Giotto: A Time-triggered Language for Embedded Programming[⋆]

Thomas A. Henzinger Benjamin Horowitz Christoph Meyer Kirsch

University of California, Berkeley
{tah,bhorowit,cm}@eecs.berkeley.edu

Abstract. Giotto provides an abstract programmer's model for the implementation of embedded control systems with hard real-time constraints. A typical control application consists of periodic software tasks together with a mode switching logic for enabling and disabling tasks. Giotto specifies time-triggered sensor readings, task invocations, and mode switches independent of any implementation platform. Giotto can be annotated with platform constraints such as task-to-host mappings, and task and communication schedules. The annotations are directives for the Giotto compiler, but they do not alter the functionality and timing of a Giotto program. By separating the platform-independent from the platform-dependent concerns, Giotto enables a great deal of flexibility in choosing control platforms as well as a great deal of automation in the validation and synthesis of control software. The time-triggered nature of Giotto achieves timing predictability, which makes Giotto particularly suitable for safety-critical applications.

1 Introduction

Giotto provides a programming abstraction for hard real-time applications which exhibit time-periodic and multi-modal behavior, as in automotive, aerospace, and manufacturing control. Traditional control design happens at a mathematical level of abstraction, with the control engineer manipulating differential equations and mode switching logic using tools such as Matlab or MatrixX. Typical activities of the control engineer include modeling of the plant behavior and disturbances, deriving and optimizing control laws, and validating functionality and performance of the model through analysis and simulation. If the validated design is to be implemented in software, it is then handed off to a software engineer who writes code for a particular platform (we use the word "platform" to stand for a hardware configuration together with a real-time operating system). Typical activities of the software engineer include decomposing the necessary computational activities into periodic tasks, assigning tasks to CPUs and setting task priorities to meet the desired hard real-time constraints under the given scheduling mechanism and hardware performance, and achieving a degree of fault tolerance through replication and error correction.

[⋆] This research was supported in part by the DARPA SEC grant F33615-C-98-3614 and by the MARCO GSRC grant 98-DT-660.

Control design	– plant modeling – control law derivation
⇓	
Giotto program	– functionality and timing – periodic software tasks and mode switches
⇓	
Code for real-time platform	– hardware mapping – computation and communication scheduling

Fig. 1. Real-time control system design with Giotto

Giotto provides an intermediate level of abstraction, which permits the software engineer to communicate more effectively with the control engineer. Specifically, Giotto defines a software architecture of the implementation which specifies its functionality and timing. Functionality and timing are sufficient and necessary for ensuring that the implementation is consistent with the mathematical model of the design. On the other hand, Giotto abstracts away from the realization of the software architecture on a specific platform, and frees the software engineer from worrying about issues such as hardware performance and scheduling mechanism while communicating with the control engineer. After writing a Giotto program, the second task of the software engineer remains of course to implement the program on the given platform. However, in Giotto, this second task, which requires no interaction with the control engineer, is effectively decoupled from the first, and can in large parts be automated by increasingly powerful compilers. The Giotto design flow is shown in Figure 1. The separation of logical correctness concerns (functionality and timing) from physical realization concerns (mapping and scheduling) has the added benefit that a Giotto program is entirely platform independent and can be compiled on different, even heterogeneous, platforms.

Motivating example. Giotto is designed specifically for embedded control applications. Consider a typical fly-by-wire flight control system [LRR92,Col99], which consists of three types of interconnected components (see Figure 2): sensors, CPUs for computing control laws, and actuators. The sensors include an inertial navigation unit (INU), for measuring linear and angular acceleration; a global positioning system (GPS), for measuring position; an air data measurement system, for measuring such quantities as air pressure; and the pilot's controls, such as the pilot's stick. Each sensor has its own timing properties: the INU, for example, outputs its measurement 1,000 times per second, whereas the pilot's stick outputs its measurement only 500 times per second. Three separate control laws —for pitch, lateral, and throttle control— need to be computed.

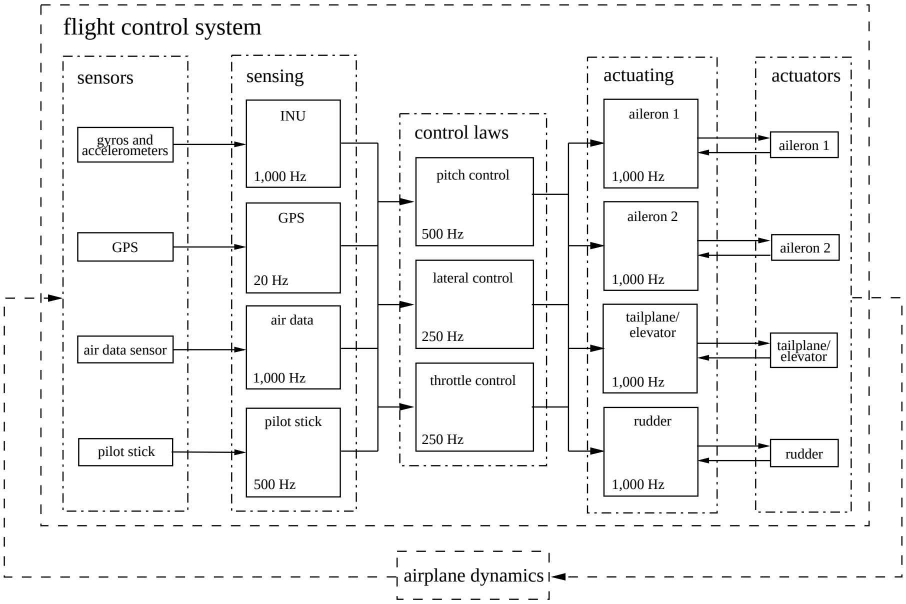

Fig. 2. A fly-by-wire flight control system

The system has four actuators: two for the ailerons, one for the tailplane, and one for the rudder. The timing requirements on the control laws and actuator tasks are also shown in Figure 2. The reader may wonder why the actuator tasks need to run more frequently than the control laws. The reason is that the actuator tasks are responsible for the stabilization of quickly moving mechanical hardware, and thus need to be an order of magnitude more responsive than the control laws.

We have just described one operational mode of the fly-by-wire flight control system, namely the cruise mode. There are four additional modes: the takeoff, landing, autopilot, and degraded modes. In each of these modes, additional sensing tasks, control laws, and actuating tasks need to be executed, as well as some of the cruise tasks removed. For example, in the takeoff mode, the landing gear must be retracted. In the autopilot mode, the control system takes inputs from a supervisory flight planner, instead of from the pilot's stick. In the degraded mode, some of the sensors or actuators have suffered damage; the control system compensates by not allowing maneuvers which are as aggressive as those permitted in the cruise mode.

The Giotto abstraction. Giotto provides a programmer's abstraction for specifying control systems that are structured like the previous fly-by-wire example. The basic functional unit in Giotto is the *task*, which is a periodically executed piece of, say, C code. Several concurrent tasks make up a *mode*. Tasks can be added or removed by switching from one mode to another. Tasks communicate with each other, as well as with sensors and actuators, by so-called *drivers*, which is code that transports and converts values between *ports*. While a task represents scheduled computation on the application level and consumes logical time, a 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). The periodic invocation of tasks, the reading of sensor values, the writing of actuator values, and the mode switching are all triggered by real time. For example, one task t_1 may be invoked every 2 ms and read a sensor value upon each invocation, another task t_2 may be invoked every 3 ms and write an actuator value upon each completion, and a mode switch may be contemplated every 6 ms. This time-triggered semantics enables efficient reasoning about the timing behavior of a Giotto program, in particular, whether it conforms to the timing requirements of the mathematical (e.g., Matlab) model of the control design.

A Giotto program does not specify where, how, and when tasks are scheduled. The Giotto program with tasks t_1 and t_2 can be compiled on platforms that have a single CPU (by time sharing the two tasks) as well as on platforms with two CPUs (by parallelism); it can be compiled on platforms with preemptive priority scheduling (such as most RTOSs) as well as on truly time-triggered platforms (such as TTA [Kop97]). All the Giotto compiler needs to ensure is that the logical semantics of Giotto —functionality and timing— is preserved. A Giotto program can be annotated with *platform constraints*, which may be understood as directives to the compiler in order to make its job easier. A constraint may map a particular task to a particular CPU, it may schedule a particular task in a particular time interval, or it may schedule a particular communication event between tasks in a particular time slot. These annotations, however, in no way modify the functionality and timing of a Giotto program; they simply aid the compiler in realizing the logical semantics of the program.

Outline of the paper. We first give an informal introduction to Giotto in Section 2, followed by a formal definition of the language in Section 3. In Section 4, we briefly describe annotated Giotto, a refinement of Giotto for guiding distributed code generation. In Section 5, we relate Giotto to the literature and mention ongoing application work.

2 Informal Description of Giotto

Ports. In Giotto all data is communicated through ports. A port represents a typed variable with a unique location in a globally shared name space. We use the global name space for ports as a virtual concept to simplify the definition of Giotto. An implementation of Giotto is not required to be a shared memory system. Every port is persistent in the sense that the port keeps its value over time, until it is updated. There are mutually disjoint sets of sensor ports, actuator ports, and task ports in a Giotto program. The sensor ports are updated by the environment; all other ports are updated by the Giotto program. The task ports are used to communicate data between concurrent tasks and from one mode to

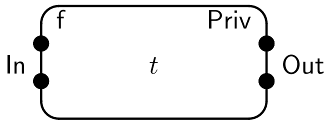

Fig. 3. A task t

the next. In any given mode, a task port may or may not be used; the used ports are called mode ports. Every mode port is explicitly assigned a value every time the mode is entered.

Tasks. A typical Giotto task t is shown in Figure 3. The task t has a set In of two input ports and a set Out of two output ports, all of which are depicted by bullets. The input ports of t are distinct from all other ports in the Giotto program. The output ports of t may be shared with other tasks as long as they are not invoked in the same mode. In general, a task may have an arbitrary number of input and output ports. A task may also maintain a state, which can be viewed as a set of private ports whose values are inaccessible outside the task. The state of t is denoted by Priv. Finally, the task has a function f from its input ports and its current state to its output ports and its next state. The task function f is implemented by a sequential program, and can be written in an arbitrary programming language. It is important to note that the execution of f has no internal synchronization points and cannot be terminated prematurely; in Giotto all synchronization is specified explicitly outside of tasks. For a given platform, the Giotto compiler will need to know the worst-case execution time of f on each CPU.

Tasks invocations. Giotto tasks are periodic tasks: they are invoked at regularly spaced points in time. An invocation of a task t is shown in Figure 4. If the task t is invoked in the mode m, then the output ports of t are included in the mode ports of m, along with the output ports of some other tasks. The task invocation has a frequency ω_{task} given by a non-zero natural number; the real-time frequency will be determined later by dividing the real-time period of the current mode by ω_{task}. The task invocation specifies a driver d which provides values for the input ports In. The first input port is loaded with the value of some other port p, and the second input port is loaded with the constant value κ. In general, a driver is a function that converts the values of sensor ports and mode ports of the current mode to values for the input ports, or loads the input ports with constants. Drivers can be guarded: the guard of a driver is a predicate on sensor and mode ports. The invoked task is executed only if the driver guard evaluates to *true*; otherwise, the task execution is skipped.

The time line for an invocation of the task t is shown in Figure 5. The invocation starts at some time τ_{start} with a communication phase in which the driver guard is evaluated and the input port values are loaded. The Giotto semantics prescribes that the communication phase —i.e., the execution of the

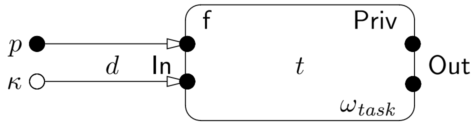

Fig. 4. An invocation of task t

driver d— takes zero time. The synchronous communication phase is followed by a scheduled computation phase. The Giotto semantics prescribes that at time τ_{stop} the state and output ports of t are updated to the (deterministic) result of f applied to the state and input ports of t at time τ_{start}. The length of the interval between τ_{start} and τ_{stop} is determined by the frequency ω_{task}. The Giotto logical abstraction does not specify when, where, and how the computation of f is physically performed between τ_{start} and τ_{stop}. However, the time at which the task output ports are updated is determined, and therefore, for any given real-time trace of sensor values, all values that are communicated between tasks are determined. Instantaneous communication and time-deterministic as well as value-deterministic computation are the three essential ingredients of the Giotto logical abstraction. A compiler must be faithful to this abstraction; for example, task inputs may be loaded after time τ_{start}, and the execution of f may be preempted by other tasks, as long as at time τ_{stop} the values of the task output ports are those specified by the Giotto semantics.

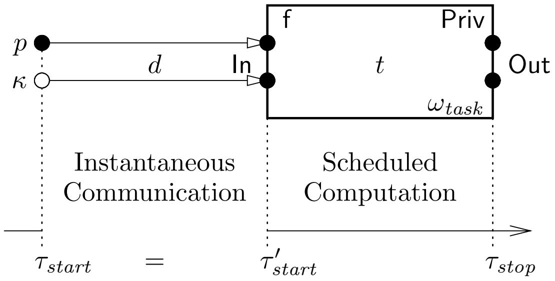

Fig. 5. The time line for an invocation of task t

Modes. A Giotto program consists of a set of modes, each of which repeats the invocation of a fixed set of tasks. The Giotto program is in one mode at a time. A mode may contain mode switches, which specify transitions from the mode to other modes. A mode switch can remove some tasks, and add others. Formally, a mode consists of a period, a set of mode ports, a set of task invocations, a set of actuator updates, and a set of mode switches. Figure 6 shows a mode m that contains invocations of two tasks, t_1 and t_2. The period π of m is 10 ms; that

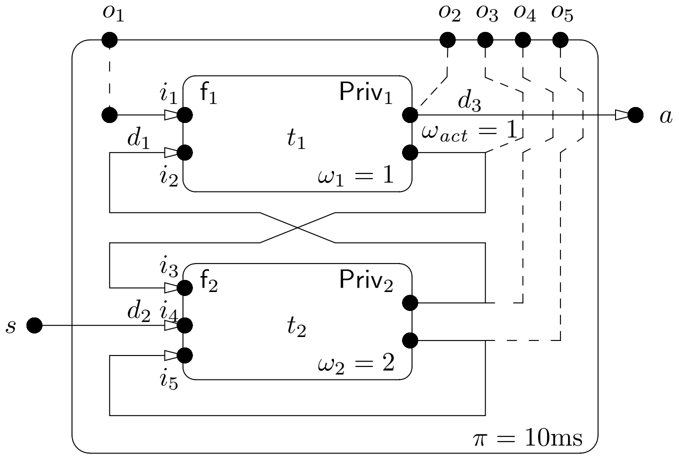

Fig. 6. A mode m

is, while the program is in mode m, its execution repeats the same pattern of task invocations every 10 ms. The task t_1 has two input ports, i_1 and i_2, two output ports, o_2 and o_3, a state Priv_1, and a function f_1. The task t_2 is defined in a similar way. Moreover, there is one sensor port, s, one actuator port, a, and a mode port, o_1, which is not updated by any task in mode m. The value of o_1 stays constant while the program is in mode m; it can be used to transfer a value from a previous mode to mode m. The invocation of t_1 in mode m has the frequency $\omega_1 = 1$, which means that t_1 is invoked once every 10 ms while the program is in mode m. The invocation of t_1 in mode m has the driver d_1, which copies the value of the mode port o_1 into i_1 and the value of the output port o_4 of t_2 into i_2. The invocation of t_2 has the frequency $\omega_2 = 2$, which means that t_2 is invoked once every 5 ms, as long as the program is in mode m. The invocation of t_2 has the driver d_2, which connects the output port o_3 of t_1 to i_3, the sensor port s to i_4, and o_5 to i_5. Note that the mode ports of m, which include all task output ports used in m, are visible outside the scope of m as indicated by the dashed lines. A mode switch may copy the values at these ports to mode ports of a successor mode. The mode m has one actuator update, which is a driver d_3 that copies the value of the task output port o_2 to the actuator port a with the actuator frequency $\omega_{act} = 1$; that is, once every 10 ms.

Figure 7 shows the exact timing of a single round of mode m, which takes 10 ms. As long as the program is in mode m, one such round follows another. The round begins at the time instant τ_0 with an instantaneous communication phase for the invocations of tasks t_1 and t_2, during which the two drivers d_1 and d_2 are executed. The Giotto semantics does not specify how the computations of the task functions f_1 and f_2 are physically scheduled; they could be scheduled in any order on a single CPU, or in parallel on two CPUs. Logically,

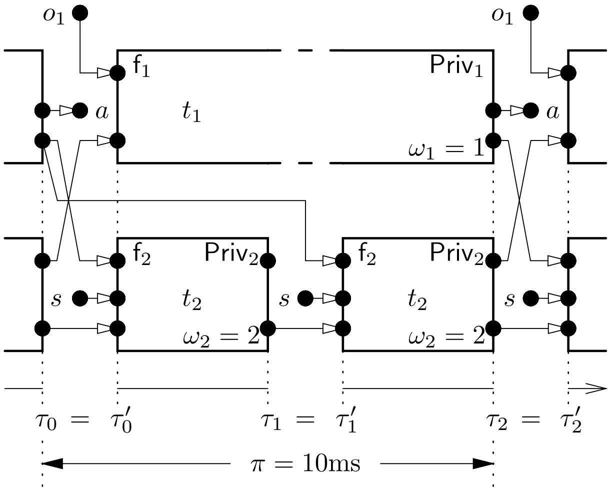

Fig. 7. The time line for a round of mode m

after 5 ms, at time instant τ_1, the results of the scheduled computation of f_2 are written to the output ports of t_2. The second invocation of t_2 begins with another execution of driver d_2, still at time τ_1, which samples the most recent value from the sensor port s. However, the two invocations of t_2 start with the same value at input port i_3, because the value stored in o_3 is not updated until time instant $\tau_2 = 10$ ms, no matter if physically f_1 finishes its computation before τ_1 or not. Logically, the output values of the invocation of t_1 must not be available before τ_2. Any physical realization that schedules the invocation of t_1 before the first invocation of t_2 must therefore keep available two sets of values for the output ports of t_1. The round is finished after writing the output values of the invocation of t_1 and of the second invocation of t_2 to their output ports at time τ_2, and after updating the actuator port a at the same time. The beginning of the next round shows that the input port i_3 is loaded with the new value produced by t_1.

Mode switches. In order to give an example of mode switching we introduce a second mode m', shown in Figure 8. The main difference between m and m' is that m' replaces the task t_2 by a new task t_3, which has a frequency ω_3 of 4 in m'. Note that t_3 has a new output port, o_6, but also uses the same output port o_4 as t_2. Moreover, t_3 has a new driver d_4, which connects the output port o_3 of t_1 to the input port i_6, the sensor port s to i_7, and o_6 to i_8. The task t_1 in mode m' has the same frequency and uses the same driver as in mode m. The period of m', which determines the length of each round, is again 10 ms. This means that in mode m', the task t_1 is invoked once per round, every 10 ms; the task t_3 is invoked 4 times per round, every 2.5 ms; and the actuator a is updated once per round, every 10 ms.

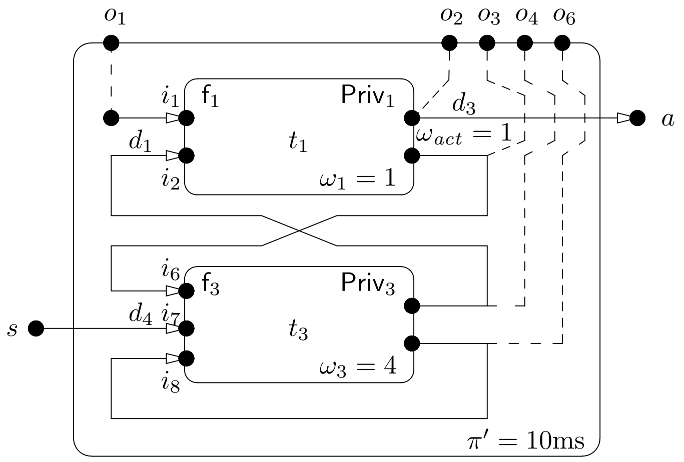

Fig. 8. A mode m'

A mode switch describes the transition from one mode to another mode. For this purpose, a mode switch specifies a switch frequency, a target mode, and a driver. Figure 9 shows a mode switch η from mode m to target mode m' with the switch frequency $\omega_{switch} = 2$ and the driver d_5. The guard of the driver is called *exit condition*, as it determines whether or not the switch occurs. The exit condition is evaluated periodically, as specified by the switch frequency. As usual, the switch frequency of 2 means that the exit condition of d_5 is evaluated every 5 ms, in the middle and at the end of each round of mode m. The exit condition is a boolean-valued condition on sensor ports and the mode ports of m. If the exit condition evaluates to true, then a switch to the target mode m' is performed. The mode switch happens by executing the driver d_5, which provides values for all mode ports of m'; specifically, d_5 loads the constant κ into o_1, the value of o_5 into o_6, and ensures that o_2, o_3, and o_4 keep their values (this is omitted from Figure 9 to avoid clutter). The explicit mention of the persistence of o_2, o_3, and o_4 is helpful, because like tasks, with a mode switch these ports may physically migrate from one CPU to another CPU, and thus may need to be copied. Like all drivers, mode switches are logically performed in zero time.

Figure 10 shows the time line for the mode switch η performed at time τ_1. The program is in mode m until τ_1 and then enters mode m'. Note that until time τ_1 the time line corresponds to the time line shown in Figure 7. At time τ_1, first the invocation of task t_2 is completed, then the mode driver d_5 is executed. This finishes the mode switch. All subsequent actions follow the semantics of the target mode m' independently of whether the program entered m' just now through a mode switch, at 5 ms into a round, or whether it started the current round already in mode m'. Specifically, the driver for the invocation of task t_3 is executed, still at time τ_1. Note that the output port o_6 of t_3 has just received the

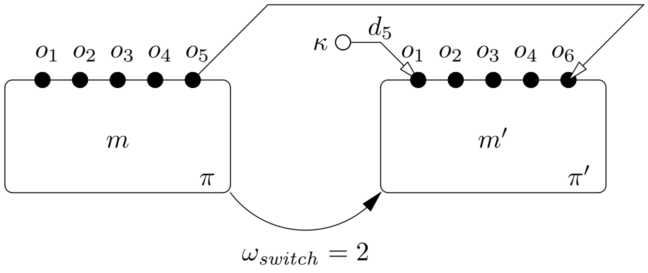

Fig. 9. A mode switch η from mode m to mode m'

value of the output port o_5 from task t_2 by the mode driver d_5. At time τ_2, task t_3 is invoked a second time, and at time τ_3, the round is finished, because this is the earliest time after the mode switch at which a complete new round of mode m' can begin. Now the input port i_1 of task t_1 is loaded with the constant κ from the mode port o_1. In this way, task t_1 can detect that a mode switch occurred.

For a mode switch to be legal, the target mode is constrained so that all task invocations that may be logically interrupted by a mode switch can be logically continued in the target mode. In our example, the mode switch η can occur at 5 ms into a round of mode m, while the task t_1 is logically running. Hence the target mode m' must also invoke t_1. Moreover, since the period of m' is 10 ms, as for mode m, the frequency of t_1 in m' must be identical to the frequency of t_1 in m, namely, 1. If, alternatively, the period of m' were 20 ms, then the frequency of t_1 in m' would have to be 2.

3 Formal Definition of Giotto

3.1 Syntax

Rather than specifying a concrete syntax for Giotto, we formally define the components of a Giotto program in a more abstract way. However, Giotto programs can also be written in a C like concrete syntax [HHK01]. A *Giotto program* consists of the following components:

1. A set of *port declarations*. A port declaration $(p, \mathsf{Type}, \mathsf{init})$ consists of a port name p, a type Type, and an initial value $\mathsf{init} \in \mathsf{Type}$. We require that all port names are uniquely declared; that is, if $(p, \cdot, \cdot)$ and $(p', \cdot, \cdot)$ are distinct port declarations, then $p \neq p'$. The set Ports of declared port names is partitioned into a set $\mathsf{SensePorts}$ of *sensor ports*, a set $\mathsf{ActPorts}$ of *actuator ports*, a set $\mathsf{InPorts}$ of *task input ports*, a set $\mathsf{OutPorts}$ of *task output ports*, and a set $\mathsf{PrivPorts}$ of *task private ports*. Given a port $p \in \mathsf{Ports}$, we use notation such as $\mathsf{Type}[p]$ for the type of p, and $\mathsf{init}[p]$ for the initial value of p. A *valuation* for a set $\mathsf{P} \subseteq \mathsf{Ports}$ of ports is a function that maps each port $p \in \mathsf{P}$ to a value in $\mathsf{Type}[p]$. We write $\mathsf{Vals}[\mathsf{P}]$ for the set of valuations for P.

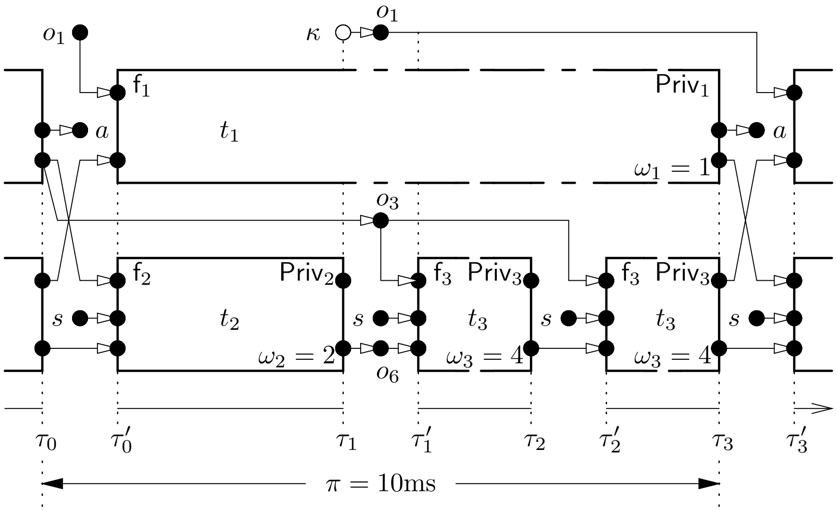

Fig. 10. The time line for the mode switch η at time τ_1

2. A set of *task declarations.* A task declaration $(t, \mathsf{In}, \mathsf{Out}, \mathsf{Priv}, \mathsf{f})$ consists of a task name t, a set $\mathsf{In} \subseteq \mathsf{InPorts}$ of *input ports*, a set $\mathsf{Out} \subseteq \mathsf{OutPorts}$ of *output ports*, a set $\mathsf{Priv} \subseteq \mathsf{PrivPorts}$ of *private ports*, and a *task function* f: $\mathsf{Vals}[\mathsf{In} \cup \mathsf{Priv}] \rightarrow \mathsf{Vals}[\mathsf{Out} \cup \mathsf{Priv}]$. If $(t, \mathsf{In}, \mathsf{Out}, \mathsf{Priv}, \cdot)$ and $(t', \mathsf{In}', \mathsf{Out}', \mathsf{Priv}', \cdot)$ are distinct task declarations, then we require that $t \neq t'$ and $\mathsf{In} \cap \mathsf{In}' = \mathsf{Priv} \cap \mathsf{Priv}' = \emptyset$. Tasks may share output ports as long as the tasks are not invoked in the same mode; see below. We write Tasks for the set of declared task names.
3. A set of *driver declarations.* A driver declaration $(d, \mathsf{Src}, \mathsf{g}, \mathsf{Dst}, \mathsf{h})$ consists of a driver name d, a set $\mathsf{Src} \subseteq \mathsf{Ports}$ of *source ports*, a *driver guard* g: $\mathsf{Vals}[\mathsf{Src}] \rightarrow \mathbb{B}$, a set $\mathsf{Dst} \subseteq \mathsf{Ports}$ of *destination ports*, and a *driver function* h: $\mathsf{Vals}[\mathsf{Src}] \rightarrow \mathsf{Vals}[\mathsf{Dst}]$. When the driver d is called, the guard g is evaluated, and if the result is *true*, then the function h is executed. We require that all driver names are uniquely declared, and we write $\mathsf{Drivers}$ for the set of declared driver names.
4. A set of *mode declarations.* A mode declaration $(m$, π, $\mathsf{ModePorts}$, $\mathsf{Invokes}$, $\mathsf{Updates}$, $\mathsf{Switches})$ consists of a mode name m, a *mode period* $\pi \in \mathbb{Q}$, a set $\mathsf{ModePorts} \subseteq \mathsf{OutPorts}$ of *mode ports*, a set $\mathsf{Invokes}$ of *task invocations*, a set $\mathsf{Updates}$ of *actuator updates*, and a set $\mathsf{Switches}$ of *mode switches.* We require that all mode names are uniquely declared, and we write Modes for the set of declared mode names.
   (a) Each task invocation $(\omega_{task}, t, d) \in \mathsf{Invokes}[m]$ consists of a *task frequency* $\omega_{task} \in \mathbb{N}$, a task $t \in \mathsf{Tasks}$ such that $\mathsf{Out}[t] \subseteq \mathsf{ModePorts}[m]$, and a *task driver* $d \in \mathsf{Drivers}$ such that $\mathsf{Src}[d] \subseteq \mathsf{ModePorts}[m] \cup \mathsf{SensePorts}$ and $\mathsf{Dst}[d] = \mathsf{In}[t]$. The invoked task t only updates mode ports; the task

driver d reads only mode and sensor ports, and updates the input ports of t. If $(\cdot, t, \cdot)$ and $(\cdot, t', \cdot)$ are distinct task invocations in $\mathsf{Invokes}[m]$, then we require that $t \neq t'$ and $\mathsf{Out}[t] \cap \mathsf{Out}[t'] = \emptyset$; that is, tasks sharing output ports must not be invoked in the same mode.

(b) Each actuator update $(\omega_{act}, d) \in \mathsf{Updates}[m]$ consists of an *actuator frequency* $\omega_{act} \in \mathbb{N}$, and an *actuator driver* $d \in \mathsf{Drivers}$ such that $\mathsf{Src}[d] \subseteq \mathsf{ModePorts}[m]$ and $\mathsf{Dst}[d] \subseteq \mathsf{ActPorts}$. The actuator driver d reads only mode ports, no sensor ports, and updates only actuator ports. If $(\cdot, d)$ and $(\cdot, d')$ are distinct actuator updates in $\mathsf{Updates}[m]$, then we require that $\mathsf{Dst}[d] \cap \mathsf{Dst}[d'] = \emptyset$; that is, in each mode, an actuator can be updated by at most one driver.

(c) Each mode switch $(\omega_{switch}, m', d) \in \mathsf{Switches}[m]$ consists of a *mode switch frequency* $\omega_{switch} \in \mathbb{N}$, a *target mode* $m' \in \mathsf{Modes}$, and a *mode driver* $d \in \mathsf{Drivers}$ such that $\mathsf{Src}[d] \subseteq \mathsf{ModePorts}[m] \cup \mathsf{SensePorts}$ and $\mathsf{Dst}[d] = \mathsf{ModePorts}[m']$. The mode driver d reads only mode and sensor ports, and updates the mode ports of the target mode m'. If $(\cdot, \cdot, d)$ and $(\cdot, \cdot, d')$ are distinct mode switches in $\mathsf{Switches}[m]$, then we require that for all valuations $v \in \mathsf{Vals}[\mathsf{Ports}]$ either $\mathsf{g}[d](v) = \mathit{false}$ or $\mathsf{g}[d'](v) = \mathit{false}$. It follows that all mode switches are deterministic.

5. A *start mode* $\mathsf{start} \in \mathsf{Modes}$.

The program is *well-timed* if for all modes $m \in \mathsf{Modes}$, all task invocations $(\omega_{task}, t, \cdot) \in \mathsf{Invokes}[m]$, and all mode switches $(\omega_{switch}, m', \cdot) \in \mathsf{Switches}[m]$, if $\omega_{task}/\omega_{switch} \not\in \mathbb{N}$, then there exists a task invocation $(\omega'_{task}, t, \cdot) \in \mathsf{Invokes}[m']$ with $\pi[m]/\omega_{task} = \pi[m']/\omega'_{task}$. The well-timedness condition ensures that mode switches do not terminate tasks: if a mode switch occurs when a task may not be completed, then the same task must be present also in the target mode.

3.2 Operational semantics

The *mode frequencies* of a mode $m \in \mathsf{Modes}$ include (i) the task frequencies ω_{task} for all task invocations $(\omega_{task}, \cdot, \cdot) \in \mathsf{Invokes}[m]$, (ii) the actuator frequencies ω_{act} for all actuator updates $(\omega_{act}, \cdot) \in \mathsf{Updates}[m]$, and (iii) the mode switch frequencies ω_{switch} for all mode switches $(\omega_{switch}, \cdot, \cdot) \in \mathsf{Switches}[m]$. The least common multiple of the mode frequencies of m is called the number of *units* of the mode m, and is denoted $\omega_{max}[m]$. A *program configuration* $C = (\tau, m, u, v, \sigma_{active})$ consists of a *time stamp* $\tau \in \mathbb{Q}$, a mode $m \in \mathsf{Modes}$, an integer $u \in \{0, \ldots, \omega_{max}[m] - 1\}$ called the *unit counter*, a valuation $v \in \mathsf{Vals}[\mathsf{Ports}]$ for all ports, and a set $\sigma_{active} \subseteq \mathsf{Tasks}$ of *active tasks*. The set $\sigma_{active} \subseteq \mathsf{Tasks}$ contains all tasks that are logically running, whether or not they are physically running by expending CPU time.

A program configuration is updated essentially as follows: first, some tasks are completed (i.e., removed from the active set); second, some actuators are updated; third, a mode switch may occur; fourth, some new tasks are activated. We therefore need the following definitions:

– A task invocation $(\omega_{task}, t, \cdot) \in \mathsf{Invokes}[m]$ is *completed* at configuration C if $t \in \sigma_{active}$ and $u \cdot \omega_{task}/\omega_{max}[m] \in \mathbb{N}$.
– An actuator update $(\omega_{act}, d) \in \mathsf{Updates}[m]$ is *enabled* at configuration C if $u \cdot \omega_{act}/\omega_{max}[m] \in \mathbb{N}$ and $\mathsf{g}[d](v) = true$.
– A mode switch $(\omega_{switch}, \cdot, d) \in \mathsf{Switches}[m]$ is *enabled* at configuration C if $u \cdot \omega_{switch}/\omega_{max}[m] \in \mathbb{N}$ and $\mathsf{g}[d](v) = true$.
– A task invocation $(\omega_{task}, \cdot, d) \in \mathsf{Invokes}[m]$ is *enabled* at configuration C if $u \cdot \omega_{task}/\omega_{max}[m] \in \mathbb{N}$ and $\mathsf{g}[d](v) = true$.

For a program configuration C and a set $\mathsf{P} \subseteq \mathsf{Ports}$, we write $C[\mathsf{P}]$ for the valuation in $\mathsf{Vals}[\mathsf{P}]$ that agrees with C on the values of all ports in P. The program configuration C_{succ} is a *successor configuration* of $C = (\tau, m, u, v, \sigma_{active})$ if C_{succ} results from C by the following nine steps. These are the steps a Giotto program performs whenever it is invoked, initially at time $\tau = 0$ with $u = 0$ and $\sigma_{active} = \emptyset$:

1. [**Task output and private ports**] Let $\sigma_{completed}$ be the set of tasks t such that a task invocation of the form $(\cdot, t, \cdot) \in \mathsf{Invokes}[m]$ is completed at configuration C. Consider a port $p \in \mathsf{OutPorts} \cup \mathsf{PrivPorts}$. If $p \in \mathsf{Out}[t] \cup \mathsf{Priv}[t]$ for some task $t \in \sigma_{completed}$, then define $v_{task}(p) = \mathsf{f}[t](C[\mathsf{In}[t] \cup \mathsf{Priv}[t]])(p)$; otherwise, define $v_{task}(p) = v(p)$. This gives the new values of all task output and private ports. Note that ports are persistent in the sense that they keep their values unless they are modified. Let C_{task} be the configuration that agrees with v_{task} on the values of $\mathsf{OutPorts} \cup \mathsf{PrivPorts}$, and otherwise agrees with C.
2. [**Actuator ports**] Consider a port $p \in \mathsf{ActPorts}$. If $p \in \mathsf{Dst}[d]$ for some actuator update $(\cdot, d) \in \mathsf{Updates}[m]$ that is enabled at configuration C_{task}, then define $v_{act}(p) = \mathsf{h}[d](C_{task}[\mathsf{Src}[d]])(p)$; otherwise, define $v_{act}(p) = v(p)$. This gives the new values of all actuator ports. Let C_{act} be the configuration that agrees with v_{act} on the values of $\mathsf{ActPorts}$, and otherwise agrees with C_{task}.
3. [**Sensor ports**] Consider a port $p \in \mathsf{SensePorts}$. Let $v_{sense}(p)$ be any value in $\mathsf{Type}[p]$; that is, sensor ports change nondeterministically. This is not done by the Giotto program, but by the environment. All other parts of a configuration are updated deterministically, by the Giotto program. Let C_{sense} be the configuration that agrees with v_{sense} on the values of $\mathsf{SensePorts}$, and otherwise agrees with C_{act}.
4. [**Target mode**] If a mode switch $(\cdot, m_{target}, \cdot) \in \mathsf{Switches}[m]$ is enabled at configuration C_{sense}, then define $m' = m_{target}$; otherwise, define $m' = m$. This determines if there is a mode switch. Recall that at most one mode switch can be enabled at any configuration. Let C_{target} be the configuration with mode m' that otherwise agrees with C_{sense}.
5. [**Mode ports**] Consider a port $p \in \mathsf{OutPorts}$. If $p \in \mathsf{Dst}[d]$ for some mode switch $(\cdot, \cdot, d) \in \mathsf{Switches}[m]$ that is enabled at configuration C_{sense}, then define $v_{mode}(p) = \mathsf{h}[d](C_{target}[\mathsf{Src}[d]])(p)$; otherwise, we define $v_{mode}(p) = C_{target}[\mathsf{OutPorts}](p)$. This gives the new values of all mode ports of the target mode. Note that mode switching updates also the output ports of all tasks t that are logically running. This does not affect the execution of t. When

t completes, its output ports are again updated, by t. Let C_{mode} be the configuration that agrees with v_{mode} on the values of $\mathsf{OutPorts}$, and otherwise agrees with C_{target}.

6. [**Unit counter**] If no mode switch in $\mathsf{Switches}[m]$ is enabled at configuration C_{sense}, then define $u' = (u+1) \bmod \omega_{max}[m]$. Otherwise, suppose that a mode switch is enabled at configuration C_{sense} to the target mode m'. Let $\sigma_{running} = \sigma_{active} \setminus \sigma_{completed}$. If $\sigma_{running} = \emptyset$, then define $u' = 1$. Otherwise, let $u_{complete}$ be the least common multiple of the set $\{\omega_{max}[m]/\omega_{task} \mid (\omega_{task}, t, \cdot) \in \mathsf{Invokes}[m]$ for some $t \in \sigma_{running}\}$; this is the least number of units of m at which all running tasks complete simultaneously. Let u_{actual} be the least multiple of $u_{complete}$ such that $u_{actual} \geq u$; this is the earliest unit number after u at which all running tasks complete simultaneously. Let $\delta = (\pi[m]/\omega_{max}[m]) \cdot (u_{actual} - u)$; this is the duration until the next simultaneous completion point. Let $u_{togo} = (\omega_{max}[m']/\pi[m']) \cdot \delta$; this is the number of units of the target mode m' until the next simultaneous completion point. Finally, define $u' = (1 - u_{togo}) \bmod \omega_{max}[m']$; this is the unit number in mode m' with $u_{togo} - 1$ units to go until the last simultaneous completion point in a round of mode m'. Thus a mode switch always jumps as close as possible to the end of a round of the target mode. Let C_{unit} be the configuration with the unit counter u' that otherwise agrees with C_{mode}.
7. [**Task input ports**] Consider a port $p \in \mathsf{InPorts}$. If $p \in \mathsf{Dst}[d]$ for some task invocation $(\cdot, \cdot, d) \in \mathsf{Invokes}[m']$ that is enabled at configuration C_{unit}, then define $v_{input}(p) = \mathsf{h}[d](C_{unit}[\mathsf{Src}[d]])(p)$; otherwise, define $v_{input}(p) = v(p)$. This gives the new values of all task input ports. Let C_{input} be the configuration that agrees with v_{input} on the values of $\mathsf{InPorts}$, and otherwise agrees with C_{unit}.
8. [**Active tasks**] Let $\sigma_{enabled}$ be the set of tasks t such that a task invocation of the form $(\cdot, t, \cdot) \in \mathsf{Invokes}[m']$ is enabled at configuration C_{input}. The new set of active tasks is $\sigma'_{active} = (\sigma_{active} \setminus \sigma_{completed}) \cup \sigma_{enabled}$. Let C_{active} be the configuration with the set σ'_{active} of active tasks that otherwise agrees with C_{input}.
9. [**Time stamp**] The next time instant at which the Giotto program is invoked is $\tau' = \tau + \pi[m']/\omega_{max}[m']$. An implementation may use a timer interrupt set to τ'. Let C_{succ} be the configuration with the time stamp τ' that otherwise agrees with C_{active}.

An *execution* of a Giotto program is an infinite sequence $C_0, C_1, C_2, \ldots$ of program configurations C_i such that (i) $C_0 = (0, \mathsf{start}, 0, v, \emptyset)$ with $v(p) = \mathsf{init}[p]$ for all ports $p \in \mathsf{Ports}$, and (ii) C_{i+1} is a successor configuration of C_i for all $i \geq 0$. Note that there can be a mode switch at time 0, but there can never be two mode switches in a row without any time passing.

4 Annotated Giotto

A Giotto program can in principle be run on a single sufficiently fast CPU, independent of the number of modes and tasks. However, taking into account performance constraints, the timing requirements of a program may or may not be achievable on a single CPU. Additionally, a particular application may require that tasks be located in specific places, e.g., close to the physical processes that the tasks control, or on processors particularly suited for the operations of the tasks. Lastly, in order to achieve fault tolerance, redundant, isolated CPUs may be desirable. For these reasons, it may be necessary to distribute the work of a Giotto program between multiple CPUs. In order to aid the compilation on distributed, possibly heterogeneous, platforms, we allow the annotation of Giotto programs with platform constraints. While pure Giotto is platform-independent, annotated Giotto contains directives for mapping and scheduling a program on a particular platform. An *annotated Giotto program* is a formal refinement of a pure Giotto program in the sense that the logical semantics of the pure Giotto program, as defined in Section 3.2, is preserved.

Annotated Giotto consists of multiple annotation levels. Conceptually, annotations at the higher levels occur prior to annotations at the lower levels. This structured approach has several advantages. First, it permits the incremental refinement of a pure Giotto program into an executable image. Specifically, it allows a modular architecture for the Giotto compiler, with separate modules for mapping and scheduling. Second, it enables the generation of formal models at all annotation levels. These models can be checked for consistency with the annotations at the higher levels [VB93], in particular, for consistency with the pure Giotto semantics.

Formally, a *hardware configuration* consists of a set of *hosts* and a set of *networks*. A host is a CPU that can execute Giotto tasks. A network connects two or more hosts and can transport values. The passing of a value from one port to another (e.g., from a sensor port or a task output port to a task input port) is called a *connection*. Annotated Giotto consists of the following three levels of annotations:

Giotto-H (H for "hardware") specifies a set of hosts, a set of networks, and worst-case execution time information. The WCET information includes the time needed to execute tasks on hosts, and the time needed to transfer connections on networks.

Giotto-HM (M for "map") specifies, in addition, an assignment of task invocations to hosts, and an assignment of connections to networks. The same task, when invoked in different modes, may be assigned to different hosts. The mapping of a task invocation also determines the physical location of the task output ports.

Giotto-HMS (S for "schedule") specifies, in addition, scheduling information for each host and network. For example, every task invocation may be assigned a priority, and every connection may be assigned a time slot.

An annotation is *complete* if it fully determines all assignments at its annotation level, and is *partial* otherwise. In particular, a complete HM annotation maps every task invocation to a host, and maps every connection to a network. The information that a complete Giotto-HMS program needs to specify may vary depending on the scheduling strategy of the RTOS on the hosts, and on the communication protocols on the networks. For instance, a Giotto-HMS program may specify priorities for task invocations, relative deadlines, or time slots, depending on whether the underlying RTOS uses a priority-driven, deadline-driven, or time-triggered scheduling strategy.

An annotated Giotto program may be *overconstrained*, in that it does not permit any execution that is consistent with the annotations. An annotated Giotto program is *valid* if (i) it is not overconstrained, and (ii) it is consistent with the semantics of the underlying pure Giotto program. A Giotto compiler takes a partially annotated program and can have one of three outcomes: either it determines that the input program is not valid, or it produces a completely annotated, valid HMS refinement (which can then be turned into executable code), or it gives up and asks for more annotations from the programmer. For answering the validity question, a Giotto compiler can generate a formal model on each annotation level. For example, the constraints imposed by a Giotto-HM program can be expressed as a graph of conditional process graphs [EKP$^+$98], one for each mode, which can be checked for validity. A completely annotated Giotto-HMS program, provided it is not overconstrained, specifies a unique behavior of all hosts and networks for every given real-time trace of sensor valuations. These behaviors can be checked for conformance against the higher-level graph model to guarantee Giotto semantics. Given a partially annotated Giotto program, a compiler can generate the missing HMS-annotations based on holistic schedulability analysis for distributed real-time systems that use time-triggered communication protocols [TC94]. Such a compiler can be evaluated along several dimensions: (i) how many annotations it requires to generate valid code, and (ii) what the cost is of the generated code. For instance, a compiler can use a cost function that minimizes jitter of the actuator updates.

5 Discussion

While many of the individual elements of Giotto are derived from the literature, we believe that the study of strictly time-triggered task invocation together with strictly time-triggered mode switching as a possible organizing principle for *abstract, platform-independent real-time programming* is an important, novel step towards separating *reactivity*, i.e., functionality and timing requirements, from *schedulability*, i.e., scheduling guarantees on computation and communication. Giotto decomposes the development process of embedded control software into high-level real-time programming of reactivity and low-level real-time scheduling of computation and communication. Programming in Giotto is real-time programming in terms of the requirements of control designs, i.e., their reactivity, not their schedulability.

The strict separation of reactivity from schedulability is achieved in Giotto through time- and value-determinism: given a real-time trace of sensor valuations, the corresponding real-time trace of actuator valuations produced by a Giotto program is uniquely determined. The separation of reactivity from schedulability has at least two important ramifications. First, reactive (i.e., functional and timing) properties of a Giotto program may be subject to formal verification against a mathematical model of the control design [Hen00]. Second, Giotto is compatible with any scheduling algorithm, which therefore becomes a parameter of the Giotto compiler. There are essentially two reasons why even the best Giotto compiler may fail to generate an executable: (i) not enough platform utilization, or (ii) not enough platform performance. Then, independently of the program's reactivity, utilization can be improved by a better scheduling module, while performance can be improved by faster hardware or leaner software that implements the actual functionality (i.e., the individual tasks) more efficiently.

5.1 Related work

Giotto is inspired by the time-triggered architecture (TTA) [Kop97], which first realized the time-triggered paradigm for meeting hard real-time constraints in safety-critical distributed settings. However, while the TTA encompasses a hardware architecture and communication protocols, Giotto provides a hardware-independent and protocol-independent abstract programmer's model for time-triggered applications. Giotto can be implemented on any platform that provides sufficiently accurate clock primitives or supports a clock synchronization scheme. The TTA is thus a natural platform for Giotto programs.

Giotto is similar to architecture description languages (ADLs) [Cle96]. Like Giotto, ADLs shift the programmer's perspective from small-grained features such as lines of code to large-grained features such as tasks, modes, and inter-component communication, and they allow the compilation of scheduling code to connect tasks written in conventional programming languages. The design methodology [KZF^{+}91] for the MARS system, a predecessor of the TTA, distinguishes in a similar way *programming-in-the-large* and *programming-in-the-small*. The inter-task communication semantics of Giotto is particularly similar to the MetaH language [Ves97], which is designed for real-time, distributed avionics applications. MetaH supports periodic real-time tasks, multi-modal control, and distributed implementations. Giotto can be viewed as capturing the time-triggered fragment of MetaH in an abstract and formal way. In particular, unlike MetaH, Giotto specifies not only inter-task communication but also mode switches in a time-triggered fashion, and it does not constrain the implementation to a particular scheduling scheme.

The goal of Giotto —to provide a platform-independent programming abstraction for real-time systems— is shared also by the synchronous reactive programming languages [Hal93], such as Esterel [Ber00], Lustre [HCRP91], or Signal [BGJ91]. While the synchronous reactive languages are designed around zero-delay value propagation, Giotto is based on the formally weaker notion of unit-delay value propagation, because in Giotto, scheduled computation (i.e., the execution of tasks) takes time, and synchronous computation (i.e., the execution of drivers) consists only of independent, non-interacting processes. This decision shifts the focus and the level of abstraction in essential ways. In particular, for analysis and compilation, the burden for the well-definedness of values is shifted from logical fixed-point considerations to physical constraints about platform resources and performance (in Giotto all values are, logically, always well-defined). Thus, Giotto can be seen as identifying a class of synchronous reactive programs that support (i) typical real-time control applications as well as (ii) efficient schedule synthesis and code generation.

5.2 Giotto implementations

We briefly review the existing Giotto implementations. The first implementation of Giotto was a simplified Giotto run-time system on a distributed platform of Lego Mindstorm robots. The robots use infrared transceivers for communication. Then we implemented a full Giotto run-time system on a distributed platform of Intel x86 robots running the real-time operating system VxWorks. The robots use wireless Ethernet for communication. We also implemented a Giotto program running on five robots, three Lego Mindstorms and two x86-based robots, to demonstrate Giotto's applicability for heterogeneous platforms. The communication between the Mindstorms and the x86 robots is done by an infrared-Ethernet bridge implemented on a PC. For an informal discussion of this implementation, and embedded control systems development with Giotto in general, we refer to the earlier report [HHK01].

In collaboration with Marco Sanvido and Walter Schaufelberger at ETH Zürich, we have been working on a high-performance implementation of a Giotto run-time system on a single-processor platform that controls an autonomously flying model helicopter [San99]. The implementation language is a subset of Oberon for embedded real-time systems [Wir99]. The existing helicopter control software has been reimplemented as a combination of a Giotto program and Oberon code that implements the controller tasks. We have implemented a Giotto compiler that generates, from such a Giotto program, the Giotto executable as Oberon code. The executable uses the Giotto run-time system on the helicopter to control the hard real-time scheduling of the navigation and controller software. We have also been working on an implementation of a virtual hard real-time scheduling machine [Kir01], as an alternative to the Giotto run-time system on the helicopter. The Giotto compiler can generate machine code of the virtual machine instead of Giotto executables in Oberon. This approach has two advantages: (i) code generation is more flexible, because the virtual machine semantics is finer-grained than the API of the Giotto run-time system, and (ii) increased portability of the generated code.

Acknowledgments. We thank Rupak Majumdar for implementing a prototype Giotto compiler for Lego Mindstorms robots. We thank Dmitry Derevyanko and Winthrop Williams for building the Intel x86 robots. We thank Edward Lee and Xiaojun Liu for help with implementing Giotto as a "model of computation" in Ptolemy II [DGH$^+$99]. We thank Marco Sanvido for his suggestions on the design of the Giotto drivers.

References

[Ber00] G. Berry. The foundations of Esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pp. 425–454. MIT Press, 2000.

[BGJ91] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: The Signal language and its semantics. *Science of Computer Programming*, 16:103–149, 1991.

[Cle96] P. Clements. A survey of architecture description languages. In *Proc. 8th International Workshop on Software Specification and Design*, pp. 16–25. IEEE Computer Society Press, 1996.

[Col99] R.P.G. Collinson. Fly-by-wire flight control. *Computing & Control Engineering*, 10:141–152, 1999.

[DGH+99] J. Davis, M. Goel, C. Hylands, B. Kienhuis, E.A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, and Y. Xiong. *Ptolemy II: Heterogeneous Concurrent Modeling and Design in Java*. Technical Report UCB/ERL-M99/44, University of California, Berkeley, 1999.

[EKP+98] P. Eles, K. Kuchcinski, Z. Peng, A. Doboli, and P. Pop. Process scheduling for performance estimation and synthesis of hardware/software systems. In *Proc. 24th EUROMICRO Conference*, pp. 168–175, 1998.

[Hal93] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer, 1993.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proc. IEEE*, 79:1305–1320, 1991.

[Hen00] T.A. Henzinger. Masaccio: A formal model for embedded components. In *Proc. First IFIP International Conference on Theoretical Computer Science*, LNCS 1872, pp. 549–563. Springer-Verlag, 2000.

[HHK01] T.A. Henzinger, B. Horowitz, and C.M. Kirsch. Embedded control systems development with Giotto. In *Proc. SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, ACM Press, 2001.

[Kir01] C.M. Kirsch. *The Embedded Machine*. Technical Report UCB/CSD-01-1137, University of California, Berkeley, 2001.

[Kop97] H. Kopetz. *Real-time Systems: Design Principles for Distributed Embedded Applications*. Kluwer, 1997.

[KZF+91] H. Kopetz, R. Zainlinger, G. Fohler, H. Kantz, P. Puschner, and W. Schütz. The design of real-time systems: From specification to implementation and verification. *IEE/BCS Software Engineering Journal*, 6:72–82, 1991.

[LRR92] D. Langer, J. Rauch, and M. Rößler. *Real-time Systems: Engineering and Applications*, chapter 14, pp. 369–395. Kluwer, 1992.

[San99] M. Sanvido. *A Computer System for Model Helicopter Flight Control; Technical Memo 3: The Software Core*. Technical Report 317, Institute of Computer Systems, ETH Zürich, 1999.

[TC94] K. Tindell and J. Clark. Holistic schedulability for distributed hard real-time systems. *Microprocessing and Microprogramming*, 40:117–134, 1994.

[VB93] S. Vestal and P. Binns. Scheduling and communication in MetaH. In *Proc. 14th Annual Real-Time Systems Symposium*. IEEE Computer Society Press, 1993.

[Ves97] S. Vestal. MetaH support for real-time multi-processor avionics. In *Proc. Fifth International Workshop on Parallel and Distributed Real-Time Systems*, pp. 11–21. IEEE Computer Society Press, 1997.

[Wir99] N. Wirth. *A Computer System for Model Helicopter Flight Control; Technical Memo 2: The Programming Language Oberon SA*, second edition. Technical Report 285, Institute of Computer Systems, ETH Zürich, 1999.