A Giotto-Based Helicopter Control System[⋆]

Christoph M. Kirsch[1], Marco A.A. Sanvido[1], Thomas A. Henzinger[1], and Wolfgang Pree[2]

[1] Department of Electrical Engineering and Computer Sciences
University of California, Berkeley, USA
{cm,msanvido,tah}@eecs.berkeley.edu

[2] Software Research Lab
University of Salzburg, Austria
pree@SoftwareResearch.net

Abstract. We demonstrate the feasibility and benefits of Giotto-based control software development by reimplementing the autopilot system of an autonomously flying model helicopter. Giotto offers a clean separation between the platform-independent concerns of software functionality and I/O timing, and the platform-dependent concerns of software scheduling and execution. Functionality code such as code computing control laws can be generated automatically from Simulink models or, as in the case of this project, inherited from a legacy system. I/O timing code is generated automatically from Giotto models that specify real-time requirements such as task frequencies and actuator update rates. We extend Simulink to support the design of Giotto models, and from these models, the automatic generation of Giotto code that supervises the interaction of the functionality code with the physical environment. The Giotto compiler performs a schedulability analysis on the Giotto code, and generates timing code for the helicopter platform. The Giotto methodology guarantees the stringent hard real-time requirements of the autopilot system, and at the same time supports the automation of the software development process in a way that produces a transparent software architecture with predictable behavior and reusable components.

1 Introduction

We present a methodology for control software development based on the embedded programming language Giotto [3], by implementing the controller for an autonomously flying model helicopter. A Giotto program specifies the real-time interaction of functional components with the physical world as well as among the components themselves. For the helicopter, we isolated the functional components from existing code, specified the timing of the component interaction using a Simulink model and automatically transformed it into a Giotto program. The actual timing code was then generated automatically by the Giotto compiler. The original helicopter system [1] was developed at the ETH Zürich as a customized system based on the programming language Oberon [9,10] and the real-time operating system HelyOS [7]. The reengineering in Giotto introduces a

[⋆] This research was supported in part by the DARPA SEC grant F33615-C-98-3614, the MARCO GSRC grant 98-DT-660, and the AFOSR MURI grant F49620-00-1-0327.

negligible overhead, and at the same time increases the reusability and reliability of the software. We started from a system that already met the desired objectives, i.e., a fully working system with a well-modularized software architecture. By reimplementing the system using Giotto we inductively proved that the Giotto concept is suited for complex control problems and automates the design and implementation of well-engineered control software. This shows that the implementation of difficult control tasks can be significantly simplified by adequate tools and programming languages.

This article begins with a conceptual overview of the Giotto methodology and discusses how Giotto helps to automate control software development. Then, a brief overview of the helicopter system is given. Next, the helicopter control software is presented in two steps. We first introduce Giotto's core constructs by means of Simulink's visual syntax, and then give their translation into Giotto's textual syntax. Finally, the paper discusses the compilation and execution of the resulting Giotto control system.

2 Overview of the Giotto Methodology

The goal of Giotto is to provide the high-level, domain-specific abstractions that allow control engineers and control software developers to focus on the control system aspects instead of the platform. By platform, we mean the specific hardware and operating system on which the control system runs. A Giotto program specifies the control system's *reactivity*, which is checked for its *schedulability* by the compiler. The term reactivity expresses what we mean by control system aspects: the system's functionality, in particular, the control laws, and the system's timing requirements. The term schedulability expresses what we mean by platform-dependent aspects, such as platform performance, platform utilization (scheduling), and fault tolerance. The Giotto programmer specifies reactivity; the Giotto compiler checks schedulability for a specific platform, and generates timing code for the platform. *Timing code* determines when a sensor is read or an actuator is updated as well as when a control law computation is invoked. *Functionality code* implements the actual sensor reading, actuator update, and control law computation. Functionality code must be written in a programming language such as C or Oberon. Timing and functionality code are executed on a runtime system that consists of a virtual machine called the *Embedded Machine* [4] and a real-time operating system. The timing code, also called *E code*, is interpreted by the Embedded Machine whereas the functionality code is scheduled for execution by the operating system's scheduler. The scheduling scheme of the operating system and the schedulability test of the Giotto compiler must be compatible.

The separation of reactivity and schedulability implies a shift in control software development away from low-level, platform-dependent implementation details towards high-level, domain-specific issues. This is analogous to high-level general-purpose programming languages, which abstract from the details of the underlying hardware. In this sense Giotto represents a high-level programming language for embedded control systems. Developers of such systems benefit from the separation of concerns in several ways. First, the development effort is significantly reduced, as the tedious programming of the timing code is handed over to the compiler. Also, the automation of the timing code implementation eliminates a common source of errors. Second, a Giotto program

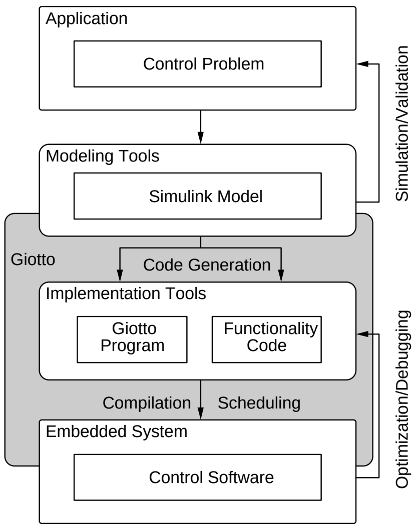

Fig. 1. The Giotto-based development process

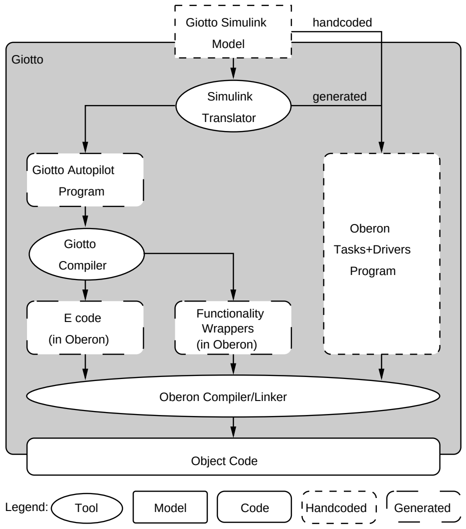

Fig. 2. The Giotto tool chain

specifies reactivity in a modular fashion, which facilitates the exchange and addition of functionality. Also, functionality code can be packaged as software components and reused. Third, the system can easily be ported to other platforms for which an implementation of the Embedded Machine is available. Fourth, the reactive properties of a control system specified by a Giotto program (functionality and timing) can be subject to formal verification.

2.1 Giotto-Based Control-Systems Development

The traditional and the Giotto-based development of control systems is aided by *modeling tools* and *implementation tools* as shown in Figure 1. A modeling tool such as MathWorks' Simulink supports the development of a controller model that solves a given control problem. The controller model can be simulated and validated with respect to the requirements of the control problem. In this step the platform constraints such as CPU performance and code size are ignored. The design at this step is therefore *solution-oriented* and *platform-independent*. Once a controller model has been validated, a controller program that implements the model is generated. The functionality code of the controller program may be generated automatically by code generators for Simulink. The timing code, on the other hand, is typically hand-written. Then, implementation tools such as compilers and debuggers support the implementation and optimization of the controller program under platform-dependent constraints. Due to restricted platform performance, a controller program often needs to be optimized with respect to platform-dependent properties such as execution speed or size in order to implement the controller model correctly. Without Giotto the close correspondence between the controller model and program is typically lost in the optimization step, in particular, if it is done by hand: functionality code and timing code are often not reusable on a different platform or for a modified model. With Giotto the correspondence is maintained

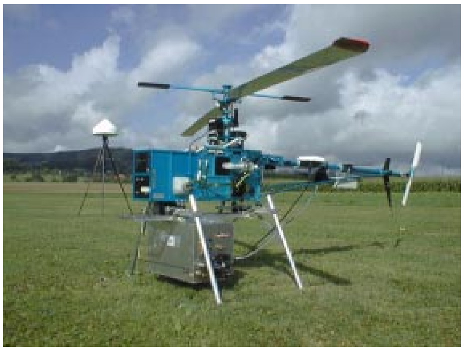

Fig. 3. The OLGA helicopter

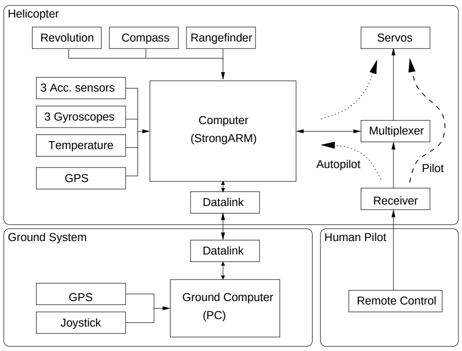

Fig. 4. The OLGA ECS structure

because timing aspects such as the real-time frequencies of controller tasks are specified in a platform-independent way already in the controller model: Giotto decreases the gap between model and program. Then, a Giotto program is automatically generated from the model. In a final step, the Giotto compiler generates the timing code from the Giotto program for a given platform.

2.2 The Giotto Tool Chain

Figure 2 shows the Giotto tool chain used for the reengineering of the helicopter autopilot system. The developer starts by specifying a control model using the Giotto constructs made available in Simulink. The control model specifies the I/O timing and the timing of the interaction between functional units such as tasks, which perform computation, and drivers, which interface tasks with the hardware. Tasks and drivers are functionality code written in Oberon that we have reused from the original autopilot implementation. In principle, most of the functionality code could have been generated automatically by a Simulink code generator. From the control model in Simulink, we obtain executable code in two steps. First, a Giotto translator tool generates a Giotto program from the model. Second, the Giotto compiler checks the schedulability of the Giotto program on a given platform based on worst-case execution times for tasks and drivers, and then generates two Oberon programs: the E code (timing code), and so-called functionality wrappers that interface tasks and drivers with E code. Finally, the Oberon code is compiled and linked into an executable that is guaranteed to exhibit the same timing behavior as the original controller model. The same tool chain can be used if Oberon is replaced by another programming language and HelyOS by another RTOS. For example, the Giotto tools also support C and OSEKWorks [19].

3 The Helicopter System

The original helicopter system [1] was developed at ETH Zürich as part of an interdisciplinary project to build an autonomously flying model helicopter for research purposes.

The helicopter is a custom-crafted model with a single-CPU (StrongARM-based) control platform that was also developed at ETH Zürich [1]. All functional components are implemented in the programming language Oberon [9,10] on top of the custom-designed real-time operating system HelyOS [7].

The OLGA system (for *Oberon Language Goes Airborne*) consists of an aircraft, i.e., the model helicopter and a ground system. Figure 3 shows a picture of the helicopter; Figure 4 shows the system structure. The ground system (bottom of Figure 4) supports mission planning, flight command activation, and flight monitoring. As this part of the system is not relevant for the implementation of the autopilot system, it is not discussed here. All sensors for navigation purposes (except the GPS receiver used for the differential GPS) and the computational power needed for navigation and flight control are airborne. The sensors used on the helicopter are a GPS receiver, a compass, a revolution sensor, a laser altimeter (range finder), three accelerometers, three gyroscopes, and a temperature sensor. Note that the arrow on the right of Figure 4 labeled *Pilot*, which connects the boxes *Receiver* and *Servos*, represents the alternative control by a human pilot. It is required as a back-up for safety reasons. A human pilot is able to remotely switch to fully manual mode at any time during operation, short-cutting OLGA in case of any malfunctions.

The complexity of helicopter flight control results from the number of different sensors and actuators the system has to handle concurrently, the difficulty in flying the helicopter, and the limitations of the autopilot system (electrical consumption, limited computational power, vibrations, jitter, etc.). Moreover, the helicopter is a dangerous and expensive platform, where a trial-and-error approach cannot be used. The control and navigation algorithms are based on hard real-time assumptions that have to be guaranteed under all circumstances by the implementation. In our specific autopilot example, the controller and navigation frequency were chosen to run at 40 Hz. The computational power required for each step was in the worst case 12 ms, which gave a CPU utilization for control and navigation of more then 45%, leaving not much for the housekeeping activities such as background and monitoring. The complexity of the problem is evidenced by the fail/success rate, i.e., the number of research projects on autonomously flying helicopters, and the number of such projects that have not managed to implement a fully working system. In most cases the failure was not due to financial or human-resource constraints, but due to the complexity of the implementation itself. Simulating an autopilot system is relatively easy (look at the number of master's theses on this subject), but turning it into a working system is not. And turning it into a clean, structured, and well-engineered system (both hardware and software) is even harder. The references [11,12,13,14,15,16,17,18] are well-known ongoing academic helicopter projects. In [2] an overview of all major autonomous model helicopter projects is given.

4 The Autopilot Software

The autopilot system has six different modes of operation (see Figure 5). In each mode different tasks are active. The modes are `Init`, `Idle`, `Motor`, `TakeOff`, `ControlOff`, and `ControlOn`. The first three modes are needed in order to correctly handle the initialization procedure. The `Motor` and `TakeOff` modes handle the

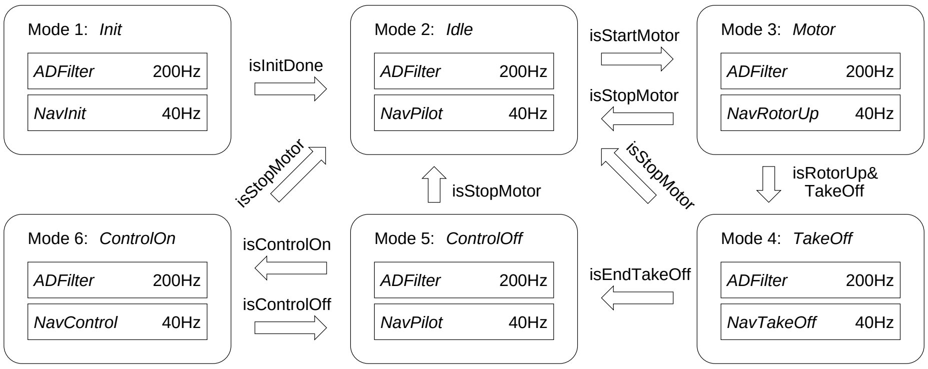

Fig. 5. The operating modes of the controller

transition from a 0 rpm rotor speed to a safe speed of 300 rpm. At this speed the helicopter is guaranteed not to take off, and only an active command from the ground station allows the transition to mode `TakeOff`. When the take-off procedure is finished, the helicopter is in mode `ControlOff`. In this mode, the rotor is at a nominal speed of 1200 rpm and the pilot has full control over the helicopter. At this point, the pilot is able to switch, at any time, to the `ControlOn` mode, activating the autopilot. For simplicity, we will henceforth focus only on the two modes `ControlOff` and `ControlOn`. In the `ControlOff` mode, a human pilot controls the helicopter, whereas in the `ControlOn` mode, the helicopter operates autonomously. The `ControlOff` mode consists of the 200 Hz task `ADFilter` and the 40 Hz task `NavPilot`. The `ADFilter` task decodes and preprocesses sensor values. The `NavPilot` task keeps track of the helicopter's position and velocity using the preprocessed data from the `ADFilter` task, and translates pilot commands received via the wireless link into servo commands. The `ControlOff` mode switches to the `ControlOn` mode if the pilot pushes a button on the remote control. Besides the 200 Hz task `ADFilter`, the `ControlOn` mode has the 40 Hz task `NavControl`, which replaces the `NavPilot` task. Besides keeping track of position and velocity this task implements the controller that stabilizes the helicopter autonomously. The `ControlOn` mode switches back to the `ControlOff` mode if the pilot pushes a take-over button on the remote control.

4.1 A Simulink Specification of the Giotto Model

We have extended Simulink with the capability of expressing Giotto models. A Giotto model in Simulink specifies the real-time interaction of its components with the physical world. All components of a Giotto model execute periodically. A Giotto model has a single parameter that specifies the hyper-period of the components. The hyper-period is the least common multiple period of all component periods. As an example consider Figure 6, which shows the Simulink specification of a Giotto model called `helicopter controller`, which is connected to a continuous-time model of the helicopter dynamics. The dynamics block contains only standard continuous-time Simulink blocks,

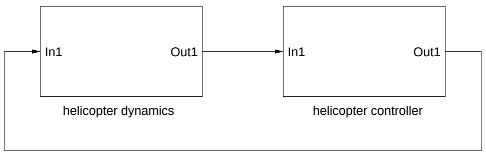

Fig. 6. The Giotto helicopter model in Simulink

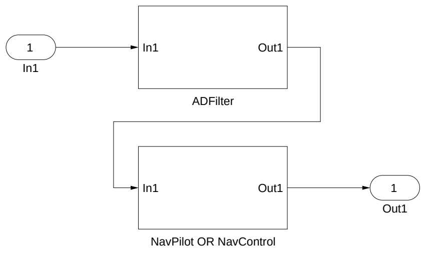

Fig. 7. The Giotto tasks in Simulink

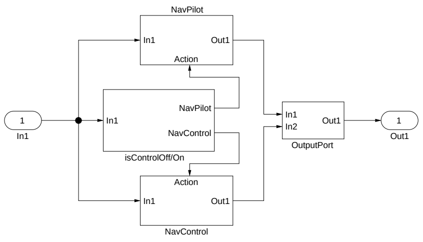

Fig. 8. The Giotto case block in Simulink

whereas the controller block is a so-called *Giotto model block*, and exhibits Giotto semantics. The controller block contains a Giotto model that has two modes of operation: human control (`ControlOff`) and autonomous flight (`ControlOn`). The helicopter controller has a hyper-period of 25 ms.

Figure 7 shows the contents of the `helicopter controller` block. The block labeled `ADFilter` is a *Giotto task block*, which represents a single Giotto task — the basic functional unit in Giotto. A Giotto task is a periodically executed piece of Oberon code. In the example, the `ADFilter` block contains only standard discrete-time Simulink blocks that implement the decoding and preprocessing of sensor values. The second block in the Giotto model is an example of a *Giotto case block*, which may contain multiple Giotto tasks. Upon each invocation a case block picks exactly one of its tasks to execute. In the example, the case block contains the Giotto task blocks `NavPilot` and `NavControl`. The `NavPilot` task computes the helicopter position and velocity, and reads pilot commands from which it produces the correct servo values. Thus, every time the `NavPilot` task executes, the human pilot has full control of the helicopter. The `NavControl` task, by contrast, implements autonomous flight: it also computes position and velocity, but produces the servo values based on a control law computation. Each case block has a frequency given as an integer value relative to the hyper-period of the Giotto model. Here the case block has a frequency of 1, i.e., it executes with a period of 25ms. Both tasks in the case block inherit that frequency. Note that the `ADFilter` task block in the Giotto model is actually an abbreviation for a case block containing a single task. In fact, Giotto model blocks may contain case blocks only. The virtual case block around the `ADFilter` task has a frequency of 5, which means that the task runs five times per 25 ms, i.e., with a period of 5 ms.

Figure 8 shows the contents of the case block. Besides the two task blocks, there is a *Giotto switch block*, labeled `isControlOff/On`. A Giotto switch block may contain any standard discrete-time Simulink blocks in order to determine, based on its input port values, at least one task that gets to execute. If no task is chosen to execute all previous output port values are held. The `isControlOff/On` block reads a pilot command to switch from manual to autonomous mode and back. In our example, it always chooses between the `NavPilot` task and the `NavControl` task. The switch block is evaluated once for each invocation of the surrounding case block at the beginning of its period. Thus it is evaluated once every 25 ms. Note that the block labeled `OutputPort` is necessary only for connecting the outputs of the two tasks to a single output port. Moreover, the tasks and the switch block in a case block may only read from the input ports of the case block but not from any task output ports in that block. In order to do this one has to establish a link outside of the case block from an output port to an input port.

The time-triggered semantics of Giotto enables efficient reasoning about the timing behavior of a Giotto model, in particular, whether it conforms to the timing requirements of the control design. Moreover, Giotto models are compositional in the sense that any number of Giotto models may be added side by side without changing their individual semantics. For example, additional functionality can be added to the helicopter controller without changing the real-time behavior of the controller. This, of course, assumes the provision of sufficient computational resources, which is checked by the Giotto compiler for a specified platform. In order to simulate Giotto models in Simulink we have developed a translator that reads Simulink specifications of Giotto models and transforms them into standard discrete-time multi-rate Simulink models. The tool also generates the corresponding Giotto programs in textual form, which can then be processed by the Giotto compiler for schedulability analysis and to generate code.

4.2 The Giotto Program

Giotto defines the exact timing and communication between a Giotto program and its environment as well as among Giotto tasks. For this purpose, a Giotto program needs to make explicit semantical details that are left implicit or unspecified in the Simulink specification of a Giotto model. In order to transport values between ports, Giotto uses the concept of drivers. We distinguish *Giotto task*, *actuator*, and *mode drivers* from *Giotto device drivers*. The purpose of a Giotto task and actuator driver is to transport values from sensors and task output ports to task input ports and actuators, respectively; a Giotto mode driver evaluates a mode-switch condition and, if it evaluates to true, transports initial values to task output ports of the target mode; a Giotto device driver transports values from a hardware device or a non-Giotto task to a port or vice versa. In the Simulink specification of a Giotto model task and actuator drivers exist only implicitly as links, while Giotto device drivers are absent entirely. However, device drivers are required in a complete implementation of a Giotto program to link it to the hardware or non-Giotto tasks such as low-level event-triggered or non-time-critical tasks.

From a Giotto model block in Simulink, we generate a *Giotto program*, which is a collection of Giotto modes. Each *Giotto mode* has a hyper-period, a set of task invocations with specified frequencies, a set of actuator updates with specified frequencies, and a set of mode switches with specified frequencies. A task invocation executes the task driver followed by the task, an actuator update executes the actuator driver, and a mode switch evaluates a mode driver, possibly followed by a switch to the target mode. The following

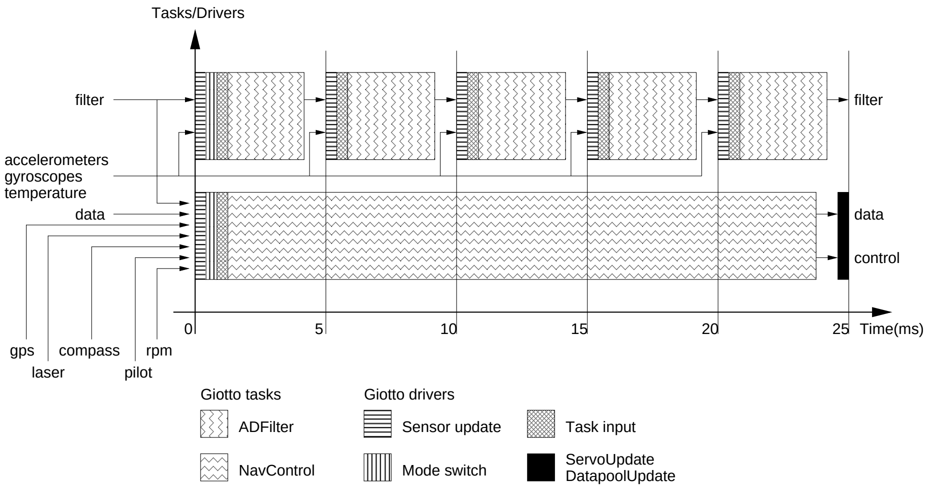

Fig. 9. The logical execution of the Giotto program in the `ControlOn` mode

example shows the Giotto program `helicopter controller`, which specifies the `ControlOff` and `ControlOn` modes:

```
mode ControlOff() period 25 {
  actfreq  1 do ServoUpdate;
  actfreq  1 do DataPoolUpdate;
  exitfreq 1 do ControlOn;
  taskfreq 5 do ADFilter;
  taskfreq 1 do NavPilot;}
mode ControlOn() period 25 {
  actfreq  1 do ServoUpdate;
  actfreq  1 do DataPoolUpdate;
  exitfreq 1 do ControlOff;
  taskfreq 5 do ADFilter;
  taskfreq 1 do NavControl;}
```

The hyper-period of both modes is 25 ms. The frequency of the tasks, mode switches, and actuator updates is specified relative to this period. For example, the `ADFilter` task runs in both modes five times per 25 ms, i.e., at 200 Hz. The helicopter `servos` and the `datapool`, which contains messages that are sent to the ground station, are updated once every 25 ms by invocations of the Giotto actuator drivers `ServoUpdate` and `DataPoolUpdate`, respectively. Figure 9 shows the logical execution of a single hyper-period of the `ControlOn` mode (the actual execution is shown in Figure 11 and will be discussed later). Logically, the `ADFilter` task runs five times exactly for 5 ms, while the `NavControl` task runs once exactly for 25 ms. Note that all Giotto drivers are executed in logical zero time. While a Giotto task represents *scheduled* computation on the application level and consumes logical time, a Giotto driver is *synchronous*, bounded code, which is executed logically instantaneously on the system level (since drivers cannot depend on each other, no issues of fixed-point semantics arise).

Intertask communication as well as communication from and to sensors and actuators works through *Giotto ports*. In the Giotto program we declare all sensor ports globally as follows:

```
sensor
  GPSPort     gps uses            GPSGet;
  LaserPort   laser uses          LaserGet;
  CompassPort compass uses        CompassGet;
  RPMPort     rpm uses            RotorGet;
  ServoPort   pilot uses          ServoGet;
  AnalogPort  accelerometers uses AccGet;
  AnalogPort  gyroscopes uses     GyrosGet;
  AnalogPort  temperature uses    TempGet;
  BoolPort    startswitch uses    StartSwitchGet;
  BoolPort    stopswitch uses     StopSwitchGet;
```

Besides a type name, we declare a Giotto device driver for each sensor port. For example, the sensor port `gps` has the type `GPSPort` and uses the Giotto device driver `GPSGet` to get new sensor values from the GPS device. Types and device drivers are implemented externally to Giotto. Here they are Oberon types and procedures. In Figure 9, at the 0 ms instant, the first action is to read the latest sensor values by calling the Giotto device drivers for all Giotto sensor ports. Subsequently, every 5 ms until the end of the hyper-period, the device drivers are called only for the sensor ports that are read by the `ADFilter` task. Giotto device drivers are always called in a time-triggered fashion. However, some devices require immediate attention using an event-triggered (interrupt-driven) driver. For example, the `GPSGet` device driver does not access directly the GPS device but a buffer in which an interrupt handler that is bound to the GPS device places the latest GPS readings. The interrupt handler is external to the Giotto program. The opposite direction for communication from a port to a device is done in a similar way and will be discussed below. At the 0 ms instant right after executing the Giotto device drivers for the sensor ports, the Giotto mode driver is called to determine whether to switch into the `ControlOff` mode or not. The driver is declared as follows:

```
driver ControlOff(stopswitch) output () {
  switch isControlOff(stopswitch)}
```

The driver has a single driver input port `stopswitch`, which is a globally declared sensor or task output port. In this case it is a sensor port whose Giotto device driver `StopSwitchGet` has just been called. The device driver reads the pilot switch `AutopilotOn/Off` from an FPGA, which decodes the pilot switch position transmitted via the wireless link from the remote control to the airborne system. Based on the value of the `stopswitch` port, the Oberon implementation of the `isControlOff` predicate returns true or false determining whether to switch to the `ControlOff` mode or not. Suppose that we stay in the `ControlOn` mode. The next step is to load the task input ports of the `ADFilter` and `NavControl` tasks with the latest values of the sensor and task output ports to which the tasks are connected as specified in the task declarations below. Before declaring the tasks all task output ports are declared globally as follows:

```
output
  AnalogPort    filter  :=  FilterInit;
  ServoPort     control :=  ServoInit;
  DataPoolPort  data    :=  DataPoolInit;
```

The `filter` port is the only task output port of the `ADFilter` task. The `control` and `data` ports are the task output ports of the `NavControl` task. For each task output port, in addition to the type, an initialization driver is specified, which is invoked once at start-up time to initialize the port. Here the initialization drivers are implemented by Oberon procedures. Initial values for all task output ports sufficiently describe a unique start configuration of a Giotto program. Then, for a given behavior of the sensors, a Giotto program computes a *deterministic* trace of actuator values, provided all tasks meet their deadlines [3]. The ADFilter and NavControl tasks are declared as follows:

```
task ADFilter(accelerometers, gyroscopes, temperature, filter)
  output (filter) {
  schedule ADFilterImplementation(accelerometers, gyroscopes,
                                  temperature, filter)}
task NavControl(gps, laser, compass, filter, rpm, pilot, data)
  output (control, data) {
  schedule NavControlImplementation(gps, laser, compass, filter,
                                    rpm, pilot, control, data)}
```

The ADFilter task reads from the accelerometers, gyroscopes, temperature, and filter ports. The filter port is also a task output port, which makes the port a state variable of the task. Prior to the invocation of the task, the values of all four ports are copied by a task driver to some local memory, which is only accessible to the task itself. The task driver does not have to be declared explicitly. The Oberon ADFilterImplementation corresponds to the functional part of the ADFilter implementation in the original OLGA system. The NavControl task is declared in a similar way. Now, the Giotto program is ready to invoke the ADFilter and NavControl tasks. The NavControl task runs logically for 25 ms, while the ADFilter task finishes after 5 ms. Then, new sensor values are read and the task input ports of the ADFilter task are loaded, before invoking the task again. This process repeats until the 25 ms time instant is reached. At that time instant new values in the control and data ports of the NavControl task are available. The new values are now transferred by the Giotto actuator drivers ServoUpdate and DataPoolUpdate to the servos and datapool actuator ports, respectively. In order to declare the actuator drivers we first need to declare the actuator ports globally as follows:

```
actuator
  ServoPort    servos   uses ServoPut;
  DataPoolPort datapool uses DataPoolPut;
```

Besides a type name, we declare a Giotto device driver for each actuator port. For example, the actuator port servos has the type ServoPort and uses the Giotto device driver ServoPut to transfer new actuator values to the helicopter servos. Again, types and device drivers are implemented externally to Giotto. Before the device drivers are called, the actuator drivers ServoUpdate and DataPoolUpdate are executed. The actuator drivers are declared as follows:

```
driver ServoUpdate(control) output (servos) {
  call ServoUpdateImplementation(control, servos)}
driver DataPoolUpdate(data) output (datapool) {
  call DataPoolUpdateImplementation(data, datapool)}
```

In Figure 9, at the 25 ms instant after the NavControl task finished, the helicopter servos and datapool are updated by first executing the Oberon ServoUpdateImplementation and DataPoolUpdateImplementation, which transport the values from the control and data ports to the servos and datapool ports, respectively. Then, the `ServoPut` device driver is called, which takes the new value from the `servos` port and then directly updates the servo devices. The `DataPoolPut` device driver is also called but instead of accessing a device it puts the value from the `datapool` port into a buffer, which gets transmitted over the wireless link as soon as the Giotto system becomes idle. The actual transmission is done by the asynchronous message handler task of the original OLGA system. This task is external to the Giotto program. The 25 ms hyper-period is now finished. In the next section, we will discuss the execution of the Giotto program and present the actual task schedule of the program.

5 The Giotto Compiler and Execution Environment

The Giotto-based implementation of the autopilot system has the same functionality as the original system but uses the Giotto programming language for the explicit specification of real-time requirements and inter-task communication. The autopilot functions, i.e., the navigation and control tasks, are released from any timing or scheduling code but otherwise correspond to their original OLGA implementations. The system architecture of the Giotto-based system is shown in Figure 10. The upper left portion shows the Giotto program, including the Oberon implementation of the Giotto device drivers as well as the Giotto tasks and drivers. The non-Giotto tasks shown in the upper right portion of Figure 10 implement event-triggered or non-time-critical tasks, which are interfaced to the Giotto system through Giotto device drivers. Event-triggered tasks must be taken into account by the schedulability analysis performed by the Giotto compiler; background tasks are performed only when the Giotto system is idle. In the middle of Figure 10, the original OLGA software system is shown extended by an implementation of the Embedded Machine [4] in the kernel of the HelyOS real-time operating system. The software system runs on the OLGA computer system.

A Giotto program does not specify where, how, and when tasks are scheduled. For example, the helicopter-control program can be compiled on platforms that have a single CPU (by time sharing the tasks) as well as on platforms with two CPUs (by parallelism); it can be compiled on platforms with preemptive priority scheduling (such as most real-time operating systems) as well as on truly time-triggered platforms. The mapping from the Giotto program to executable code for the helicopter platform is performed by the Giotto compiler. The Giotto compiler needs to ensure that the logical semantics of Giotto —functionality and timing— is preserved. The compiler targets the Embedded Machine, which interprets the generated E code in real-time. The E code instructions provide a portable API to the underlying RTOS. There are E code instructions to call or schedule the native implementation of tasks and drivers, respectively, as well as instructions to invoke the Embedded Machine at specific time instants or occurrences of events. The generated E code implements the logical semantics of the Giotto program provided the E code is *time-safe* [4], which intuitively means that all tasks meet their deadlines. The compiler performs a schedulability analysis by checking time safety of the given Giotto program [5] for given worst-case execution times of the tasks and drivers. Time-safe E code programs are *environment-determined* [4], which means that, for any given behavior of the physical world seen through the sensors, the E code computes a deterministic trace of actuator values at deterministic time instants. In other words, a

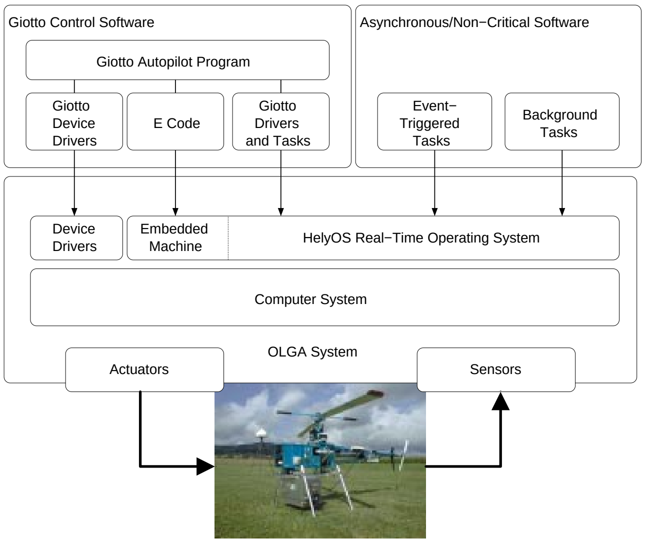

Fig. 10. The Giotto-based autopilot system

Giotto system exhibits no internal race conditions, which makes the behavior of Giotto systems predictable and verifiable.

Figure 11 shows the actual execution of the Giotto program from the previous section, as specified by the generated E code. The top row shows the execution of the `ADFilter` task and the drivers from the top row in Figure 9. The middle row shows the execution of the `NavControl` task and the drivers from the bottom row in Figure 9; note that the `NavControl` task is physically preempted. The bottom row shows the execution of background tasks. The Giotto compiler generates E code that accesses sensors and actuators as close as possible to the specified time instants in order to reduce I/O jitter. The existing I/O behavior of the system would not change if we were to add more Giotto tasks, provided the compiler succeeds in showing the resulting E code to be time-safe despite the additional load.

6 Conclusion

The successful reengineering of the OLGA system using Giotto shows the feasibility of the Giotto approach for high-performance, hard real-time embedded control systems. The Giotto compiler automatically generates timing code for a system with multiple modes of operation, multiple levels of task priorities, and time-triggered as well as event-triggered task activation. Giotto implies an overhead through predicate checks, calls of wrapper functions, and the copying of ports. Measurements have shown that this amounts to less than 2% of the 25 ms period, which is easily acceptable for helicopter flight control. The implementation of the Embedded Machine on top of HelyOS was accomplished in one week, and its source code is only 6 KB. Embedded control systems that are based on Giotto can expect a high degree of modularization. In particular, in

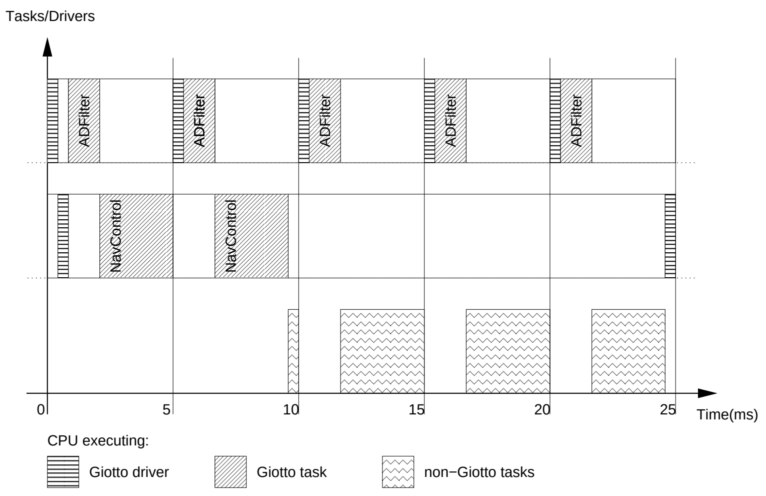

Fig. 11. The actual execution of the Giotto program in the `ControlOn` mode based on rate-monotonic scheduling

the helicopter control system, the timing concerns and the inter-process communication are decoupled from the navigation and control algorithms. The Giotto-based autopilot system is nonetheless functionally comparable to systems that use considerably more complex software architectures [8]. The functionality and reliability of the original ETH Zürich helicopter system has been proven by many flight tests. The Giotto-based reimplementation inherits the same functionality but with increased software reusability, flexibility, and transparency. The reimplementation was shown to be correct and working by means of on-the-ground tests. No actual on-flight tests have been done, not for technical reasons but for reasons outside our control. A similar reengineering approach has been used to assess the feasibility of other methodologies, for example, Meta-H has been applied to embedded missile control systems [6].

The case study has had substantial impact on the ongoing development of the Giotto concept. In particular, it has guided the refinement of Giotto from a theoretical to a practical language for embedded control software. For example, the precise interaction between Giotto ports (a concept of the formal Giotto semantics) and actual devices (such as sensors and actuators) needed much elaboration and is made concrete through the concept of Giotto device drivers. Second, while the formal Giotto semantics is purely time-triggered, the helicopter system shows how Giotto can, within its semantical requirements, interact with asynchronous events, such as communication from the ground station. On the other hand, the helicopter system has only a single CPU, and our future focus is therefore on extending the Giotto approach to distributed platforms.

Acknowledgments

We thank Niklaus Wirth and Walter Schaufelberger for their advice and support of the reengineering effort of the ETH Zürich helicopter system using Giotto.

References

1. J. Chapuis, C. Eck, M. Kottmann, M. Sanvido, and O. Tanner. Control of helicopters. In *Control of Complex Systems*, pages 359–392. Springer Verlag, 1999.
2. C. Eck. *Navigation Algorithms with Applications to Unmanned Helicopters*. PhD thesis 14402, ETH Zürich, 2001.
3. T.A. Henzinger, B. Horowitz, and C.M. Kirsch. Giotto: a time-triggered language for embedded programming. In *Proc. First International Workshop on Embedded Software (EMSOFT)*, LNCS 2211, pages 166–184. Springer Verlag, 2001.
4. T.A. Henzinger and C.M. Kirsch. The Embedded Machine: predictable, portable real-time code. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 315–326. ACM Press, 2002.
5. T.A. Henzinger, C.M. Kirsch, R. Majumdar, and S. Matic. Time safety checking for embedded programs. In *Proc. Second International Workshop on Embedded Software (EMSOFT)*, LNCS. Springer Verlag, 2002.
6. D.J. McConnel, B. Lewis, and L. Gray. Reengineering a single-threaded embedded missile application onto a parallel processing platform using MetaH. *Real-Time Systems*, 14:7–20, 1998.
7. M. Sanvido. *A Computer System for Model Helicopter Flight Control; Technical Memo 3: The Software Core*. Technical Report 317, Institute for Computer Systems, ETH Zürich, 1999.
8. L. Wills, S. Kannan, S. Sander, M. Guler, B. Heck, V.D. Prasad, D. Schrage, and G. Vachtsevanos. An open platform for reconfigurable control. *IEEE Control Systems Magazine*, 21:49–64, 2001.
9. N. Wirth. *A Computer System for Model Helicopter Flight Control; Technical Memo 6: The Oberon Compiler for the StrongARM Processor*. Technical Report 314, Institute for Computer Systems, ETH Zürich, 1999.
10. N. Wirth and J. Gutknecht. *Projekt Oberon: The Design of an Operating System and Compiler*. ACM Press, 1992.
11. http://www.cs.cmu.edu/afs/cs/project/chopper/www/heli_project.html. The Robotics Institute, Carnegie Mellon University.
12. http://controls.ae.gatech.edu/labs/uavrf/. The UAV Lab, Georgia Institute of Technology.
13. *Aerial Robotics*. http://gewurtz.mit.edu/research/heli.htm. Laboratory for Information and Decision Systems, Massachusetts Institute of Technology.
14. *Autonomous Flying Vehicles*. http://www-robotics.usc.edu/˜avatar. Robotics Research Laboratory, University of Southern California.
15. *Autonomous Helicopter Project*. http://www.heli.ethz.ch. Measurement and Control Laboratory, ETH Zürich.
16. *BEAR: Berkeley Aerobot*. http://robotics.eecs.berkeley.edu/bear. Electronic Research Laboratory, University of California at Berkeley.
17. *The Hummingbird Helicopter*. http://sun-valley.stanford.edu/˜heli. Aerospace Robotics Laboratory, Stanford University.
18. *Marvin*. http://pdv.cs.tu-berlin.de/MARVIN. Institute for Technical Computer Science, Technische Universität Berlin.
19. *OSEKWorks Operating System*. http://www.windriver.com/products/html/osekworks.html WindRiver.