatel, Daimler-Chrysler, Ericsson, Magneti-Marelli, BMW, Bull. He is the founder and Scientific Director of the Project on Advanced Research on Architectures and Design of Electronic Systems, Rome, Italy, a European group of economic interest supported by Cadence, Magneti-Marelli and ST Microelectronics. He is a member of the Advisory Board of the Lester Center for Innovation of the Haas School of Business, University of California, Berkeley, and of the Center for Western European Studies and a member of the Berkeley Roundtable of the International Economy. He is the author or coauthor of more than 600 papers and 15 books in the area of design tools and methodologies, large-scale systems, and embedded and hybrid systems.

Dr. Sangiovanni-Vincentelli has been a Member of the National Academy of Engineering since 1998. He was the Technical Program Chairperson of the International Conference on Computer Aided Design and its General Chair. He was the Executive Vice-President of the IEEE Circuits and Systems Society. In 1981, he received the Distinguished Teaching Award of the University of California. He has also received the Guillemin-Cauer Award (1982–1983), the Darlington Award (1987–1988), and two awards for the best paper published in the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS and IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN, three best paper awards and one best presentation awards at the Design Automation Conference. He received the worldwide 1995 Graduate

Teaching Award of the IEEE (a Technical Field award for "inspirational teaching of graduate students"). In 1999, he was awarded the CASS Golden Jubilee Medal. In 2001, he was given the prestigious Kaufman Award of the Electronic Design Automation Council for pioneering contributions to electronic design automation. In 2002, he received the Aristotle Award of the Semiconductor Research Corporation given to "faculty whose deep commitment to the educational experience of SRC students has had a profound and continuing impact on their professional performance over a long period of time."

S. Shankar Sastry (Fellow, IEEE) received the M.S. degree (*honoris causa*) from Harvard University, Cambridge, MA, in 1994, and the Ph.D. degree from the University of California, Berkeley, in 1981.

From 1980 to 1982, he was an Assistant Professor at Massachusetts Institute of Technology, Cambridge. In 2000, he was Director of the Information Technology Office at the Defense Advanced Research Projects Agency, Arlington, VA. He is currently the NEC Distinguished Professor of Electrical Engineering and Computer Sciences and Bioengineering and the Chairman of the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley. He has coauthored more than 250 technical papers and has authored, coauthored, or coedited several books, including his latest, *Nonlinear Systems: Analysis, Stability and Control* (New York: Springer-Verlag, 1999). Books on embedded software and structure from motion in computer vision are in progress. He served as Associate Editor for numerous publications, including *Journal of Mathematical Systems, Estimation and Control*, *IMA Journal of Control and Information*, *International Journal of Adaptive Control and Signal Processing*, and *Journal of Biomimetic Systems and Materials.* His research interests are embedded and autonomous software, computer vision, computation in novel substrates such as DNA, nonlinear and adaptive control, robotic telesurgery, control of hybrid systems, embedded systems, sensor networks, and biological motor control.

Dr. Sastry was elected into the National Academy of Engineering in 2001 "for pioneering contributions to the design of hybrid and embedded systems." He also received the President of India Gold Medal in 1977, the IBM Faculty Development award for 1983–1985, the National Science Foundation Presidential Young Investigator Award in 1985, the Eckman Award of the American Automatic Control Council in 1990, the distinguished Alumnus Award of the Indian Institute of Technology in 1999, and the David Marr prize for the best paper at the International Conference in Computer Vision in 1999. He was a chaired Gordon McKay professor at Harvard University, Cambridge, MA, in 1994. He has served as Associate Editor for IEEE TRANSACTIONS ON AUTOMATIC CONTROL, IEEE CONTROL SYSTEMS MAGAZINE, and IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS.