From Control Models to Real-Time Code Using Giotto

©JENE NELSON AT CN GRAPHICS INC.

Software-Enabled Control

We present a methodology for control software development based on Giotto [2], a domain-specific high-level programming language for control applications. A Giotto program explicitly specifies the exact real-time interaction of software components with the physical world. The Giotto compiler automatically generates timing code that ensures the specified behavior on a given platform. This article illustrates the Giotto methodology by reimplementing the controller for an autonomously flying model helicopter originally developed at the Swiss Federal Institute of Technology (ETH) Zürich [3]. We demonstrate that Giotto introduces a negligible overhead and at the same time increases the reliability and reusability of the control software.

The article begins with a conceptual overview of the Giotto methodology. We explain how Giotto helps to automate the control software development process and to improve the quality of the resulting code. We then report on the Giotto helicopter project. We use the autopilot software for the helicopter to guide an informal presentation of the Giotto language and of the Simulink/Giotto (S/G) translator, which extracts a Giotto program from a Giotto control model specified in Simulink. This is followed by a brief dis-

By Thomas A. Henzinger, Christoph M. Kirsch, Marco A.A. Sanvido, and Wolfgang Pree

cussion of the compilation and execution of the Giotto-based control system. The article concludes with a summary of available Giotto implementations and pointers to related work.

The Giotto Methodology

The Giotto methodology presents a systematic attempt to decompose the difficult task of control systems development: domain-specific (control) concerns are separated from platform-specific (implementation) concerns, and time-related concerns are separated from data-related (functionality) concerns.

Separating Reactivity from Scheduling

Traditionally, a control system is designed using tools for mathematical modeling and simulation, such as MathWorks' Simulink. The control model is implemented manually or automatically, and the code is then tested and optimized for the given platform until it exhibits satisfactory timing behavior. In the process, the tight correspondence between model and code is often lost. The resulting

More details on Giotto, as well as a distribution of the Giotto software tools used in this article, can be found at http://www.eecs.berkeley.edu/~fresco/giotto.

Sanvido (msanvido@eecs.berkeley.edu), Henzinger, and Kirsch are with the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, 94720, U.S.A. Pree is with the Software Research Laboratory, University of Salzburg, Austria.

software is brittle, difficult to reuse on a different platform, and difficult to enhance with additional functionality.

Giotto addresses this problem by offering an intermediate layer of abstraction between the mathematical model and the code. We call this layer an embedded software model. An embedded software model specifies a solution to a given control problem independent of an execution platform (i.e., operating system and hardware), but it is closer to executable code than a mathematical model. Whereas the entities of a mathematical model are typically matrices, equations, and perhaps state diagrams, the entities of a software model are data structures and procedures. For example, a software model may specify the representation for storing a matrix and the algorithm and precision for evaluating an equation.

Common software models are high-level programming languages, but for embedded software, the model needs to include constructs for expressing concurrency and timing. An embedded software model specifies the logical concurrency and interaction of software processes, as well as the timing of process interactions with the physical environment. However, an embedded software model must not specify the physical distribution of the software processes, or a scheduling mechanism, both of which require knowledge about the platform. In Giotto, we specify when a sensor is read, which sensor reading is used for computing an actuator value, and when the actuator is set, without specifying a CPU or a priority for the computation.

In short, an *embedded software model* separates the platform-independent from the platform-dependent issues in embedded software development. This facilitates code generation by partitioning the task into two steps. In the first step, called program generation, a given mathematical model is transformed into an embedded software model (i.e., a high-level embedded program). This step is entirely independent of any execution platform; it specifies only the *reactivity* (i.e., real-time response) of the system relative to a physical environment. In the second step, called compilation, the software model is transformed into executable code for a target platform. This step must ensure the *schedulability* (i.e., real-time execution) of the system in a specific execution environment. While program generation provides a fully specified algorithmic solution to the control problem at hand, compilation is usually concerned also with nonreactive properties of the solution, such as resource utilization and fault tolerance.

The explicit use of a software model during code generation offers improved flexibility in the verification, optimization, integration, and reuse of embedded components. All of these activities are easier to carry out at the level of an embedded software model, rather than at the level of platform-specific code.

Separating Timing from Functionality

Giotto provides an embedded software model for control applications. Consequently, a Giotto-based systems development methodology can separate high-level control concerns, such as sampling rates and control laws, which depend on the given control problem, from low-level implementation concerns, such as output jitter and device drivers, which depend on the chosen platform. In addition, the Giotto methodology also separates timing concerns, such as sampling rates and output jitter, from functionality concerns, such as control laws and device drivers. A Giotto program supervises the interaction between software

Giotto is a high-level programming language for control applications.

processes and the physical world but does not itself transform data. All computation is encapsulated inside the supervised software processes, which can be written in any nonembedded programming language, such as C. We refer to a Giotto program as a *timing program* and to the supervised processes called by the Giotto program as *functionality programs*.

In this way, Giotto enforces a programming discipline, the strict separation of timing and functionality, that is often violated in real-time programs, especially when the programs are hand-optimized to meet timing constraints. This programming discipline offers additional benefits in the verification, optimization, integration, and reuse of embedded components. All of these activities are easier to carry out independently for timing and functionality programs. Similarly, the separation of timing and functionality further facilitates code generation: the timing (Giotto) program can be compiled into executable timing code, independent of the compilation of the functionality (e.g., C) programs into executable pieces of functionality code. After linking, the timing code supervises the execution of the functionality code.

It is important to note that this scheme offers great flexibility for the compiler. A Giotto program specifies only the reactivity of the functionality programs; i.e., when they are invoked and when their outputs are read, but not their scheduling. Hence, on the level of the Giotto software model, all functionality programs are atomic execution units, without priorities or internal synchronization points. In other words, the software processes supervised by a Giotto program are subroutines (functions) rather than coroutines (threads) [4]. This makes the Giotto software model transparent and particularly attractive for verification [5]. The Giotto compiler, on the other hand, is free to produce timing code that preempts functionality code and will generally do so for scheduling and optimization purposes. Thus, on the level of platform code, the pieces of

functionality code that are supervised by the timing code are indeed coroutines, not subroutines. This illustrates strikingly the main property of a good software model: the software model should emphasize transparency (simplicity), and thus improve reliability and enable reuse, whereas the compiler should emphasize performance.

The Giotto Tool Chain

Figure 1 shows the control systems development process without an embedded software model that separates timing from functionality; Figure 2 shows how the process changes if Giotto is used. The Giotto methodology partitions the code generation from control models into four independent phases. The four phases are supported by four different tools with the following outputs:

1) From a given Simulink model, the Simulink code generation facilities, e.g., MathWorks' Real-Time Workshop (RTW) Embedded Coder, generate C programs for the functional entities of the design, such as control law computations. Other C programs that implement functional units, such as device drivers, may be taken from libraries. The individual functionality programs are sequential and independent, and their execution needs to be supervised by a timing program.

2) From the Simulink model, the *S/G translator* generates a Giotto program, which specifies the timing behavior of a control system in a platform-independent way. The Giotto program supervises the execution of the functionality programs in response to the behavior of the physical environment.

3) From the functionality programs, the C compiler generates executable pieces of code for a chosen platform.

4) From the Giotto program, the *Giotto compiler* generates platform code that, after linking, supervises the execution of functionality code in a way that guarantees the real-time behavior specified by the Giotto program. To generate code with this property, the Giotto compiler needs to perform a schedulability analysis and reject a Giotto program if its timing constraints cannot be met on the target platform.

In this code-generation tool chain, there are two Giotto-specific tools, namely, the S/G translator and the Giotto compiler.

In addition, Giotto offers a simulation tool, the *S/G simulator*, which allows the simulation of Giotto models that are specified in Simulink. The benefits of using the S/G simulator rather than Simulink's own simulation facilities is that
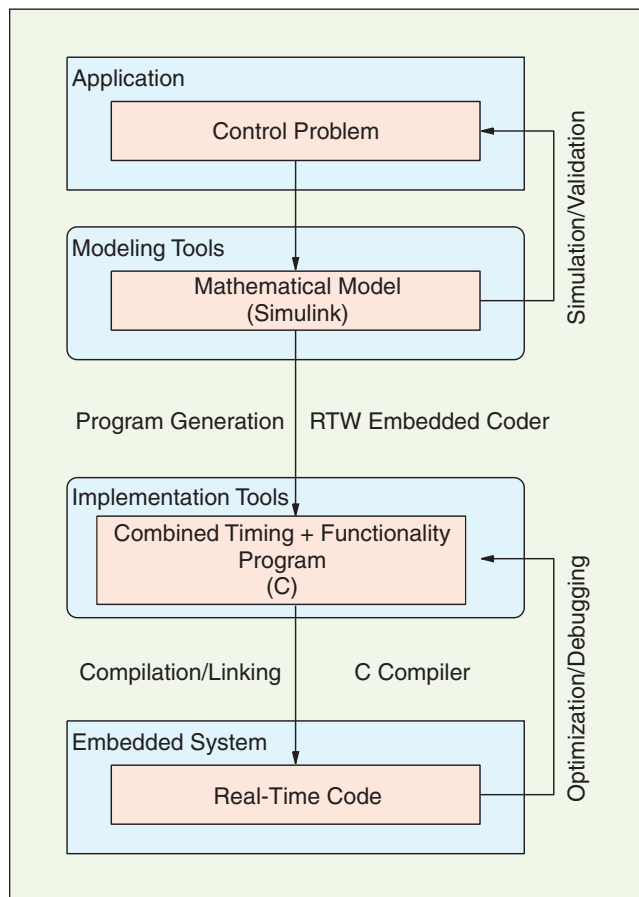

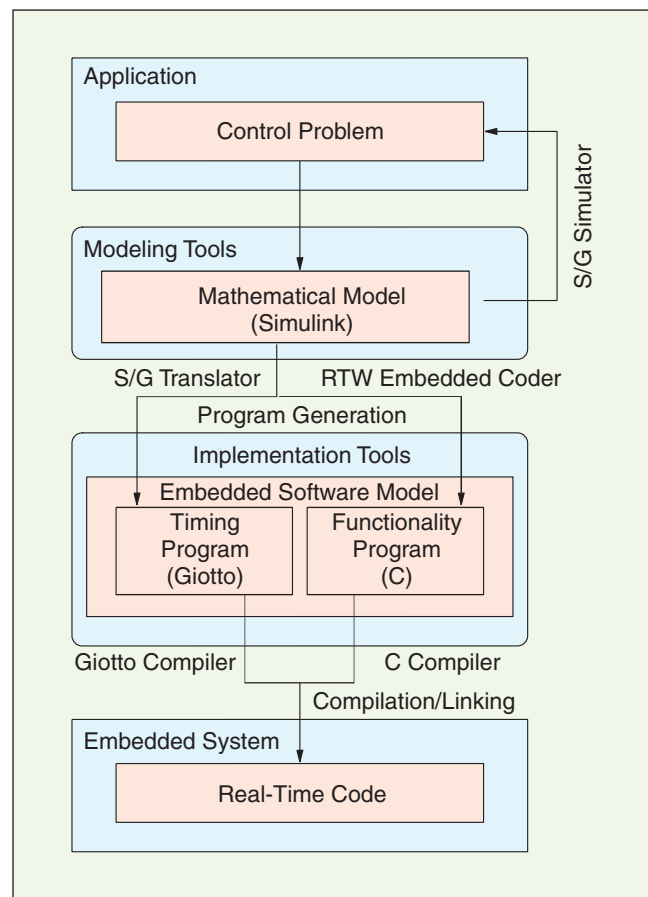**Figure 1.** *The traditional control systems development process.*

Figure 2. *The Giotto-based control systems development process.*

the Giotto tool chain guarantees that the real-time and functional behavior of the generated code matches exactly the result of S/G simulation. The S/G simulator supports the rapid prototyping and validation of Giotto control systems in Simulink, because the Giotto code-generation tools are fully compliant with the S/G simulation semantics.

A Giotto-Based Autonomous Helicopter System

We illustrate the Giotto methodology by reengineering an existing complex control system: the autopilot software of an autonomously flying model helicopter. We started from a system that already met the desired objectives: a fully working system with a well-modularized software architecture. This made it easy to isolate the functional components from the existing code. A similar reengineering approach has been used to assess the feasibility of other methodologies, such as MetaH, in the context of embedded missile control systems [6].

The Helicopter System

The original helicopter control system [3] was developed at ETH Zürich as part of an interdisciplinary project to build an autonomously flying model helicopter for research purposes. The control system hardware is a custom-crafted computer board with a single 200-MHz StrongARM processor and specialized I/O devices. All functional components are implemented in the programming language Oberon [7], [8] on top of the custom-designed real-time operating system HelyOS [9]. The control system (hardware, HelyOS, and autopilot software) is called OLGA, which stands for Oberon Language Goes Airborne. OLGA is now sold by weControl, a spin-off company of ETH Zürich, under the name of wePilot1000 [10].

The complete helicopter system consists of an aircraft (i.e., the model helicopter), the OLGA control system, and a ground system. Figure 3 shows a picture of the helicopter, and Figure 4 shows the system structure. The ground system (bottom of Figure 4) supports mission planning, flight command activation, and flight monitoring. Since the ground system is not relevant for the reimplementation of the OLGA autopilot software, it is not discussed here. All sensors and computational resources used for flight control and navigation are airborne (with the exception of a secondary GPS receiver used for the differential GPS). The sensors used on the helicopter are a GPS receiver, a compass, a revolution sensor, a laser altimeter, three accelerometers, three gyroscopes, and a temperature sensor. The actuators are six servos that control the main and tail rotor blades as well as the throttle of the two-stroke combustion engine. The OLGA control system generates pulse-width-modulated (PWM) servo commands. Takeoff and landing are under control of a human pilot. The transition from manual to autonomous flight is done when the helicopter is hovering. To allow a smooth transition, OLGA tracks the commands from the hu-

man pilot and, upon transition, uses them as the initial operating point of the controller. In case of an emergency, the human pilot is able to bypass OLGA at any time during flight.

The complexity of helicopter flight control results from the number of different sensors and actuators the control system has to handle concurrently, the difficulty of flying a helicopter, and the physical limitations of the control system (electrical consumption, limited computational resources, vibrations, jitter, etc.). Since the helicopter is a dangerous and expensive platform, 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 the OLGA control system, the controller runs at 40 Hz and data fusion (sensing) at 200 Hz. The worst-case execution time of the controller and data fusion implementation is 12 ms within a single control cycle of 25 ms, which results in a CPU utilization of more then 45%, leaving little room for other activities such as bookkeeping and monitoring.

Research efforts in unmanned aerial vehicles (UAVs), and more specifically in autonomous helicopters, have pro-

Figure 3. *The model helicopter and the OLGA control system (aluminum box).*
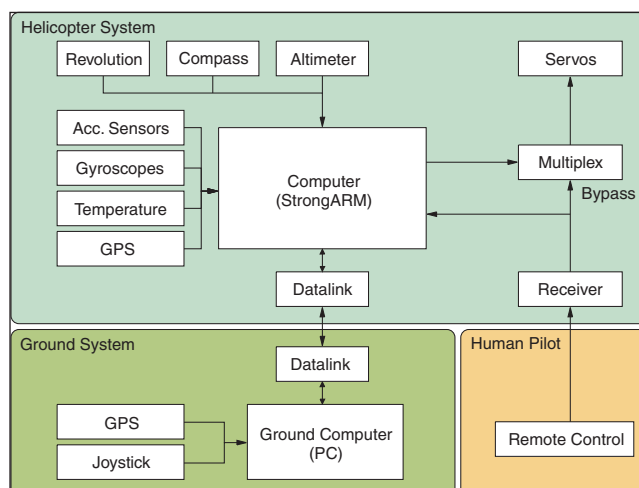

Figure 4. *The OLGA control system hardware structure.*

gressively shifted focus from a pure control problem to an interdisciplinary problem, where the software performance, reliability, and quality have become major critical factors. This is demonstrated by the academic helicopter projects [11]-[18]. In [19] an overview of other autonomous model helicopter projects is given.

The Structure of the Autopilot Software

The autopilot software has six different modes of operation (see Figure 5). In each mode, `Init`, `Idle`, `Motor`, `Take-Off`, `ControlOff`, and `ControlOn`, different tasks are active. In all modes except for the `ControlOn` mode, the human pilot controls the rotor blades, whereas the rotor speed is always controlled by OLGA. The first three modes are needed to handle the initialization procedure correctly. In particular, the `Motor` mode handles the 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 takeoff 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 rotor blades. 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 `ControlOff` and `ControlOn` modes.

In the `ControlOff` mode, the autopilot software reads the commands from the human pilot received via the wireless link and forwards them to the servos. The `ControlOff` mode consists of the 200-Hz task `ADFilter` and the 40-Hz task `NavPilot`. The `ADFilter` task decodes and preprocesses sensor values (data fusion). The `NavPilot` task keeps track of the position and velocity of the helicopter using the preprocessed data from the `ADFilter` task and forwards the pilot commands to the servos. The `ControlOff` mode switches to the `ControlOn` mode if the pilot pushes a button on the remote control. In addition to 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.

The Giotto Model of the Autopilot Software in Simulink

A Giotto model specifies the real-time interaction of a set of components with the physical world, as well as the real-time interaction between the components. All components of a Giotto model execute periodically. For this purpose, a Giotto model has a parameter called hyper-period, which is the least common multiple of all component periods. Figure 6 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. The controller block is a so-called *Giotto model block*, which exhibits a special semantics described below. 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. A Giotto task is a periodically executed function. The `ADFilter` block contains only standard discrete-time Simulink blocks that implement the decoding and preprocessing of sensor values. The functionality program associated with a Giotto task block might be obtained in a number of ways. In our case study, the functionality program is written in Oberon and taken from the legacy OLGA software. Alternatively, functionality programs that implement Giotto task blocks might be handwritten or automatically generated using Simulink's code-generation facilities.

The second block in the Giotto model is an example of a *Giotto case block*, which may contain multiple Giotto tasks. Like a task block, a Giotto case block is invoked periodically. Upon each invocation, the case block selects one of its tasks for execution. 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 produces the servo values based on a control law computation. Each Giotto case block has a frequency, which is given as an integer value relative to the hyper-period of the Giotto model. The case block of the example has the frequency 1; that is, it is invoked with a period of 25 ms. Both tasks in the case block inherit the frequency. Note that the `ADFilter` task block of the Giotto model is actually an abbreviation for a case block containing a single task. In
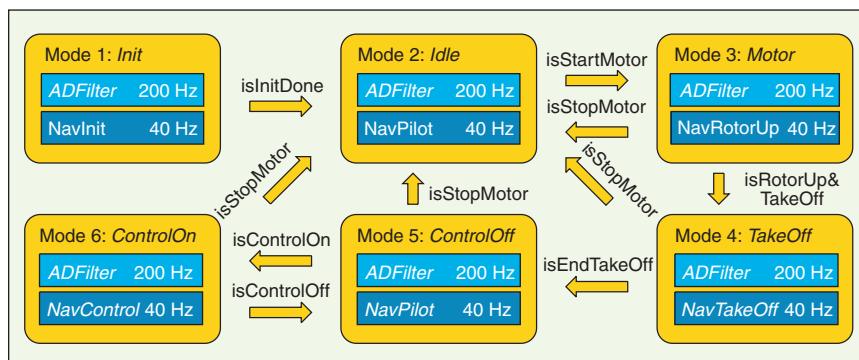

Figure 5. *The operating modes of the helicopter controller.*

fact, Giotto model blocks contain only case blocks. The virtual case block around the `ADFilter` task has the frequency 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 to select, based on its input values, at most one of the tasks for execution. If no task is chosen for execution, then all previous output values are held. The `isControlOff/On` block reads the pilot command to switch from manual to autonomous mode and back and accordingly 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. The block labeled `OutputPort` is necessary only for multiplexing the outputs of the two tasks into a single output. The tasks and the switch block in a case block may only read from the inputs of the case block but not from any task outputs within that block. To do this, it is necessary to establish a feedback link outside of the case block from an output to an input.

Generally, a Giotto model may consist of multiple case blocks that all run concurrently. More precisely, a set of tasks—at most one per case block—run concurrently, and each such combination of tasks is called a *Giotto mode*. For example, the `ADFilter` and `NavPilot` tasks implement the `ControlOff` mode of the helicopter system, whereas the `ADFilter` and `NavControl` tasks implement the `ControlOn` mode. Thus, any case switching during execution implements a Giotto mode switch.

Giotto Semantics

The key property of the Giotto semantics is the *fixed logical execution time* (FLET) assumption, which assumes that the execution times associated with all computation and communication activities are fixed and determined by the model, not the platform. In Giotto, the logical execution time of a task is always exactly the period of the task (i.e., the period of the surrounding case block), and the logical execution times of all other activities (switch blocks, data transfer across links, etc.) are always zero [2]. Note that the FLET assumption has all concurrent task executions within a Giotto mode run logically in parallel.

The logical execution time of a task is an abstract notion that is possibly very different from the actual, physical execution time of the task on a particular CPU, which may vary from task invocation to task invocation. The power of the FLET assumption stems from the fact that logical, not physical, execution times determine when sensors are read, when actuators are written, and when data travels across links. For example, the `ADFilter` task logically executes for 5 ms, which implies that 1) it reads its input at the beginning of its period, and 2) its output is not available to other tasks before 5 ms, even if the actual execution of the task on the CPU finishes earlier. Similarly, the case block with the two tasks `NavPilot` and `NavControl` logically executes for exactly 25 ms.

The FLET has two important consequences. First, sensors are read only at the beginning of a task's period and actuators are updated only at the end of a task's period. This minimizes jitter. In the example, the sensors are sampled by the `ADFilter` task at a frequency of 200 Hz and the servos are updated by the case block at a frequency of 40 Hz, precisely at the end of each hyper-period. Second, all intertask communication happens at period boundaries. This eliminates race conditions between tasks. In the example, the case block can only read the output from the `ADFilter` task invocation at the end of the previous hyper-period, but not from any `ADFilter` invocation during the current hyper-period. Consequently, although the `NavControl` task does not always use the latest available data from `ADFilter`, the data used is independent of varying execution times of `ADFilter` invocations and independent of the scheduling scheme that chooses between the `NavControl` and `ADFilter` tasks when both are ready to be executed on the same CPU. We have not encountered a "stale data" problem in the helicopter control system; on the contrary, this small penalty incurred by the use of Giotto is more than compensated by the improved predictability of the overall system. In particular, as a consequence of the FLET assumption, a Giotto model is *environment determined* [20]: for any given behavior of the physical world seen through the sensors, the model computes a unique trace; that is, predictable actuator values at predictable time instants. In other
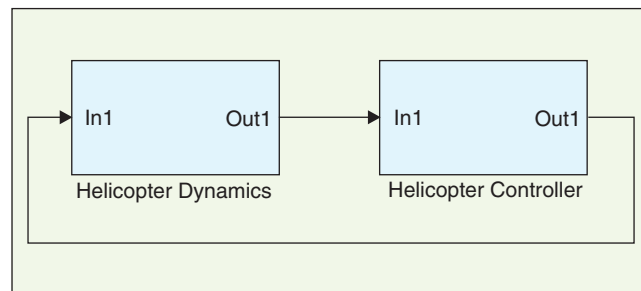

Figure 6. *The helicopter model in Simulink.*
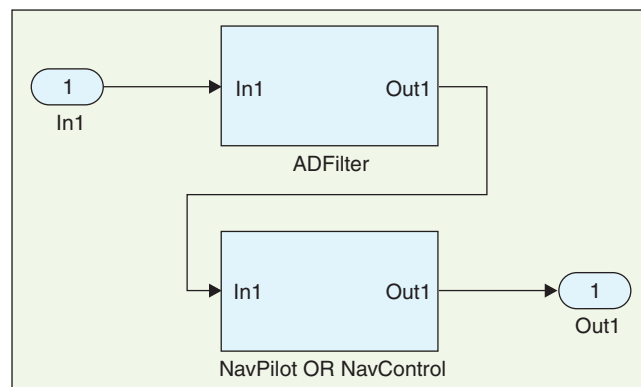

Figure 7. *The Giotto helicopter controller in Simulink.*

words, the only source of nondeterminism in a Giotto system is the physical environment. This makes the validation of the system considerably easier.

Simulating Giotto Models in Simulink

To simulate Giotto models in Simulink, we have developed the S/G simulator, which implements the Giotto semantics. In practice, the S/G simulator reads a Simulink specification of a Giotto model and transforms it into a standard discrete-time multirate Simulink model that makes the Giotto semantics (i.e., the FLET assumption) explicit in Simulink. Then the S/G simulator uses Simulink's simulation facilities to simulate the transformed model. We refer to simulation of the transformed model as *S/G simulation* of the original Giotto model.

Figure 9 shows the result of S/G simulating a stripped-down version of the helicopter controller for a duration of 140 ms with mode switches at the 50- and 100-ms time instants. The simulation illustrates only the timing of the intertask communication of the autopilot Giotto model on trivial data. For simplicity, we have replaced the implementation of the `ADFilter` task with a pulse generator that toggles the output of the task between 0 and 1. The simplified implementation of the `NavPilot` task reads that output and adds 1; the simplified `NavControl` task instead adds 3. All task outputs are initially 0. In this simplified example, where no sensor values are read, environment determinedness ensures that there is a unique (infinite) trace of the program (i.e., there are no race conditions between the two tasks).

Figure 9 shows the prefix of the trace up to 140 ms. At the 5-ms time instant, the first invocation of the `ADFilter` task is finished, and thus the output of the task changes from 0 to 1. At the 10-ms instant, the second invocation is finished and the output changes back to 0, and so on. Note that the pulse generator inside the `ADFilter` task implements a 0/1 toggle independent of any physical execution times: it is the task block frequency that makes it a 5-ms pulse generator. At the 25-ms instant, the first invocation of the `NavPilot` task is finished. This task was chosen by the surrounding case block to execute at the 0-ms instant. As it read its input also at the 0 ms instant, its input was the initial output value 0 from the `ADFilter` task. Thus, the output of the `NavPilot` task at the 25-ms instant is 0+1 (i.e., 1). The second invocation reads 1 from the `ADFilter` task and thus outputs 2 at the 50-ms instant. Now the case block chooses the `NavControl` task to execute, which reads 0 from the `ADFilter` task. The `NavControl` task outputs 0+3 (i.e., 3) at the 75-ms instant. For its second invocation, the task reads 1 from the `ADFilter` task and thus outputs 4 at the 100-ms instant. Now the case block again chooses the `NavPilot` task to execute, which reads 0 from the `ADFilter` task. Hence its output is 1, which is available at the 125 ms instant.

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 tasks that presently make up the controller. This, of course, assumes the provision of sufficient computational resources, which is checked by the Giotto compiler for a specified platform (see below).

From Giotto Models to Giotto Programs

The code generation from Giotto models proceeds in several steps (recall Figure 2). First, the S/G translator takes a Giotto model block specified in Simulink and generates the corresponding Giotto program in textual form, which is then processed by the Giotto compiler for schedulability analysis and to generate timing code. Here we discuss the Giotto program that results from S/G translating a given Giotto model in Simulink; the Giotto compiler will be discussed in the next section.

A Giotto program defines the exact timing and communication between Giotto tasks and between the program and the physical environment. All communication happens explicitly across links that connect the blocks in a Giotto model. The sources and destinations of these links are implemented as *Giotto ports*, which are locations in a global name space that carry values. The Giotto ports are partitioned into task input and task output ports, sensor ports, and actuator ports. In Figure 7, the `ADFilter` task block has an incoming link from sensor ports to the task input ports and an outgoing link from the task output ports to task in-
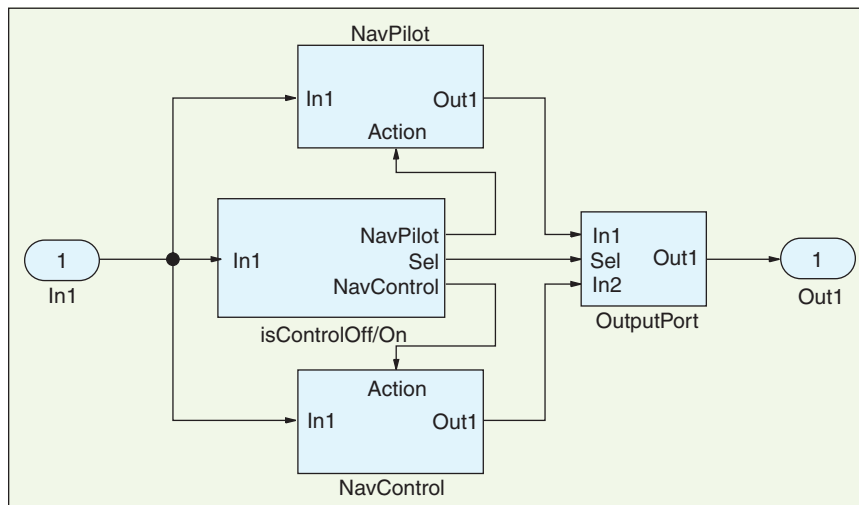

Figure 8. *The Giotto case block in Simulink.*

put ports within the case block; the case block has an outgoing link from task output ports to actuator ports. Sensor ports are written outside of the Giotto model and read inside; actuator ports are written inside of the Giotto model and read outside. In particular, from the perspective of the Giotto model, it makes no difference if a sensor port obtains its value from the physical environment or from software outside of the Giotto model.

A Giotto program needs to make explicit details that are left implicit or unspecified in the Simulink specification of a Giotto model, such as port declarations and declarations of functionality programs for reading and writing port values. To transport values between ports and to interface with the hardware, Giotto uses the concept of drivers. We distinguish between *Giotto link drivers* and *Giotto device drivers*. (There are also initialization and port drivers, which will be discussed later.) The Giotto link drivers implement the switch blocks and data links in the Simulink specification of a Giotto model; they can be further partitioned into task, actuator, and mode drivers. The purpose of task and actuator drivers is to transport values from sensor ports and task output ports to task input ports and actuator ports, respectively; a 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 Giotto port, or vice versa. For example, a device driver may read sensor values and write the results to a sensor port of the Giotto model; it may write values produced by an asynchronous (event-triggered) task to a sensor port of the Giotto model; or it may update an actuator setting using the value of an actuator port of the Giotto model. In addition to transporting data, drivers may perform some preprocessing of the data, such as type conversion. In the Simulink specification of a Giotto model, task and actuator drivers exist only implicitly as links, mode drivers correspond to switch blocks, and device drivers are absent entirely.

The Giotto Program for the Autopilot Software

From a Giotto model block in Simulink, the S/G translator generates 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 example shows the Giotto program `helicopter controller`, which specifies the `ControlOff` and `ControlOn` modes of the control system:
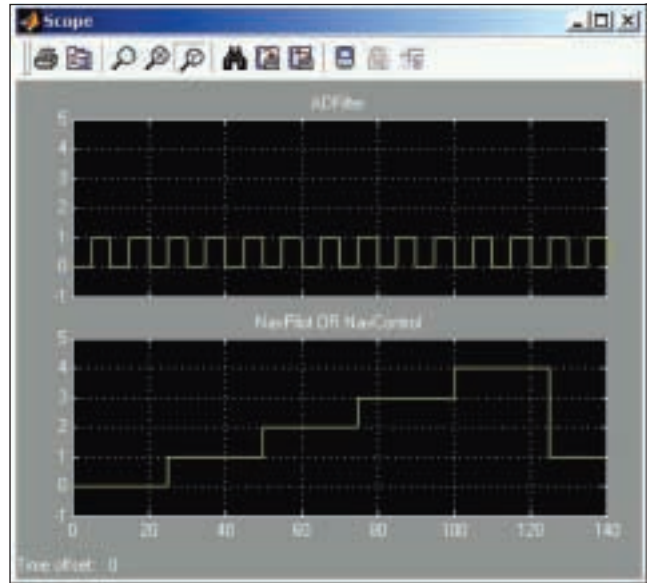
```
mode ControlOff() period 25 {
```


Figure 9. *A trace of the simplified Giotto helicopter controller with mode switching at 50 and 100 ms.*

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

The hyper-period of both modes is 25 ms. The frequencies of the task invocations, actuator updates, and mode switches are specified relative to this period using the keywords `taskfreq`, `actfreq`, and `exitfreq`, respectively. 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 invoking the actuator drivers `ServoUpdate` and `DataPoolUpdate`, respectively. In mode `ControlOff`, a switch to mode `ControlOn` is contemplated every 25 ms by executing the mode driver `SwitchOn`, which evaluates a mode-switch condition.

Figure 10 shows the logical execution of a single hyper-period of the `ControlOn` mode (a possible physical execution is shown in Figure 13 and will be discussed later). The logical execution satisfies the FLET assumption: the `ADFilter` task runs five times exactly for 5 ms, whereas the `NavControl` task runs once exactly for 25 ms. Both tasks run logically in parallel. All Giotto link and device drivers are executed in logical zero time. In practice, link and device drivers are bounded code that satisfies the synchrony as-

sumption [21] (as drivers cannot depend on each other, no issues of fixed-point semantics arise in Giotto).

Intertask communication, as well as communication with the environment of the Giotto program, 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 10, 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 de-

vices require immediate attention using an asynchronous (interrupt-driven) task. For example, the GPSGet device driver does not access the GPS device directly but reads a buffer into which the asynchronous task (interrupt handler) that is bound to the GPS device places the latest GPS readings. The asynchronous task 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 mode driver SwitchOff is called to determine whether or not to switch into the ControlOff mode. The mode driver is declared as follows:

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

The driver has a single input, stopswitch, that is a sensor port whose Giotto device driver StopSwitchGet has just been called. The device driver reads the value of a variable that represents the take-over button, whose value is transmitted asynchronously (i.e., externally to Giotto) via the wireless link from the remote control to the airborne control system. Based on the value of the stopswitch port, the Oberon implementation of the isControlOff predicate returns true or false, determining whether or not to switch to the ControlOff mode.

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
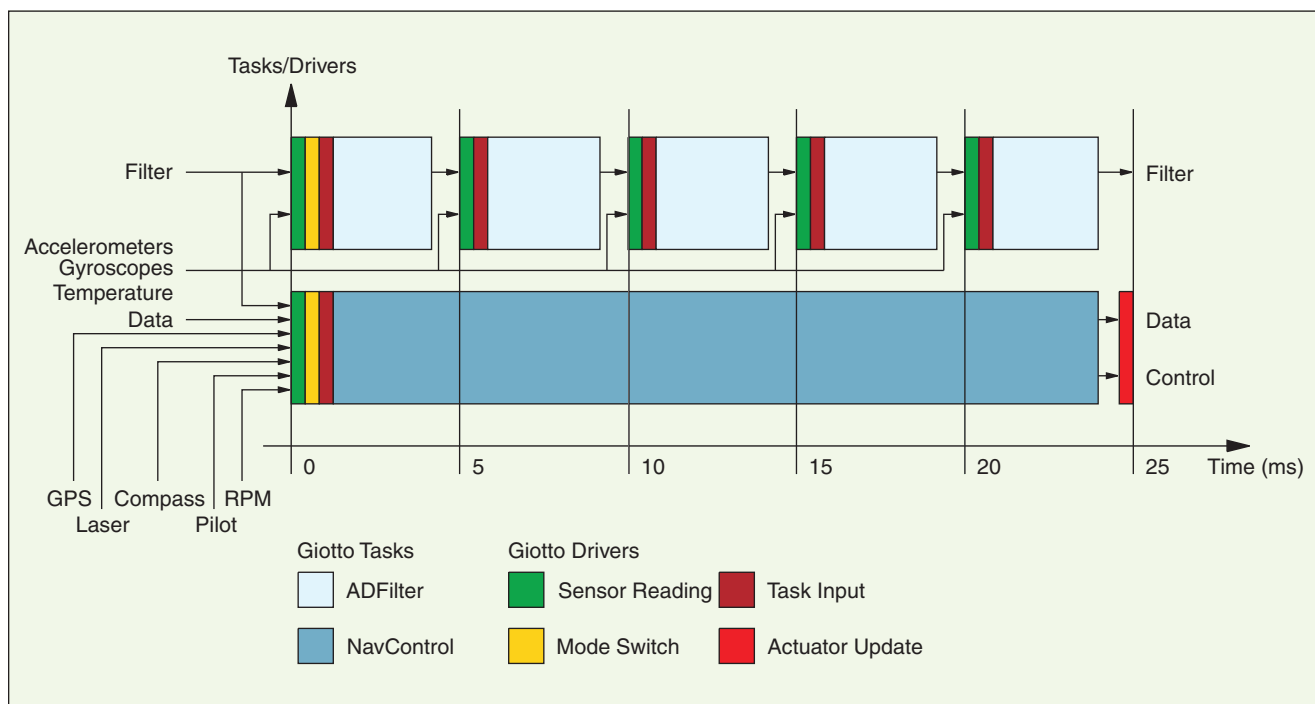
Figure 10. *The logical execution of the Giotto autopilot program in the* ControlOn *mode.*

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 will be the only task output port of the `ADFilter` task; the `control` and `data` ports will be used as task output ports by the `NavControl` task. For each task output port, in addition to a type name, an initialization driver is specified, which is invoked once at start-up time to initialize the port. For example, the task output port `filter` has the type `AnalogPort` and is initialized by the driver `FilterInit`. As usual, the initialization drivers are implemented externally to Giotto, here as Oberon procedures. Initial values for all task output ports sufficiently describe a unique start configuration of a Giotto program. The `ADFilter` and `NavControl` tasks are declared as follows:

```
task ADFilter(accelerometers, gyroscopes,
   temperature, filter)
   output (filter) {
   schedule ADFilterImplementation
      (accelerometers, gyroscopes,
       temperature, filter, 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 the task driver for `ADFilter` to some local memory, which is only accessible to the task itself. The task driver does not have to be declared explicitly in Giotto. The procedure `ADFilterImplementation`, which implements the functionality of the `ADFilter` task, is external to Giotto. In our case, it is an Oberon procedure that is taken directly from the original OLGA control software. 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; the `ADFilter` task runs logically concurrently and finishes after 5 ms. In practice, the output of `ADFilterImplementation` is kept in local memory when it becomes available and loaded into the task output port `filter` at the 5-ms instant. This is accomplished by an (undeclared) port driver for `filter`. Port drivers are used

to maintain the FLET assumption by making task outputs visible in task output ports only at the end of logical task execution, which may be later than the end of physical task execution. Then, still at 5 ms, 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, new values in the `control` and `data` output ports of the `NavControl` task are available. The new values are now transferred by the actuator drivers `ServoUpdate` and `DataPoolUpdate` to the `servos` and `datapool` actuator ports, respectively. Before declaring 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 10, at the 25-ms instant after the `NavControl` task finishes, 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` actuator ports, respectively. Then the `ServoPut` device driver is called, which takes the new value from the `servos` port and 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 a background (non-time-critical) task of the original OLGA control software. The background task is external to the Giotto program. The 25-ms hyper-period is now finished.

Giotto Compilation and Execution
Before we discuss the compilation and execution of the Giotto autopilot program on the OLGA hardware, we con-

sider the Giotto tool chain in slightly greater detail, as shown in Figure 11. From a given Simulink model that contains Giotto model blocks, the S/G translator generates a timing (Giotto) program for the Giotto model blocks, whereas we may use MathWorks' Real-Time Workshop Embedded Coder to generate the functionality (C) programs that implement the Giotto task and switch blocks inside the Giotto model blocks. Functionality (C) programs that implement the Giotto drivers need to be provided as well, typically from driver libraries. In the next step, the Giotto compiler generates timing code from the Giotto program and a C compiler generates functionality code from the C programs. The Giotto compiler targets a virtual machine called the *Embedded Machine* [20]. The compiler generates so-called *E code*, which is interpreted by the Embedded Machine in real time. There are E code instructions to call or schedule the functionality code and to invoke the Embedded Machine at specific time instants or occurrences of events. In the last step, before E code can be executed, it is linked through a common symbol table with the functionality code generated by the C compiler from the functionality programs. The execution environment for Giotto consists of an implementation of the Embedded Machine and a platform (i.e., operating system and hardware), which includes a scheduler for functionality code.

The Giotto Compiler

The main task of the Giotto compiler is to produce timing (E) code that is consistent with the FLET assumption of the Giotto program. To this end, the generated E code must be shown to be *time safe* [20] on the chosen platform, which intuitively means that all tasks meet the logical deadlines provided by the Giotto semantics. If the E code is time safe, then the execution behavior of the Giotto control system (i.e., timing and functionality code) is guaranteed to conform with the S/G simulation behavior of the original Giotto model. Because Giotto is environment determined, this conformance is precise in both timing and functionality: for any given sequence of sensor readings, the S/G simulation and the behavior of the generated code output the same sequence of actuator settings and update the actuator settings at the same points in time. In other words, a Giotto control system exhibits no internal race conditions, which makes its behavior predictable and verifiable.

The Giotto compiler checks time safety of the E code by performing a schedulability analysis on the Giotto program for given worst-case execution times of the functionality code. For single-CPU platforms, the schedulability analysis can be done in polynomial time by checking a utilization equation for each Giotto mode [22] (the analysis is exact under the reasonable assumption that all modes of the Giotto program can be reached during program execution; otherwise the analysis is conservative). The schedulability analysis requires worst-case execution time

assumptions for C code, which may be provided by non-Giotto-specific tools [23].

The code-generation process is perhaps best understood if we consider which parts need to be redone if the platform changes. To run a Giotto control system on a new platform, we must:

1) provide C programs that implement Giotto device drivers to interface the new hardware
2) recompile the functionality (C) programs for the new platform
3) have the Giotto compiler redo the time-safety check for the new worst-case execution times
4) implement the Embedded Machine on the new platform.

In particular, if the new platform offers sufficient performance, then the generated timing (E) code can also be run on the new platform. Indeed, once the Embedded Machine has been ported, the new execution environment can run any Giotto control system that passes a time-safety check.

The Giotto compiler has a great deal of freedom when generating E code. This is because a Giotto program does not specify where, how, and when tasks are scheduled. For example, the helicopter control program can be compiled on platforms with a single CPU (by time sharing between the data fusion and control tasks) or on platforms with two CPUs (by parallelism); it can be compiled on platforms with preemptive priority scheduling (such as most real-time operating
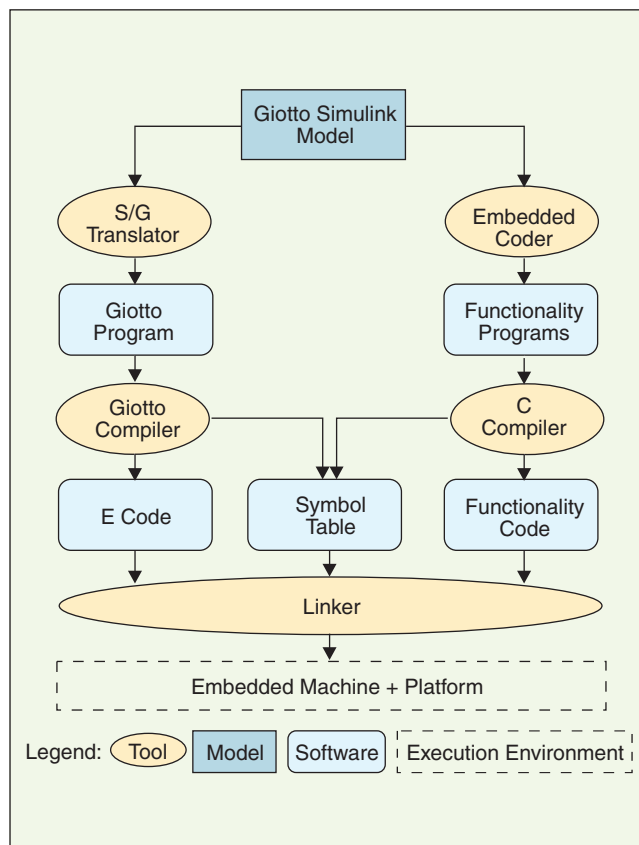

Figure 11. *The Giotto tool chain for code generation.*

systems) or on platforms with time-slice scheduling. Roughly, all the Giotto compiler needs to ensure is that 1) the sensors and actuators are read and written as close as possible to the times specified by the Giotto semantics, and 2) whenever in the Giotto semantics the logical completion of one task precedes the invocation of another task, then the same precedence is preserved during the actual execution [2]. The first requirement reduces I/O jitter; the freedom given by the second requirement can be used by the compiler to optimize performance. In particular, as we will see next, the helicopter system executes two logically concurrent, logically atomic (i.e., nonpreempted) tasks through time sharing and preemption on a single CPU.

As the Giotto compiler maintains compliance of the code with the logical semantics, Giotto systems can be composed. The existing I/O behavior of a Giotto control system does not change when new Giotto tasks are added, provided the compiler succeeds in showing that the resulting E code remains time safe despite the additional load.

The Giotto-Based Autopilot Control System

The system architecture of the Giotto-based helicopter control system is shown in Figure 12. The upper left portion shows the Giotto program and the corresponding functionality programs, namely, Oberon implementations of the Giotto device and link drivers and Giotto tasks. The non-Giotto tasks shown in the upper right portion of Figure 12 implement asynchronous (event-triggered) tasks or background (non-time-critical) tasks, which are interfaced to the Giotto system through sensor and actuator ports via Giotto device drivers. Asynchronous tasks must be taken into account by the schedulability analysis performed by the Giotto compiler; background tasks are executed only when the Giotto system is idle. In the middle of Figure 12, the original OLGA system software is extended by an implementation of the Embedded Machine in the kernel of the HelyOS real-time operating system. The underlying hardware is unchanged.

Figure 13 shows the actual, physical execution of the Giotto autopilot program in the ControlOn mode, which corresponds to the logical execution shown in Figure 10. Since the OLGA hardware has a single CPU to run the Giotto system as well as non-Giotto tasks, at any point during the actual execution, at most one of the Giotto or non-Giotto tasks can be running. This is in contrast to the logical execution,

where the two Giotto tasks ADFilter and NavControl run logically in parallel. To check the schedulability of the ControlOn mode, the Giotto compiler must verify that on the OLGA CPU, five times the worst-case execution time of ADFilter, plus the execution time of NavControl, plus worst-case assumptions for Giotto overhead (i.e., driver execution) and asynchronous non-Giotto tasks, are less than the hyper-period of the mode (25 ms).

The schedule of tasks during the actual execution is determined by the generated E code as well as the scheduler of the operating system (OS), which is part of the platform. The E code governs the timing behavior of the Giotto system: it triggers the immediate (synchronous) execution of the drivers and hands tasks that are ready to be scheduled to the OS. The HelyOS schedules Giotto tasks using a rate-monotonic scheduler and runs background tasks whenever the Giotto system is idle. There is a tradeoff between E code scheduling and scheduling by the OS. In the two extreme cases, the E code may give the OS maximal freedom by handing over tasks as soon as they are ready to be scheduled in conformance with the Giotto semantics, or it may make its own scheduling decisions and hand, at all times, only one of the ready tasks to the OS. Many intermediate solutions are also possible. E code scheduling is explicit, determined by the Giotto compiler; scheduling by the OS is implicit, depending on the scheduling strategy used by the OS. Note, however, that E code scheduling is not necessarily
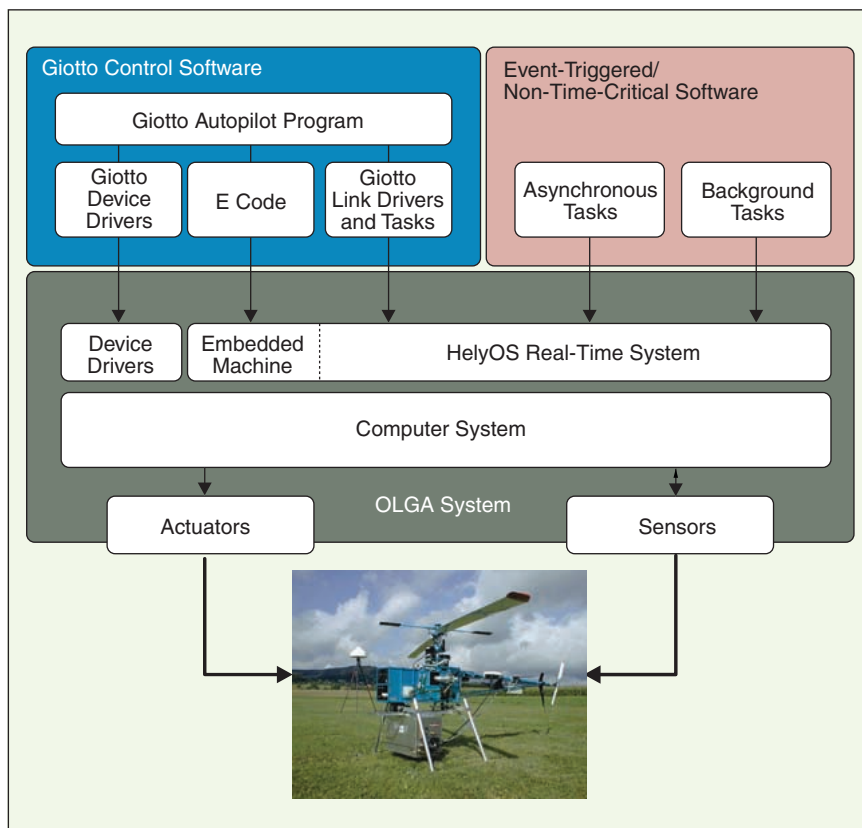

Figure 12. *The Giotto-based helicopter control system.*

static, because scheduling decisions may depend on the state of the program. E code is compatible with any scheduling strategy of the OS, but the scheduling strategy must be known when the Giotto compiler checks the generated E code for time safety.

The top row of Figure 13 shows the execution of the `ADFilter` task and the drivers from the top row of Figure 10. The middle row shows the execution of the `NavControl` task and the drivers from the bottom row of Figure 10. The bottom row shows the execution of background (non-time-critical) tasks. At 0 ms, the `ADFilter` task starts before the `NavControl` task, even though both are ready according to the Giotto semantics, because `ADFilter` runs at a higher rate and thus has a higher priority. For the same reason, `NavControl` is preempted at 5 ms by the second invocation of `ADFilter`. Note that to conform with the data flow specified by the Giotto semantics, the `NavControl` task must read the output of the last invocation of the `ADFilter` task from the previous control cycle, even though the first invocation of `ADFilter` in the current cycle is already completed when `NavControl` starts, and thus a fresher output of `ADFilter` would be available. The benefit of this strict adherence to the Giotto semantics is that the same output value, namely, the one defined by the FLET assumption, is used in all implementations, independent of the scheduling strategy and performance of the platform.

In the helicopter system, to implement the Giotto semantics, the input for the `NavControl` task is loaded at 0 ms and then buffered for 25 ms until the end of the task's period.

Similarly, task outputs are buffered and made available only at the end of a task period. Buffering task inputs and outputs is a sufficient but conservative technique for implementing the FLET assumption of the Giotto semantics. Other techniques such as scheduling with precedence constraints are possible [2].

Conclusions

Giotto is a domain-specific, high-level programming language: domain specific, as it addresses embedded control applications; high level, as it abstracts platform-dependent implementation details. Giotto increases the transparency of control software and automates scheduling and optimization through compilation.

The successful reengineering of the OLGA autopilot software using Giotto demonstrates the feasibility of the Giotto approach for high-performance control systems. The Giotto compiler automatically generates timing code for a control system with multiple modes of operation, multiple levels of task priorities, and time-triggered as well as event-triggered task activation. Giotto incurs an overhead through E code interpretation, predicate checks for mode switching, and the copying of ports. Measurements on the helicopter implementation have shown that this overhead amounts to less than 2% of a 25-ms period.

Giotto-based control systems benefit from the dual separation of concerns (reactivity versus scheduling; timing versus functionality) in many ways. First and foremost, the timing behavior of the control system is guaranteed to con-
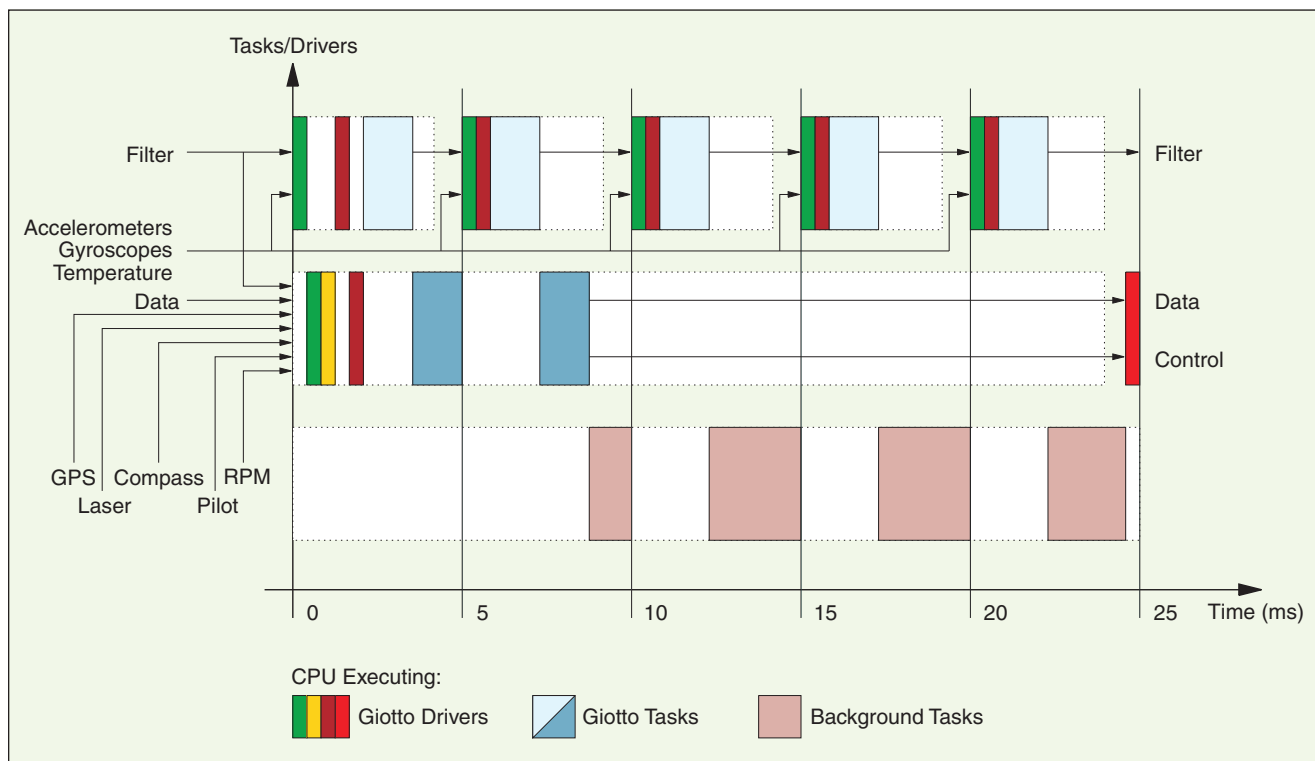
Figure 13. *The actual execution of the Giotto autopilot program in the* `ControlOn` *mode based on rate-monotonic scheduling.*

form to the simulation behavior of the corresponding Simulink model. Indeed, for any given behavior of the physical environment, the response of the Giotto system is unique and therefore predictable. Second, the development effort is significantly reduced, as the tedious programming of the timing code is handed over to the Giotto compiler. Also, the automation of the timing code generation eliminates a common source of errors. Third, the high degree of modularization and software transparency facilitates the easy exchange and addition of functionality, as well as the easy modification of timing requirements. Functionality programs can be packaged as software components and reused; timing (Giotto) programs can be composed without changing their individual behaviors. Fourth, the system can be ported to all platforms for which an implementation of the Embedded Machine is available. An implementation of the Embedded Machine on top of HelyOS was accomplished in one week, and its source code is only 6 KByte.

Current Giotto Implementations

There are several implementations of execution environments for Giotto besides the one used for the helicopter control system described in this article. The first implementation of Giotto was a simplified Giotto execution environment on a distributed platform of Lego Mindstorms [24] robots. The robots use infrared transceivers for communication. Then we implemented a full Giotto execution environment on a distributed platform of Intel x86 robots running the real-time operating system VxWorks [25]. The robots use wireless Ethernet for communication. We also wrote a Giotto program that runs on five robots, three Lego Mindstorms and two x86-based robots, to demonstrate the applicability of Giotto for heterogeneous platforms. For a discussion of this work, and embedded control systems development with Giotto in general, we refer to the earlier report [26].

We have implemented a Giotto-based automotive electronic throttle controller on a single Motorola MPC 555 processor running the real-time operating system OSEKWorks [27]. The Giotto program specifies three tasks: a 1- KHz controller task that implements various throttle controllers, a 33-Hz monitor task that monitors the system status, and a 66-Hz manager task that determines, based on the system status, which throttle controller is executed by the controller task.

The BEAR helicopter control system [16] is also being reimplemented in Giotto. In contrast to the OLGA helicopter control system, where the functionality programs have been reused from the original project, the second generation of the BEAR helicopter control system is a complete redesign of the software implementation [28].

Except for the one used in the OLGA helicopter control system (which is written in Oberon), all current versions of the Embedded Machine are written in C and are POSIX-compliant. Non-real-time implementations of the Embedded Machine are also available for Linux and Windows. The Giotto compiler is written in Java. All Giotto software tools are available for download at http://www.eecs.berkeley.edu/~fresco/giotto/.

Related Work

Giotto was originally inspired by the time-triggered architecture (TTA) [29], which first realized the time-triggered paradigm for meeting hard real-time constraints in safety-critical distributed settings. Whereas the TTA encompasses a hardware architecture and communication protocol, Giotto provides a platform-independent programmer's model for time-triggered applications.

The intertask communication semantics of Giotto is similar to the MetaH language [30], which is designed for real-time, distributed avionics applications. Giotto can be viewed as capturing a time-triggered fragment of MetaH in an abstract and formal way. In particular, unlike MetaH, Giotto specifies not only intertask communication but also mode switches in a time-triggered fashion, and it does not constrain the implementation to a particular scheduling scheme.

Many objectives of Giotto are shared by the synchronous reactive programming languages [21], including Esterel [31], Lustre [32], and Signal [33]. Giotto emphasizes the use of scheduled computation (i.e., the execution of tasks, which consume nonnegligible amounts of CPU time) at the expense of synchronous computation (i.e., the execution of drivers, which are constrained to be bounded, independent, noninteracting processes). Consequently, whereas the compilation of synchronous reactive languages focuses on fixed-point analysis, the compilation of Giotto focuses on schedulability analysis. A more detailed comparison is given in [34].

Acknowledgments

References

[1] C.M. Kirsch, M.A.A. Sanvido, T.A. Henzinger, and W. Pree, "A Giotto-based helicopter control system," in *Proc. 2nd Int. Workshop Embedded Software (EMSOFT)*, LNCS 2491, Springer Verlag, 2002, pp. 46-60.

[2] T.A. Henzinger, B. Horowitz, and C.M. Kirsch, "Giotto: A time-triggered language for embedded programming," in *Proc. 1st Int. Workshop Embedded Software (EMSOFT)*, LNCS 2211, Springer Verlag, 2001, pp. 166-184.

[3] J. Chapuis, C. Eck, M. Kottmann, M.A.A. Sanvido, and O. Tanner, "Control of helicopters," in *Control of Complex Systems*, K. Ågström, P. Albertos, M. Blanke, A. Isidori, W. Schaufelberger, and R. Sanz, Eds. New York: Springer Verlag, 1999, pp. 359-392.

[4] N. Wirth, "Tasks versus threads: An alternative multiprocessing paradigm," *Software-Concepts and Tools*, vol. 17, pp. 6-12, 1996.

[5] T.A. Henzinger, "Masaccio: A formal model for embedded components," in *Proc. 1st IFIP Int. Conf. Theoretical Computer Science*, LNCS 1872, Springer Verlag, 2000, pp. 549-563.

[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 Syst.*, vol. 14, pp. 7-20, 1998.

[7] N. Wirth, "A computer system for model helicopter flight control; Technical memo 6: The Oberon compiler for the StrongARM processor," Institute for Computer Systems, ETH Zürich, Tech. Rep. 314, 1999.

[8] N. Wirth and J. Gutknecht, *Project Oberon: The Design of an Operating System and Compiler*. New York: ACM Press, 1992.

[9] M.A.A. Sanvido, "A computer system for model helicopter flight control; Technical memo 3: The software core," Institute for Computer Systems, ETH Zürich, Tech. Rep. 317, 1999.

[10] weControl GmbH, *wePilot1000* [Online]. Available: http://www.wecontrol.ch

[11] The Robotics Institute, Carnegie Mellon University [Online]. Available: http://www.cs.cmu.edu/afs/cs/project/chopper/www

[12] The UAV Lab, Georgia Institute of Technology [Online]. Available: http://controls.ae.gatech.edu/labs/uavrf

[13] Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, "Aerial robotics" [Online]. Available: http://gewurtz.mit.edu/research/heli.htm

[14] Robotics Research Laboratory, University of Southern California, "Autonomous flying vehicles" [Online]. Available: http://www-robotics.usc.edu/~avatar

[15] Measurement and Control Laboratory, ETH Zürich, "Autonomous helicopter project" [Online]. Available: http://www.heli.ethz.ch

[16] Electronic Research Laboratory, University of California at Berkeley, "BEAR: Berkeley aerial robot" [Online]. Available: http://robotics.eecs.berkeley.edu/bear

[17] Aerospace Robotics Laboratory, Stanford University, "The Hummingbird helicopter" [Online]. Available: http://sun-valley.stanford.edu/~heli

[18] Institute for Technical Computer Science, Technische Universität Berlin, "Marvin" [Online]. Available: http://pdv.cs.tu-berlin.de/MARVIN

[19] C. Eck, "Navigation algorithms with applications to unmanned helicopters," Ph.D. dissertation, 14402, Computer Science, ETH Zürich, 2001.

[20] 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)*, ACM Press, 2002, pp. 315-326.

[21] N. Halbwachs, *Synchronous Programming of Reactive Systems*. Norwell, MA: Kluwer, 1993.

[22] T.A. Henzinger, C.M. Kirsch, R. Majumdar, and S. Matic, "Time-safety checking for embedded programs," in *Proc. 2nd Int. Workshop Embedded Software (EMSOFT)*, LNCS 2491, Springer Verlag, 2002, pp. 76-92.

[23] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm, "Reliable and precise WCET determination for a real-life processor," in *Proc.1st Int. Workshop Embedded Software (EMSOFT)*, LNCS 2211, Springer Verlag, 2001, pp. 469-485.

[24] Lego, "Mindstorms" [Online]. Available: http://mindstorms.lego.com

[25] Wind River Systems, "VxWorks operating system" [Online]. Available: http://www.windriver.com/products/html/vxwks5x.html

[26] T.A. Henzinger, B. Horowitz, and C.M. Kirsch, "Embedded control systems development with Giotto," in *Proc. ACM SIGPLAN Workshop Languages, Compilers, and Tools for Embedded Systems (LCTES)*, ACM Press, 2001, pp. 166-184.

[27] Wind River Systems, "OSEKWorks operating system" [Online]. Available: http://www.windriver.com/products/html/osekworks.html

[28] B. Horowitz, J. Liebman, C. Ma, J.T. Koo, T.A. Henzinger, A. Sangiovanni-Vincentelli, and S. Sastry, "Embedded-software design and system integration for rotorcraft UAV using platforms," in *Proc. 15th IFAC World Congress*, Elsevier, 2002 [CD-ROM].

[29] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Norwell, MA: Kluwer, 1997.

[30] S. Vestal, "MetaH support for real-time multi-processor avionics," in *Proc. 5th Int. Workshop Parallel and Distributed Real-Time Systems*, IEEE Computer Society Press, 1997, pp. 11-21.

[31] G. Berry, "The foundations of Esterel," in *Proof, Language and Interaction: Essays in Honour of Robin Milner*, G. Plotkin, C. Stirling, and M. Tofte, Eds. Cambridge, MA: MIT Press, 2000, pp. 425-454.

[32] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous dataflow programming language Lustre," *Proc. IEEE*, vol. 79, pp. 1305-1320, Sept. 1991.

[33] A. Benveniste, P. Le Guernic, and C. Jacquemot, "Synchronous programming with events and relations: The Signal language and its semantics," *Sci. Comput. Program.*, vol. 16, pp. 103-149, 1991.

[34] C.M. Kirsch, "Principles of real-time programming," in *Proc. 2nd Int. Workshop Embedded Software (EMSOFT)*, LNCS 2491, Springer Verlag, 2002, pp. 61-75.

Thomas A. Henzinger is a professor of electrical engineering and computer sciences at the University of California, Berkeley. He holds a Dipl.-Ing. degree in computer science from Kepler University, Linz, Austria; an M.S. degree in computer and information sciences from the University of Delaware; and a Ph.D. degree (1991) in computer science from Stanford University. He was an assistant professor of computer science at Cornell University (1992-1995) and a director of the Max-Planck Institute for Computer Science in Saarbrücken, Germany (1999). His research focuses on formalisms and tools for the design, implementation, and verification of reactive, real-time, and hybrid systems.

Christoph M. Kirsch is a postdoctoral researcher at the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley. He holds a Dipl.-Inform. degree (1996) and a Ph.D. degree (1999) in computer science from the University of Saarbrücken, Germany. He received both degrees while at the Max-Planck Institute for Computer Science in Saarbrücken, Germany. His research focuses on formalisms and tools for the design and implementation of real-time and embedded systems.

Marco A.A. Sanvido is a postdoctoral researcher at the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley. He holds a Dipl.-Ing. degree (1996) and a Dr.techn. degree (2002) in computer science from the Swiss Federal Institute of Technology in Zürich, Switzerland. His research focuses on tools for the design and implementation of real-time embedded systems.

Wolfgang Pree is a professor of computer science at the University of Salzburg, Austria. He holds a Dipl.-Ing. degree (1987) and a Dr.techn. degree (1992) in computer science from Kepler University in Linz, Austria. He was a visiting assistant professor at Washington University in St. Louis (1992-1993); a guest scientist at Siemens AG Munich (1994-1995); a professor of computer science at the University of Constance, Germany (1996-2001); and recently spent a sabbatical at the University of California, Berkeley. His research focuses on software construction, in particular, methods and tools for automating the development of real-time embedded software and for improving the reusability through generic software.