EMBEDDED SOFTWARE DESIGN AND SYSTEM INTEGRATION FOR ROTORCRAFT UAV USING PLATFORMS [1]

Benjamin Horowitz * **Judith Liebman** * **Cedric Ma** * **T. John Koo** *
Thomas A. Henzinger * **Alberto Sangiovanni-Vincentelli** *
Shankar Sastry *

* *EECS, University of California, Berkeley*

Abstract: Automation 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 due to missed timing constraints and subtle transient errors. In this paper, we use the concepts of platform-based design, and the Giotto programming language, to develop a methodology for the design of automation 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 rotorcraft Uninhabited Aerial Vehicle (UAV).

Keywords: helicopter control, autonomous vehicles, control system design, interfaces, programming approaches, real-time systems, time schedule controllers, embedded systems

1. INTRODUCTION

Automation of traditionally human-controlled domains has long been a driving force within the automatic control 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 the capabilities of integrated circuit technology and by advances in control theory. For cost and safety reasons, the CPUs of choice were not top of the line in terms of speed and, since most control applications require real-time responses, the control laws were often "cheap" empirically validated heuristics. Also, software designers used unsound techniques (e.g., communication among tasks using shared variables). 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incorrect software caused severe problems, e.g. the Mars Polar Lander and the Ariane rocket, point out the risks of an outdated methodology.

Automation control systems have several key properties in common:

(1) They operate under stringent real-time constraints.
(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 can utilize legacy code such as device drivers or controllers.

These traits of automation control systems present challenges to system design methodologies. Often,

[1] Research supported in part by DARPA under contract no.

legacy code result in a design overhaul. Furthermore, every design iteration requires exhaustive testing to provide high reliability. Such problems result from the improper coupling of functional design and implementation.

We present a design methodology for embedded controller design via a challenging example of automatic control: a rotorcraft UAV. The difficulty and complexity of the application serves well the purpose of underlining the features of the design method and demonstrating its power. 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. We draw on the principles of *platform-based design* (Sangiovanni-Vincentelli, 2002). A platform, in this context, is a layer of abstraction that hides the unnecessary details of the underlying implementation. We borrow the concept of platform-based design from the electronics industry and tailor it to work with automation control systems. The choice of a software platform to guarantee timing performance is of particular interest. We focus on the Giotto software platform (Henzinger *et al.*, 2001*b*) and show how it aids the development of correct controller software. To quantitatively assess the resulting design, we present a hardware-in-the-loop simulation framework.

The structure of this paper is as follows. In Section 2, we lay the groundwork for the helicopter example. Next, in Section 3, we introduce the reader to the principles of platform-based design. In Section 4, we describe a language for programming time-based controller applications. Finally, in Section 5, we present a rotorcraft UAV design which employs the concepts of the previous three sections.

2. 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 (Section 2.1). We next discuss the first generation flight control system (Section 2.2), and describe some of its limitations (Section 2.3). Finally, we describe what is needed for a second generation system (Section 2.4) to overcome these limitations.

2.1 *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 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, since the helicopter is unstable and dangerous. Moreover, it is difficult to obtain an accurate dynamic model of the helicopter (Koo and Sastry, 1998). In spite of the challenges, the BEAR team managed to build a working flight control system that makes autonomous flight possible.

2.2 *The Flight Control System*

The primary components in the first generation flight system are actuators, sensors, and a control computer. The actuators consist of servomotors controlling the collective pitch, cyclic pitch, throttle, and tail rotor. The primary sensors of the flight control system are as follows:

Inertial Navigation System (INS). The INS consists of accelerometers and rotational rate sensors that provide frequent estimations of the helicopter's position, velocity, orientation, and rate of rotation. Although this estimate is provided at a high rate —roughly 100 Hz— the error in estimate could grow unbounded over time, due to sensor noise and limits in sensor accuracy.

Global Positioning System (GPS). The GPS solves the INS drift problem by providing a position measurement whose error is small —on the order of 1 cm— and bounded over time. However, this accurate measurement is also infrequent —roughly 5 Hz.

An integrated INS-GPS solution uses a Kalman filter to provide frequent updates of the estimated state of the helicopter. This Kalman filter is run by the flight computer, which also computes a control law and sends the result to the actuators.

Experimental system identification was used to obtain a dynamic model of the helicopter: the flight control computer logged the response of the helicopter and the pilot commands, and these logs were compared. The information attained through system identification was then used to synthesize a controller for the helicopter. For more information, refer to (Shim, 2000; Shim *et al.*, 1998).

2.3 *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. The first challenge resulted from a widening choice of devices: as the fleet of helicopters became

provided or received data at different speeds, used different data formats, communicated using different protocols, and so on. 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.

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. Due 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 46 Hz, by the time the actuators used the control data, these data could be 22 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.

2.4 *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:

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, which 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.

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

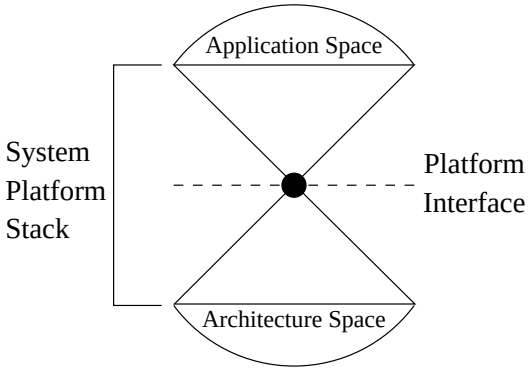

Fig. 1. The system platform stack

run on different helicopters, which may have very different physical dynamics and devices.

3. PLATFORM-BASED DESIGN METHODS

Automation 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*. 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.

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 Figure 1, is the architecture space. This space includes all of the options available for implementing the physical system. 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 Figure 1. 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. 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

4. TIME-BASED CONTROL USING GIOTTO

Giotto is a programming language for implementing time-based control applications. Giotto consists of a formal semantics (Henzinger *et al.*, 2001*b*) and a retargetable compiler. Giotto has already been used to reimplement the control system on board a small autonomous helicopter developed at ETH Zürich (Kirsch *et al.*, 2002). In this section, we discuss the abstraction that Giotto presents to the programmer. The reader wishing a more detailed introduction should consult (Henzinger *et al.*, 2001*a*).

Control applications often have periodic, concurrent tasks. Typically, the periodic tasks communicate with each other. The mechanism used to implement such communication —whether message queues, shared memory, or some other mechanism— may vary depending on the operating system. 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. 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 Figure 2. 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 5 ms, from 20 ms to 25 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 the 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 language constructs which 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. Tasks communicate with each other, and with sensors and actuators, by means of *drivers*, which are shown as diamonds. In Figure 2, 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 which executes between 0 and 10 ms receives its inputs from the `GPS` and `INS` readings at 0 ms. Similarly, the `Control` task which starts at 0 ms receives its inputs from the `Fusion` task which finishes at 0 ms, and writes its outputs to the `Servos` actuator at 5 ms.

5. CASE STUDY: END TO END DESIGN OF ROTORCRAFT UAV

In this section we discuss strategies for building a rotorcraft UAV that keep in mind the goals mentioned in Section 2.4. To achieve these goals we will use the principles of platform-based design presented in Section 3. 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 5.1 the platform-based design principles are used to specify a functional description of the Rotorcraft UAV (RUAV). In Section 5.2 we describe the process of implementing the functional description. Finally, in Section 5.3 we discuss how to assess the implementation.

5.1 *Building Functional Description*

In Section 3 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 RUAV, 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. These two views may be situated 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 RUAV 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 RUAV platform.

Interaction with devices. The RUAV platform should be able to receive transmissions from the sensors at their own rates and without loss of data. Similarly

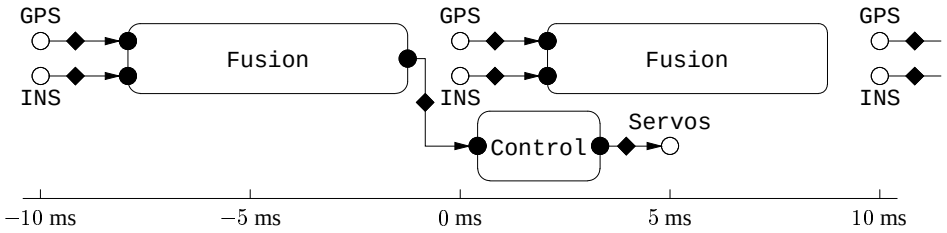

Fig. 2. An example Giotto program

the actuators in the correct formats. It 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.

Interaction with control application. The RUAV 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 5.2 we describe in detail one way to implement the functions of the platform.

5.2 *Implementing Functional Description*

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 above in Section 5.1, places constraints on the platform. Platform implementations which meet these constraints are presented next.

5.2.1. *Implementing the Controller Application* To attain the benefits of time-based control, presented in Section 2.4, the controller application is realized using the Giotto programming language, detailed in Section 4. Section 4 presented a rough sketch of the Giotto implementation in Figure 2. 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 at a frequency of 50 Hz with a deadline time as short as possible to reduce latency. The frequencies of these two tasks are chosen based on the expectations of the control law and on the limitations of the devices.

5.2.2. *Implementing the Rotorcraft Platform* Having considered a realization of the time-based con-

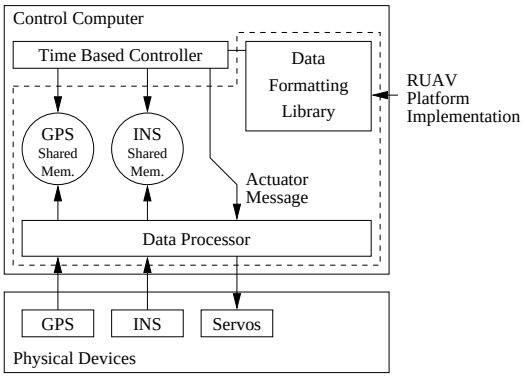

Fig. 3. Implementation of rotorcraft UAV platform

present a possible implementation of the RUAV platform that fulfills the requirements of Section 5.1. It has three main elements that are depicted in Figure 3.

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.

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.

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.

The Giotto 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

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.

5.3 *Measuring Implementation Performance*

Now that we have a platform implementation, it is desirable to evaluate its performance. Ideally, carefully controlled tests could be performed on the physical system. However the fact that we are working with an automation control system makes that a difficult proposition, for two reasons. First, testing is expensive and potentially dangerous. Second, tests are difficult to standardize. For example, the winds and GPS signal strength cannot be controlled. To ameliorate these problems, we propose the use of a hardware-in-the-loop simulator, which allows for the direct testing of the entire control system. 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.

Hardware-in-the-loop simulators (Sanvido and Schaufelberger, 2001; Ledin, 1999) 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. 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 evaluate the platform implementation, 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.

Due to the fact that the simulation computer must imitate a physical system, the simulator must meet two additional constraints. First, the simulator must run in real time. Second, 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 input-output behavior of the hardware-in-the-loop simulator. The proposed simulation framework, in combination with platform-based design, allow for the development of automation control systems that are modular and have guaranteed perfor-

6. CONCLUSION

In this paper, we have presented a design methodology for automation control systems. Our methodology employs platform-based design, which uses layers of abstraction to isolate applications from low-level system details. We also use the programming language Giotto to cleanly implement a time-based controller application. Using our design methodology, we have discussed a redesign of the control system of a rotorcraft UAV. This design goes a long way towards meeting the goals discussed in Section 2.4. 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 automation control systems, for which legacy software, independently engineered subsystems, and strict reliability and timing requirements all play a crucial role.

7. REFERENCES

Henzinger, T.A., B. Horowitz and C.M. Kirsch (2001*a*). Embedded control systems development with Giotto. In: *Proc. of the Intl. Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES '01)*. pp. 64–72.

Henzinger, T.A., B. Horowitz and C.M. Kirsch (2001*b*). Giotto: a time-triggered language for embedded programming. In: *Proc. of the 1st Intl. Workshop on Embedded Software (EMSOFT '01)*. LNCS 2211. Springer-Verlag. pp. 166–184.

Kirsch, C.M., M.A.A. Sanvido, T.A. Henzinger and W. Pree (2002). A Giotto-based helicopter control system (draft).

Koo, T.J. and S. Sastry (1998). Output tracking control design of a helicopter model based on approximate linearization. In: *Proc. 37th Conference on Decision and Control*. pp. 3635–3640.

Ledin, J.A. (1999). Hardware-in-the-loop simulation. *Embedded Systems Programming* **12**(2), 42–60.

Sangiovanni-Vincentelli, A. (2002). Defining platform-based design. *EEDesign of EETimes*.

Sanvido, M.A.A. and W. Schaufelberger (2001). Design of a framework for hardware-in-the-loop simulation and its application to a model helicopter. In: *Proc. of the 4th Intl. Eurosim Congress*.

Shim, D.H. (2000). Hierarchical Flight Control System Synthesis for Rotorcraft-based UAVs. PhD thesis. UC Berkeley.

Shim, D.H., T.J. Koo, F. Hoffmann and S. Sastry (1998). A comprehensive study of control design for an autonomous helicopter. In: *Proc. 37th Conference on Decision and Control*. pp. 3653–3658.