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 hybrid control application consists of periodic software tasks together with a mode switching logic for enabling and disabling tasks. Pure Giotto specifies time-triggered sensor readings, task invocations, and mode switches independent of any implementation platform. Pure 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.

We illustrate the use of Giotto by coordinating a heterogeneous flock of Intel x86 robots and Lego Mindstorms robots.

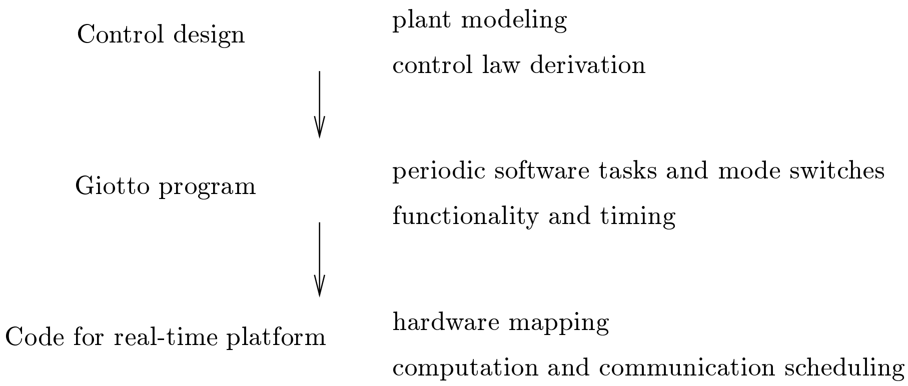

Figure 1: Real-time control system design with Giotto

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. The time-triggered nature of Giotto achieves timing predictability, which makes Giotto particularly suitable for safety-critical applications.

Traditional control design happens at a mathematical level of abstraction, with the control engineer manipulating continuous differential equations and possibly discrete mode switches using tools such as Matlab or Xmath. 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.

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.

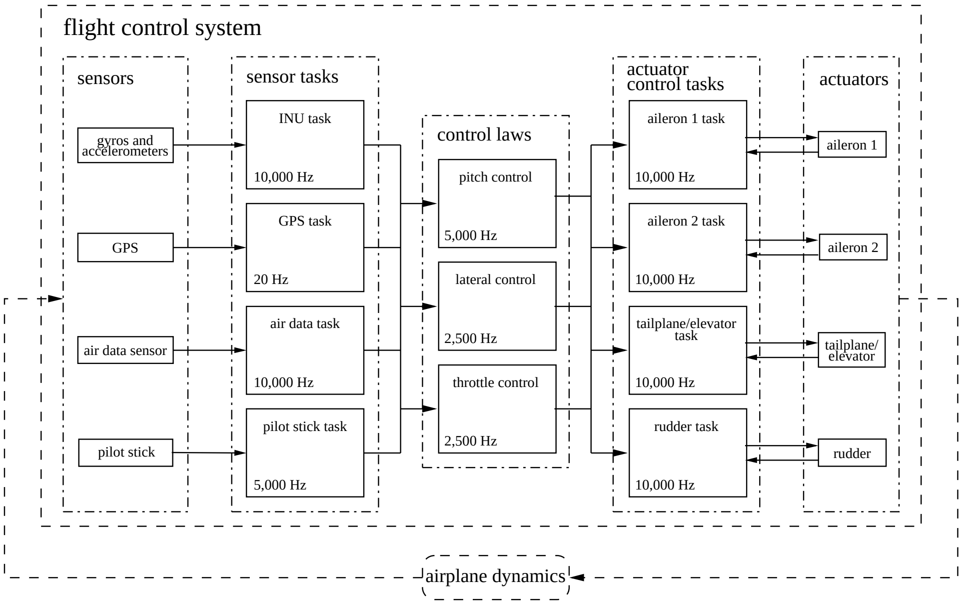

Figure 2: A fly-by-wire flight control system

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. The periodic invocation of tasks and the mode switching are triggered by real time. For example, one task t_1 may be invoked every 3 ms, another task t_2 every 1 ms, 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 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) 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, BGP00, FMD$^+$00b, FMD$^+$00a]). All the Giotto compiler needs to ensure is that the logical semantics of Giotto —functionality and timing— is preserved. In the last section, we will introduce a mechanism for annotating a Giotto program with platform constraints, which can 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, 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.

In Section 2, we begin by describing a typical aerospace control scenario, which motivates the design of Giotto. We give an informal introduction to Giotto in Section 3, followed by formal definitions of syntax (Section 4) and semantics (Section 5). In Section 6, we use Giotto to coordinate a heterogeneous flock of Intel x86 robots and Lego Mindstorms robots. We conclude with a brief discussion of ongoing work, specifically, a hierarchical version of Giotto (Section 7) Giotto annotations for compilation on static priority preemptive RTOS (Section 8). In the final Section 9, we relate Giotto to other work.

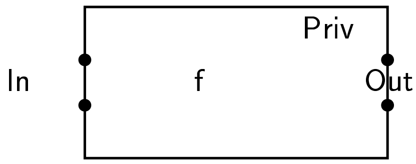

Figure 3: A task t

2 Motivating example

As a motivating example, we describe a fly-by-wire flight control system [LRR92, Col99]. The control system 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 50,000 times per second, whereas the pilot's stick outputs its measurement only 5,000 times per second.

Three separate control laws —for pitch, lateral, and throttle control— need to be computed. 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 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 actuator 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 autopilot mode, the control system takes inputs from a supervisory flight planner, instead of from the pilot's stick. In 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.

3 Informal description of Giotto

In Giotto all data is communicated through ports. A port represents a typed variable with a unique location in a globally shared name space (an implementation of the Giotto is, of course, not required to be a shared memory system; we use the global name space for ports as a virtual concept to simplify the definition of Giotto). A port is persistent in the sense that a port keeps its value over time. There are mutually disjoint sets of input, output, environment, and entry ports in a Giotto program. Each input and output port belongs to a Giotto task. The environment ports represent sensors, and are updated nondeterministically by the environment of the Giotto program. Entry ports will be discussed later in the context of modes.

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 ports of t are distinct from all other ports in the Giotto program. In general, a task may have an arbitrary number of input

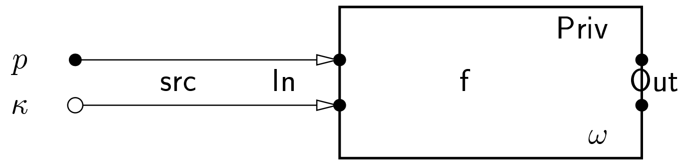

Figure 4: An invocation of task t

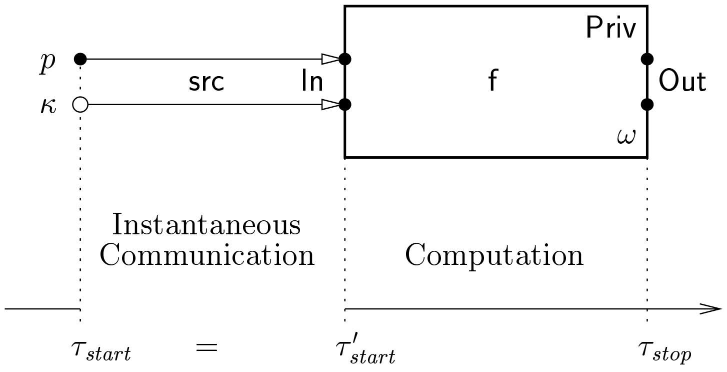

Figure 5: The time line for an invocation of task t

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. For a given platform, the Giotto compiler will need to know the worst-case execution time of f on each CPU. Each invocation of the task t executes the function f once.

Giotto tasks are periodic tasks: they are invoked at regularly spaced points in time. The features of an invocation of the task t are shown in Figure 4. The invocation has a frequency ω given by a non-zero natural number. The real-time frequency of t will be determined later by dividing the real-time period of the current mode by ω. The task invocation specifies connections src from ports or constant values to input ports. The two input ports of t are connected to a port p and a constant value κ. The port p could be an output port of t or of another task, an environment port, or an entry port of the current mode. The time line for an invocation of task t is depicted in Figure 5. The invocation starts at some time τ_{start} with a communication phase in which the value stored in port p and the constant value κ are copied to the input ports In. The Giotto semantics prescribes that the communication phase takes zero time. Instantaneous communication is part of the Giotto logical abstraction from physical implementation details. The instantaneous-communication abstraction can be realized physically by transferring new values at any time before τ_{start}. The communication phase of the invocation of task t is followed by a 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. The Giotto semantics does not specify when, where, and how the computation of f is physically performed between τ_{start} and τ_{stop}. This abstraction and instantaneous communication are the essential ingredients of the Giotto programmer's model.

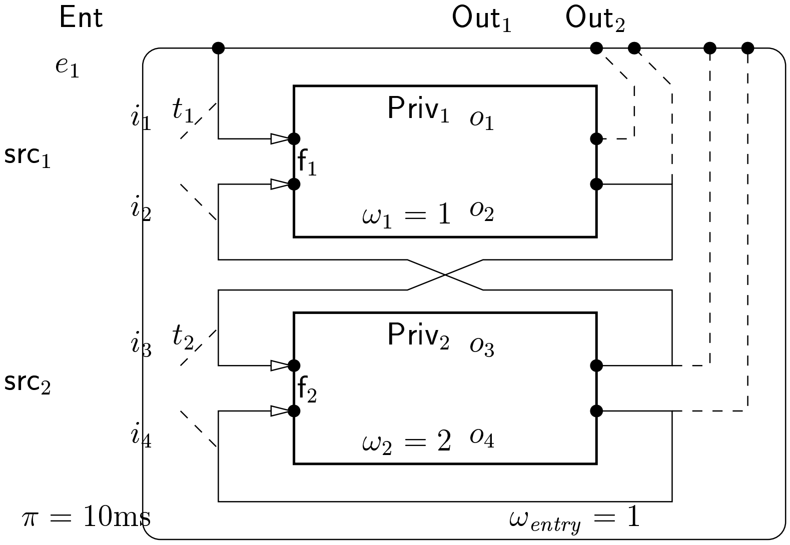

Figure 6: A mode m

A Giotto program consists of a set of modes, each of which repeats the invocation of a set of tasks. The system described by a 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 set of task invocations, a period, a set of entry ports, a set of environment ports, an entry frequency, and a set of mode switches. Figure 6 depicts a mode m, which contains invocations of a task t_1 and a task t_2. The period π of m is 10 ms; that is, while the system is in mode m, its execution repeats the same pattern of task invocations every 10 ms. The entry frequency ω_{entry} of m is one. The entry frequency of a mode specifies how often the mode may be entered (by switching out of another mode) during its period π; in particular, the mode m can be entered only once every 10 ms, at the very beginning of its period. The mode m has a singleton set of entry ports, namely, $\mathsf{Ent} = \{e_1\}$, and no environment ports. The task t_1 has two input ports i_1 and i_2 and two output ports o_1 and o_2, a state Priv_1, and a function f_1. The task t_2 is defined in a similar way. The invocation of t_1 in mode m has a frequency ω_1 of one, which means that t_1 is invoked once every 10 ms while the system is in mode m. The input port i_1 of t_1 is connected to the entry port e_1 of mode m, and the input port i_2 is connected to the output port o_3 of t_2. The invocation of t_2 has a frequency ω_2 of two, which means that t_2 is invoked once every 5 ms, as long as the system is in mode m. The input port i_3 of t_2 is connected to the output port o_2 of task t_1, and the input port i_4 is connected to the output port o_4 of t_2. Note that the output ports of the tasks in m are visible outside the scope of m as indicated by the dashed lines. We will later see that mode switches may connect these output ports to entry ports of other modes.

Figure 7 shows the exact timing of a single round of mode m, which takes 10 ms; while the system in 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. According to the connections of t_1 and t_2, the values stored in e_1 and o_3 are copied to the input ports of t_1, and the values stored in the output ports o_2 and o_4 are copied to the input ports of t_2. The Giotto semantics does not specify how the computations of 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, after 5 ms, at

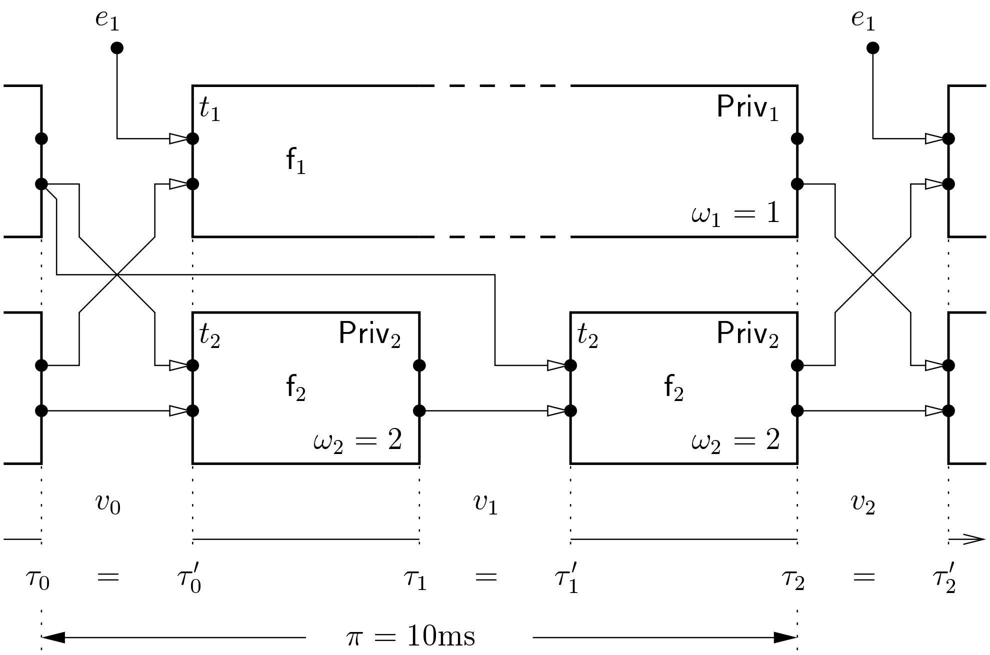

Figure 7: The time line for a round of mode m

time instant τ_1, the results of the computation of f_2 are written to the output ports of t_2. The second invocation of t_2 begins with an instantaneous transfer of values stored in o_4 and o_2 to the input ports of t_2. Note that the value stored in o_2 has not been updated since the time instant τ_0, no matter if physically f_1 has finished its computation before τ_1 or not. Logically, the output values of the invocation of t_1 are not 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. The beginning of the next round shows that the input port i_3 is loaded with the new value produced by t_1.

In order to give an example of mode switching we introduce a second mode m', depicted in Figure 8. The main difference between m and m' is that m' has a third task, t_3, in addition to the tasks t_1 and t_2 of mode m. The task t_3 has a frequency ω_3 of eight in m'. Note that m' also doubles the frequency of t_2 to four. 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_2 is invoked four times per round, every 2.5 ms; and the task t_3 is invoked eight times per round, every 1.25 ms. The connections of t_1 and t_2 in both modes are exactly the same. The entry frequency of m' is two, which means that m' may be entered at the beginning of its round or in the middle, after 5 ms. In addition to the entry port e_1 of mode m, there is a second entry port e_2 in m', which is connected to the input port i_7 of t_3. The input port i_5 is connected to the output port o_4 of t_2, and the input port i_6 is connected to the output port o_5.

A mode switch describes the transition from one mode to another mode. For this purpose, a mode switch specifies an exit frequency, an exit predicate, a target mode, and connections for the entry ports of the target mode. Figure 9 shows a mode switch s from mode m to mode m' with an exit frequency ω_{exit} of two. The exit predicate is evaluated periodically, as specified by the exit

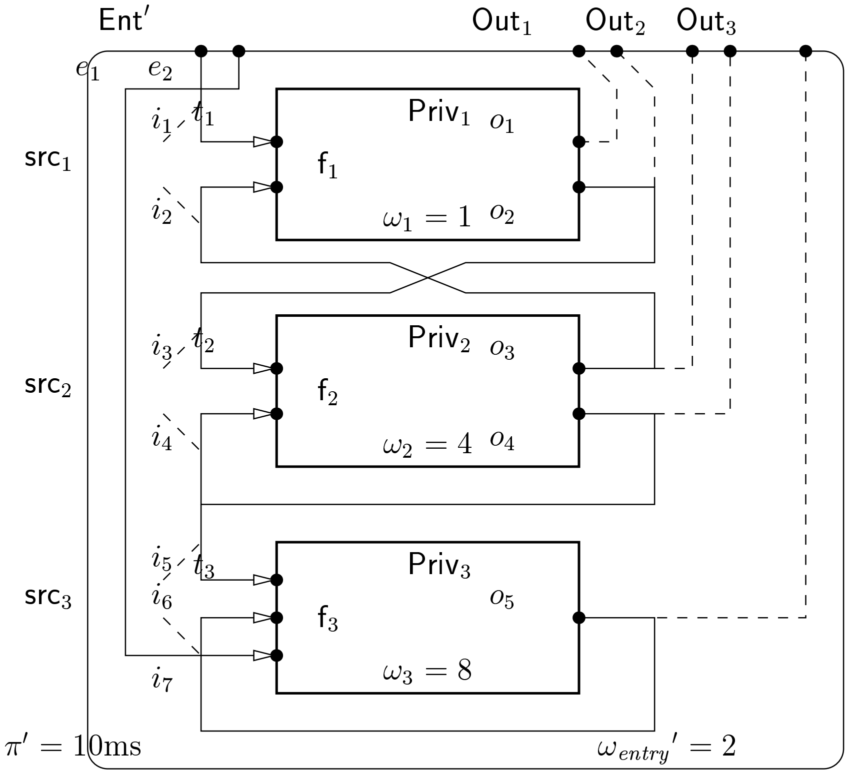

Figure 8: A mode m'

frequency. The exit frequency of two means that the exit predicate is evaluated every 5 ms, in the middle and at the end of each round of mode m. The exit predicate is a boolean-valued condition on the entry and environment ports of m, and on the output ports of the tasks t_1 and t_2 invoked in mode m. If the exit predicate evaluates to true, then a switch to the target mode m' is performed. The mode switch connects the output port o_3 of task t_2 to the entry port e_2 of mode m', and the entry port e_1 is loaded with the constant value κ. Like all Giotto connections, mode switches are logically performed in zero time.

Figure 10 depicts the time line for the mode switch s performed at time τ_1. The system 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. The execution of mode m' starts with an instantaneous communication phase required by the mode switch. At time τ_1, the constant value κ is copied to the entry port e_1 of m', and the value stored in the output port o_3 of t_2 is copied to the entry port e_2. At this point, the mode switch is already finished. All subsequent actions follow the semantics of the target mode m' independently of whether the system entered m' just now through a mode switch, at 5 ms into a round, or whether the system started the current round already in mode m'. Specifically, instantaneous communication phase for the invocations of tasks t_2 and t_3 follow: the input port i_3 of t_2 is loaded with the value from the output port o_2 of t_1; the input port i_4 is loaded with the value from the output port o_4 of t_2; the input port i_5 of t_3 is loaded with the value from the output port o_4 of t_2; the input port i_6 is loaded with the value from the output port o_5 of t_3; and the input port i_7 is loaded with the value from the entry port e_2 of m'. The output port o_5 contains either a well-defined initial value (if this is the first invocation of t_3), or the value computed by the

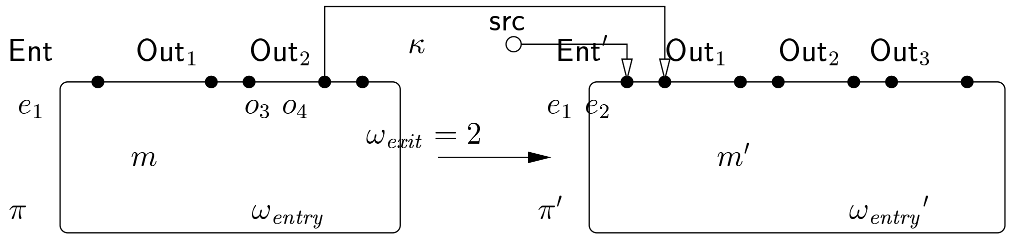

Figure 9: A mode switch s from mode m to mode m'

most recent invocation of t_3. Note that in the communication phase at time τ_5 the input port i_1 of task t_1 connected to the entry port e_1 is loaded with the constant value κ stored in the entry port. In this way, task t_1 may detect that a mode switch took place in the previous round. Since entry ports may be used in connections and exit predicates, a mode with entry ports can be seen as a parameterized version of the same mode without entry ports.

For a mode switch to be legal, the target mode is constrained so that all task invocations that can be interrupted by a mode switch can be continued, logically without preemption, in the target mode. In our example, the mode switch s 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, one. Furthermore, 5 ms into a round must be a legal entry point for m'; that is, the entry frequency of m' has to be (at least) two. All of these conditions are indeed satisfied by the target mode m', and the mode switch s at 5 ms into a round of m is followed simply by the second half of a normal round of m'. If, alternatively, the period of m' were 20 ms, then the frequency of t_1 in m' would have to be two, and the entry frequency of m' would have to be (at least) four.

4 Abstract syntax of Giotto

Rather than specifying a concrete syntax for Giotto, we formally define the components of a Giotto program in a more abstract way; fragments of sample Giotto programs written in a C like concrete syntax can be found Section 6. A Giotto program consists of four 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 $p \neq p'$. We write Ports for the set of declared port names. The set Ports is partitioned into a set $\mathsf{EntryPorts}$ of *mode entry ports*, a set $\mathsf{EnvPorts}$ of *environment 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 write $\mathsf{Type}[p]$ for the type of p, and $\mathsf{init}[p]$ for the initial value of p. For environment ports $p \in \mathsf{EnvPorts}$, the initial value $\mathsf{init}[p]$ is irrelevant and can be omitted.

   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. Given a valuation $v \in \mathsf{Vals}[\mathsf{Ports}]$, we write $v[\mathsf{P}]$ for the restriction of v to the variables in P. A *source* for the ports P from the ports $\mathsf{P}' \subseteq \mathsf{Ports}$ is a function that maps each port $p \in \mathsf{P}$ either to a value in $\mathsf{Type}[p]$, or to a port $p' \in \mathsf{P}'$ such that $\mathsf{Type}[p] = \mathsf{Type}[p']$.

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

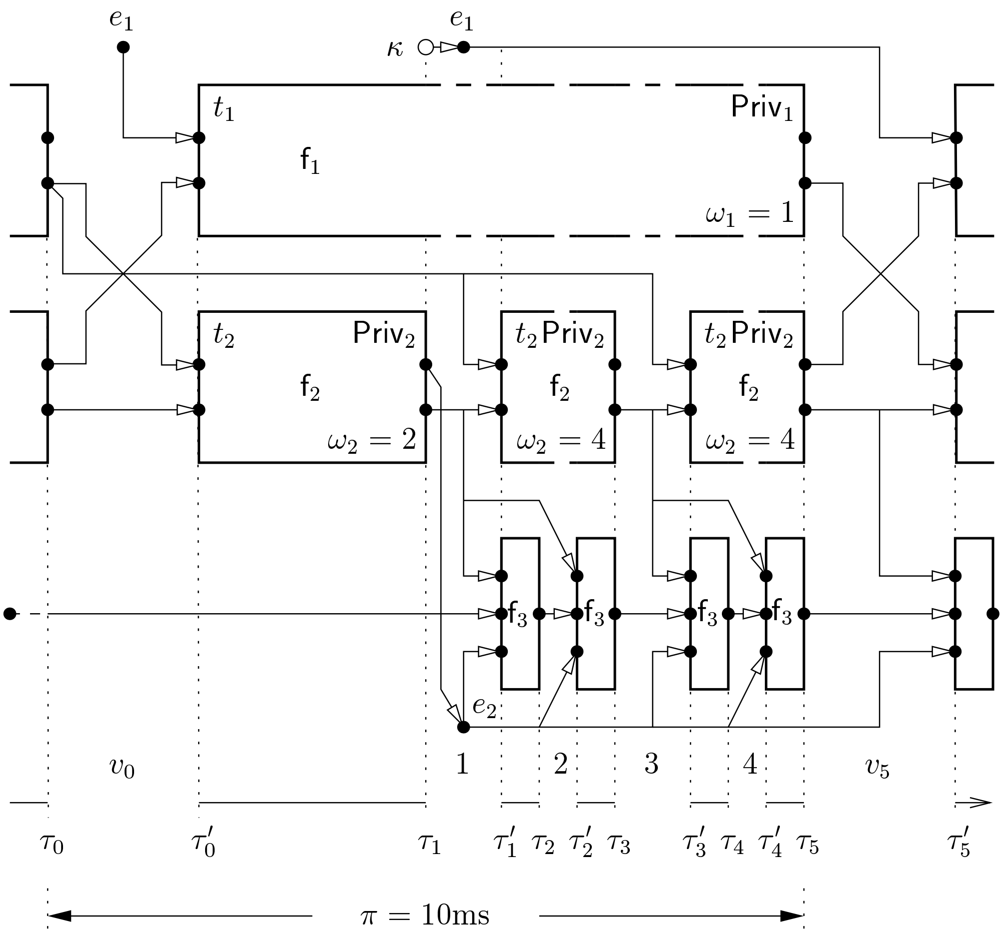

Figure 10: The time line for the mode switch s at time τ_1

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* $\mathsf{f}\colon \mathsf{Vals}[\mathsf{In} \cup \mathsf{Priv}] \to \mathsf{Vals}[\mathsf{Out} \cup \mathsf{Priv}]$. 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 $\mathsf{In} \cap \mathsf{In}' = \mathsf{Out} \cap \mathsf{Out}' = \mathsf{Priv} \cap \mathsf{Priv}' = \emptyset$. We write Tasks for the set of declared task names. Given a task $t \in \mathsf{Tasks}$, we write $\mathsf{In}[t]$ for the set of input ports, $\mathsf{Out}[t]$ for the set of output ports, $\mathsf{Priv}[t]$ for the set of private ports, and $\mathsf{f}[t]$ for the task function.

3. A set of *mode declarations*. A mode declaration $(m, \pi, \omega_{entry}, \mathsf{Ent}, \mathsf{Env}, \mathsf{Invokes}, \mathsf{Switches})$ consists of a mode name m, a *mode period* $\pi \in \mathbb{Q}$, an *entry frequency* $\omega_{entry} \in \mathbb{N}$, a set $\mathsf{Ent} \subseteq \mathsf{EntryPorts}$ of *entry ports*, a set $\mathsf{Env} \subseteq \mathsf{EnvPorts}$ of *environment ports*, a set $\mathsf{Invokes}$ of *task invocations*, and a set $\mathsf{Switches}$ of *mode switches*. Task invocations and mode switches are defined below. We require that all mode names are uniquely declared, and write Modes for the set of declared mode names. Given a mode $m \in \mathsf{Modes}$, we write $\pi[m]$ for the mode period, $\omega_{entry}[m]$ for the entry frequency, $\mathsf{Ent}[m]$ for the set of entry ports, $\mathsf{Invokes}[m]$ for the set of task invocations, and $\mathsf{Switches}[m]$ for the set of mode switches.

   (a) Each task invocation $(t, \omega, \mathsf{src}) \in \mathsf{Invokes}[m]$ consists of a task $t \in \mathsf{Tasks}$, a *task frequency* $\omega \in \mathbb{N}$, and a *task source* src for the input ports $\mathsf{In}[t]$ from the ports in $\mathsf{Ent}[m] \cup \mathsf{Env}[m] \cup$

$\mathsf{Out}[m]$, where $\mathsf{Out}[m]$ is the union of all sets $\mathsf{Out}[t]$ of output ports such that $(t, \cdot, \cdot) \in \mathsf{Invokes}[m]$. If $(t, \cdot, \cdot)$ and $(t', \cdot, \cdot)$ are distinct task invocations in $\mathsf{Invokes}[m]$, then we require that $t \neq t'$.

(b) Each mode switch $(m', \omega_{exit}, \phi_{exit}, \mathsf{src}) \in \mathsf{Switches}[m]$ consists of a *target mode* $m' \in \mathsf{Modes}$, an *exit frequency* $\omega_{exit} \in \mathbb{N}$, an *exit condition* ϕ_{exit}, and a *mode source* src for the entry ports $\mathsf{Ent}[m']$ from the ports in $\mathsf{Ent}[m] \cup \mathsf{Env}[m] \cup \mathsf{Out}[m]$. The exit condition ϕ_{exit} is a function from $\mathsf{Vals}[\mathsf{Ent}[m] \cup \mathsf{Env}[m] \cup \mathsf{Out}[m]]$ to $\mathbb{B}$. If $(\cdot, \cdot, \phi_{exit}, \cdot)$ and $(\cdot, \cdot, \phi'_{exit}, \cdot)$ are distinct mode switches in $\mathsf{Switches}[m]$, then we require that for all port valuations $v \in \mathsf{Vals}[\mathsf{Ent}[m] \cup \mathsf{Env}[m] \cup \mathsf{Out}[m]]$ either $\phi_{exit}(v) = \mathit{false}$ or $\phi'_{exit}(v) = \mathit{false}$. It follows that all mode switches are deterministic.

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

The *mode frequencies* of a mode $m \in \mathsf{Modes}$ include (i) the entry frequency $\omega_{entry}[m]$, (ii) the task frequencies ω for all task invocation $(\cdot, \omega, \cdot) \in \mathsf{Invokes}[m]$, and (iii) the exit frequencies ω_{exit} for all mode switches $(\cdot, \omega_{exit}, \cdot, \cdot) \in \mathsf{Switches}[m]$. The highest mode frequency of m is called the number of *units* of the mode m, and denoted $\omega_{max}[m]$. A Giotto program is *well-timed* if the following conditions are satisfied:

Local harmonic period For all modes $m \in \mathsf{Modes}$ and all mode frequencies ω_1 and ω_2 of m, either $\omega_1/\omega_2 \in \mathbb{N}$ or $\omega_2/\omega_1 \in \mathbb{N}$. Moreover, there exists a mode frequency of m, other than the entry frequency $\omega_{entry}[m]$, which is one.

Global harmonic period For all modes $m \in \mathsf{Modes}$, task invocations $(t, \omega, \mathsf{src}) \in \mathsf{Invokes}[m]$, and mode switches $(m', \omega_{exit}, \cdot, \cdot) \in \mathsf{Switches}[m]$, if $\omega_{exit} > \omega$, then

1. $(\pi[m]/\omega_{exit})/(\pi[m']/\omega_{entry}[m']) \in \mathbb{N}$, and
2. there exists a task invocation $(t, \omega', \mathsf{src}) \in \mathsf{Invokes}[m']$ with $\pi[m]/\omega = \pi[m']/\omega'$.

5 Formal semantics of Giotto

A *program counter* (m, u) consists of a mode $m \in \mathsf{Modes}$, and an integer $u \in \{0, \ldots, \omega_{max}[m] - 1\}$ called the *unit counter*. A *time stamp* τ is a rational number. Given a program counter (m, u) and a task invocation $(t, \omega, \cdot) \in \mathsf{Invokes}[m]$, the task t is *ready* at unit u in mode m if $u \cdot \omega / u[m] \in \mathbb{N}$ (otherwise, the task t is running at unit u in mode m). A mode switch $(\cdot, \omega_{exit}, \phi_{exit}, \cdot) \in \mathsf{Switches}[m]$ is *enabled* at unit u in mode m with respect to the port valuation $v \in \mathsf{Vals}[\mathsf{Ports}]$ if both $u \cdot \omega_{exit}/u[m] \in \mathbb{N}$ and $\phi_{exit}(v[\mathsf{Ent}[m] \cup \mathsf{Env}[m] \cup \mathsf{Out}[m]]) = \mathit{true}$. Let $t_{min}[m]$ be a task with minimal frequency in mode m, and let $\omega_{min}[m]$ denote the frequency of $t_{min}[m]$ in m. An *execution* of a Giotto program is an infinite sequence

$$(\tau_0, c_0, v_0), (\tau_1, c_1, v_1), (\tau_2, c_2, v_2), \ldots$$

of triples, each consisting of a time stamp τ_i, a program counter $c_i = (m_i, u_i)$, and a port valuation $v_i \in \mathsf{Vals}[\mathsf{Ports}]$, such that conditions (1) and (2) below are satisfied. Before we state these conditions, we first define what it means for a task to be *completed*: we call a task t completed at $(\tau_i, c_i, \cdot)$ if the following conditions hold: (1) if $c_i = (m_i, u_i)$, then $(t, \omega, \cdot) \in \mathsf{Invokes}[m_i]$; (2) t is ready at unit u_i in mode m_i; (3) there exists a previous triple (τ_j, c_j, v_j) in the execution, with $j < i$, and $\tau_j = \tau_i - \pi[m_i]/\omega$; (4) if $c_j = (m_j, \cdot)$, then $(t, \cdot, \cdot) \in \mathsf{Invokes}[m_j]$. The two conditions that must be satisfied are:

1. $\tau_0 = 0$, $c_0 = (\mathsf{start}, 0)$, and v_0 is defined as follows:

   (a) If $p \in \mathsf{EntryPorts} \cup \mathsf{OutPorts} \cup \mathsf{PrivPorts}$, then $v_0(p) = \mathsf{init}[p]$.

   (b) Suppose that $p \in \mathsf{InPorts}$. If there is no task invocation $(t, \cdot, \mathsf{src}) \in \mathsf{Invokes}[\mathsf{start}]$ such that $p \in \mathsf{In}[t]$, then $v_0(p) = \mathsf{init}[p]$. Suppose that there is such a task invocation. If $\mathsf{src}(p) \in \mathsf{Type}[p]$, then $v_0(p) = \mathsf{src}(p)$; if $\mathsf{src}(p) \in \mathsf{Ports}$, then $v_0(p) = v_0(\mathsf{src}(p))$.

2. For $i > 0$:

   (a) $\tau_i = \tau_{i-1} + \pi[m_{i-1}]/\omega_{max}[m_{i-1}]$.

   (b) If there is a mode switch $(m', \cdot, \cdot, \cdot) \in \mathsf{Switches}[m_{i-1}]$ which is enabled at unit $u_{i-1} + 1$ in mode m_{i-1} with respect to the valuation v_{i-1}, then $m_i = m'$. Otherwise, $m_i = m_{i-1}$.

   (c) To define u_i, we first introduce some intermediate definitions for expository clarity. Let u_{min} equal $(u_{i-1} + 1) \bmod (\omega_{max}[m_{i-1}]/\omega_{min}[m_{i-1}])$. Intuitively, u_{min} represents the position in m_{i-1} in terms of the slowest task $t_{min}[m]$. Next, let u_{new} equal $u_{min} \cdot (\omega_{max}[m_i]/\omega_i) \cdot (\omega_{min}[m_{i-1}]/\omega_{max}[m_{i-1}])$, where ω_i is the frequency of $t_{min}[m_{i-1}]$ in m_i. The quantity u_{new} is simply a scaling of u_{min} that accounts for changes in frequency between m_{i-1} and m_i. Finally, define u_i to be $u_{new} + (\omega_i - 1) \cdot \omega_{max}[m_i]/\omega_i$.

   (d) v_i is defined as follows:

      i. Suppose that $p \in \mathsf{EntryPorts}$. If there is no mode switch $(\cdot, \cdot, \cdot, \mathsf{src}) \in \mathsf{Switches}[m_{i-1}]$ which is enabled at unit $u_{i-1} + 1$ in mode m_{i-1} with respect to the valuation v_{i-1}, then $v_i(p) = v_{i-1}(p)$. Suppose that there is such a mode switch. If $\mathsf{src}(p) \in \mathsf{Type}[p]$, then $v_i(p) = \mathsf{src}(p)$; if $\mathsf{src}(p) \in \mathsf{Ports}$, then $v_i(p) = v_i(\mathsf{src}(p))$.

      ii. Suppose that $p \in \mathsf{InPorts}$. If there is no task invocation $(t, \cdot, \mathsf{src}) \in \mathsf{Invokes}[m_i]$ which is ready at unit u_i in mode m_i such that $p \in \mathsf{In}[t]$, then $v_i(p) = v_{i-1}(p)$. Suppose that there is such a task invocation. If $\mathsf{src}(p) \in \mathsf{Type}[p]$, then $v_i(p) = \mathsf{src}(p)$; if $\mathsf{src}(p) \in \mathsf{Ports}$, then $v_i(p) = v_i(\mathsf{src}(p))$.

      iii. Suppose that $p \in \mathsf{OutPorts} \cup \mathsf{PrivPorts}$. If there is no task invocation $(t, \cdot, \cdot) \in \mathsf{Invokes}[m_i]$ which is completed at unit u_i in mode m_i such that $p \in \mathsf{In}[t]$, then $v_i(p) = v_{i-1}(p)$. If there is such a task invocation, then $v_i(p) = \mathsf{f}[t](v_{i-1}[\mathsf{In}[t] \cup \mathsf{Priv}[t]])(p)$.

Note that for a given sequence of valuations for the environment ports, a Giotto program has exactly one execution.

6 Example: Synchronized robots

As an example of a distributed, heterogeneous real-time platform consider a set of n logically identical robots with different hardware and operating systems. Each robot has a CPU, two motors, and a touch sensor. The motors drive wheels and allow the robot to move forward, backward, and sideward. The touch sensor is connected to a bumper. The n robots share a broadcast communication medium. Figure 11 shows the behavior of the n robot system, where a circle depicts the state of a robot and an arc represents a state transition. Note that here state is a behavioral concept rather than, say, a Giotto mode. A robot which is either in the lead or evade state is called a leader. A robot which is either in the follow or stop state is called a follower. We will ensure that at all times there is a single leader and $n - 1$ followers.

Upon initialization the leader is in the lead state and determines the movements taken by all n robots. The $n - 1$ followers are in the follow state and listen to the commands of the leader. All

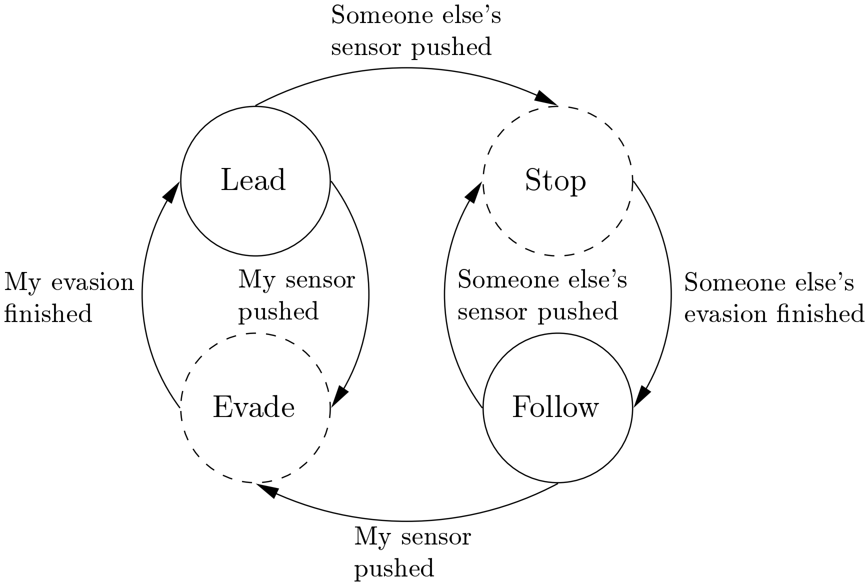

Figure 11: Robot behavior

n robots move in the same way. Now, there are two possible scenarios. Either the leader's bumper or the bumper of one of the followers is pushed. For simplicity, let us assume that, due to some arbitration scheme, no more than a single bumper is pushed at the same time. Suppose that the leader's bumper is pushed. Then the leader goes into the evade state while the $n-1$ followers go into the stop state. A robot in the evade state performs an evasion procedure for a finite amount of time, whereas a robot in the stop mode simply stops. When the leader is finished with the evasion procedure it goes back into the lead state, and the $n-1$ followers return into the follow state. Suppose now that the bumper of one of the followers is pushed. Then this robot goes into the evade state while all other robots, including the leader, go into the stop state. Note that pushing a bumper of a follower effectively makes this robot the new leader. This concludes the behavioral model of the n robot system.

Figure 12 and 13 show a Giotto program that implements a two-robot system. Figure 12 shows the port declarations. The environment port `sensorX` is true whenever the bumper of robot `X` is pushed. The ports `motorLX` and `motorRX` are connected to the left and right motor of robot `X`, respectively.

Figure 13 shows the mode declarations for four modes, where the `leadFollow1` mode is the start mode. Recall that each Giotto mode describes the behavior of the whole system. Since the system consisting of all n robots is either in the lead-and-follow state or in the evade-and-stop state, we use a `leadFollowX` mode and a `evadeStopX` mode for each leader `X`. In general, for n robots we have get $2n$ modes. All modes run at a period of 500 ms with an entry frequency of one. Consider the `leadFollow1` mode, in which robot 1 is the leader. The `command1` task runs once per round and computes a command stored in its output port `com`. There are two more tasks, `motorCtr1` and `motorCtr2`, each running at a frequency of five (i.e., every 100 ms), which control the motors of both robots according to the command in `com`. The higher frequency of these tasks allows for smoother control of the motors. The state of the touch sensors is checked once every round in the three exit conditions. If the bumper of robot 1 is pushed, we switch to the `evadeStop1` mode, in which robot one performs an evasion procedure and robot 2 stops. Similarly, if the bumper of robot 2 is pushed, we switch to the `evadeStop2` mode. Whenever both bumpers are pushed at the same

```
const STOP = 0;

// command
int com = STOP;

// mode finished
bool fin = TRUE;

// TRUE means pushed
bool sensor1; // robot 1 touch sensor
bool sensor2; // robot 2 touch sensor

// 0 means stop
int motorL1 = 0; // robot 1 left motor
int motorR1 = 0; // robot 1 right motor
int motorL2 = 0; // robot 2 left motor
int motorR2 = 0; // robot 2 right motor
```

Figure 12: Port declarations

time, we also switch to the `evadeStop2` mode. In the `evadeStop1` mode, the `evade1` task computes once per round the next evasion step, stored in `com`, and whether the current mode is finished or not, stored in the output port `fin`. Note that upon entry to the `evadeStop1` mode, `fin` always contains true. However, its value will be checked by the exit condition no earlier than at the end of the first round after `evade1` updates `fin`. There is also a task `motorCtr1`, running five times per round, which controls the motors of robot 1 according to the evasion steps in `com`. The third task `motorCtr2` always reads the constant `STOP`, which causes robot 2 to stop during the evasion maneuver. Whenever `fin` contains true, we switch to the `leadFollow1` mode. The implementation of the two modes `leadFollow2` and `evadeStop2`, in which robot 2 is the leader, works similarly as described above.

A reasonable implementation of the two-robot Giotto program may assign the tasks `commandX`, `evadeX`, and `motorCtrX` on robot `X`, respectively, with task `motorCtrX` having the highest priority because of its shortest deadline. Then, the values in `com`, `fin`, and `sensorX` have to be communicated between the two robots.

We have implemented this example with five robots on a heterogenous platform consisting of three Lego Mindstorms robots and two Intel x86 robots. The Lego robots run an implementation of Giotto written in the programming language NQC [Bau99], and communicate via infrared transceivers. The x86 robots run Giotto on VxWorks, and communicate via wireless Ethernet. In addition, there is a sixth CPU that acts as a bridge, forwarding packets between the infrared and wireless communications media. Video clips of the robots in action can be seen at

`www.eecs.berkeley.edu/~fresco/giotto`.

This example illustrates that Giotto provides a truly platform-independent abstract layer for real-time programming.

The robot example makes use of a set of libraries which provide run-time support for Giotto programs. These libraries facilitate the implementation of a distributed time-triggered system using commercial off-the-shelf technology. Our libraries currently target Intel 80486 processors and Ethernet because of the easy availability of these hardware elements.[1] We use UDP/IP as our

[1] In general, Ethernet is not well-suited to hard real-time applications, because collisions and collision recovery

```
start leadFollow1() {
  mode leadFollow1() period 500ms entryfreq 1 {
    taskfreq 1 do int com = command1();
    taskfreq 5 do (int motorL1, int motorR1) = motorCtr1(com);
    taskfreq 5 do (int motorL2, int motorR2) = motorCtr2(com);

    exitfreq 1 if (sensor1 && not(sensor2)) then evadeStop1();
    exitfreq 1 if (sensor2 && not(sensor1)) then evadeStop2();
    exitfreq 1 if (sensor1 && sensor2) then evadeStop2();
  }
  mode evadeStop1() period 500ms entryfreq 1 {
    taskfreq 1 do (int com, bool fin) = evade1();
    taskfreq 5 do (int motorL1, int motorR1) = motorCtr1(com);
    taskfreq 1 do (int motorL2, int motorR2) = motorCtr2(STOP);

    exitfreq 1 if (fin) then leadFollow1();
  }
  mode leadFollow2() period 500ms entryfreq 1 {
    taskfreq 1 do int com = command2();
    ...
  }
  mode evadeStop2() period 500ms entryfreq 1 {
    taskfreq 1 do (int com, bool fin) = evade2();
    taskfreq 1 do (int motorL1, int motorR1) = motorCtr1(STOP);
    taskfreq 5 do (int motorL2, int motorR2) = motorCtr2(com);

    exitfreq 1 if (fin) then leadFollow2();
  }
}
```

Figure 13: Two-robot Giotto program

communications protocol, and target the VxWorks operating system. We wish to emphasize that our code may be ported to any processor with a single timer, and real-time operating system able to use this timer, and to virtually any digital communications medium. In fact, since our timing and synchronization requirements are quite simple, we could equally well target processors without operating systems.

7 Hierarchical Giotto

Consider a task declaration $(t, \mathsf{In}, \mathsf{Out}, \mathsf{Priv}, \mathsf{f})$. The task function $\mathsf{f}\colon \mathsf{Vals}[\mathsf{In} \cup \mathsf{Priv}] \to \mathsf{Vals}[\mathsf{Out} \cup \mathsf{Priv}]$ can be implemented in any programming language, such as C. If we implement f itself as a Giotto program, then we obtain hierarchical Giotto. For this purpose, we need to consider *finite* executions of a Giotto program. In a finite execution, we run a Giotto program with start mode start for time $\pi[\mathsf{start}]$. Consequently, if no mode switch occurs, then exactly one round of the start mode is executed. If a mode switch occurs before the first round of start is completed, then some task must be invoked with task frequency one in mode start, and this task needs to continue in the target

can interfere with quality of service guarantees. However, we use a TDMA strategy to prevent collisions. Using this strategy, we have found that collisions do not present a significant problem.

mode. The finite execution ends with the completion of this task, possibly after several mode switches.

Consider a mode m and a task invocation $(t, \cdot, \omega) \in \mathsf{Invokes}[m]$. The task function $\mathsf{f}[t]$ is *defined* by a Giotto program g if the following three conditions are satisfied:

1. The start mode of g has the period $\pi[m]/\omega$. We write $\pi[g]$ for $\pi[m]/\omega$.

2. All input ports in $\mathsf{In}[t]$ are entry ports of the start mode of g, and all output ports in $\mathsf{Out}[t]$ are output ports of g. The set $\mathsf{Priv}[t]$ of private ports contains the entry ports of g which are not in $\mathsf{In}[t]$, the input ports of g, the output ports of g which are not in $\mathsf{Out}[t]$, and the private ports of g.

3. The program g has no environment ports. This is required, because the results of g at time $\pi[g]$ must be deterministic. In a more general version of hierarchical Giotto, we replace task functions by binary task relations, which can be implemented using Giotto programs with environment variables. This is delayed to the full paper.

If the three conditions are met, then given a valuation $v \in \mathsf{Vals}[\mathsf{In}[t] \cup \mathsf{Priv}[t]]$, we write $g(v)$ for the Giotto program which results from g by changing, for each port $p \in \mathsf{In}[t] \cup \mathsf{Priv}[t]$, the initial value of p to $v(p)$. Let $(\tau_0, c_0, v_0), \ldots, (\tau_n, c_n, v_n)$ be the (unique) finite prefix of the (unique) execution of $g(v)$ such that $\tau_n = \pi[g]$. Then, g defines the task function $\mathsf{f}[t]$ which maps each valuation $v \in \mathsf{Vals}[\mathsf{In}[t] \cup \mathsf{Priv}[t]]$ to the valuation $v_n[\mathsf{Out}[t] \cup \mathsf{Priv}[t]]$.

8 Annotating Giotto with platform constraints

A Giotto program can in principle be run on a single sufficiently fast CPU, independent of the number of modes and tasks, and of the worst-case execution time of 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. Lastly, in order to guarantee 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 annotate Giotto programs with platform constraints.

For now, we consider as target platform the most common sort of real-time operating system (static priority preemptive) and a common, readily available type of network (Ethernet). (Other operating systems —e.g., the TTA [Kop97]— and networks —e.g., the Controller Area Network [Int93]— may permit more straightforward implementations, but these platforms are less common.) Formally, a *hardware configuration* consists of a set of *hosts* and a set of *nets*. A host is a CPU which can execute Giotto tasks. A net connects two or more hosts and can transport values. The passing of a value from an output port of one task (or from an entry port of a mode) to an input port of another task (or to an entry port of a mode) is called a *connection*. An *execution priority* is a positive integer. A *time slot* is an open interval $(\tau_1, \tau_2) \subseteq \mathbb{R}$ of real time. In order to implement a Giotto program on our target platform, task invocations need to be mapped to priorities, and connections need to be mapped to time slots. Specifically, a Giotto compiler takes as input a Giotto program as well as a hardware configuration and produces, before generating code:

- An assignment of task invocations to hosts and execution priorities. If two task invocations are mapped to the same host, then the invocation with the higher priority will preempt the invocation with the lower priority.

- An assignment of connections to nets and time slots. If a connection is assigned a time slot (τ_1, τ_2), then the transmission of this connection may only occur in the time interval $(\tau_1 + \tau_r, \tau_2 + \tau_r)$, where τ_r is time at which the current round of execution began. A Giotto compiler guarantees the absence of transmission collisions by ensuring that two hosts do not transmit data on the same network at the same time.

If no assignments that preserve the Giotto semantics exist, then the compiler fails.

It may be thought that the frequencies of task invocations are enough to determine priorities. Indeed, using standard rate-monotonic techniques [Kle93], it would seem that, on a single host, task invocations with higher frequencies simply need to have higher priorities than task invocations with lower frequencies. However, it is important to design the task schedule together with the communication schedule. A low-frequency task may produce an output that has to travel a long way, and thus it may have to be transmitted earlier than the outputs of high-frequency tasks which have little communication delays.

We permit the programmer to aid the compiler by giving directives, which are partial assignments of tasks to hosts and execution priorities, and partial assignments of connections to nets and time slots. Directives are given in the form of program annotations. They are useful for simplifying the compilation task but also if external considerations need to be taken into account; for example, a high-security task may have to be assigned to a particular host.

9 Related work

Giotto is inspired by the time-triggered architecture (TTA) [Kop97] on one hand, and by synchronous programming languages [Hal93, Ber00] on the other hand.

The time-triggered, rather than event-triggered, paradigm to meet hard real-time constraints in safety-critical distributed settings is realized in the TTA. Also the concept of mode and mode switching is addressed there. However, while TTA encompasses a hardware architecture and a communication protocol, Giotto provides a hardware-independent and protocol-independent abstract layer for programming hard real-time constraints using time-triggered task invocation and mode switching. While Giotto can be implemented on any platform that provides sufficiently accurate clock primitives (or supports a clock synchronization algorithm), it is of course most straight-forward to compile a Giotto program on the TTA.

The goal of Giotto —to provide a platform-independent programming abstraction for real-time systems— is shared also by synchronous programming languages. There are other similarities, such as the concept of multiform time and the assumption that value passing is instantaneous. The spirit of Giotto, however, is fundamentally different from the spirit of synchronous languages in at least two respects. First, while synchronous languages are designed around zero-delay value propagation, Giotto is based on unit-delay value propagation, because in Giotto, computation does take time. This decision shifts the focus in essential ways; for example, for analysis and compilation, the burden for the well-definedness of values is shifted from fixed-point considerations to constraints about platform resources and performance (without platform constraints, in Giotto all values are trivially well-defined). Second, Giotto is built around the concept of mode and mode switch, which is fundamental for handling multi-modal control, fault tolerance, uncertain environments, and resource sharing. The mode concept is not a primitive of synchronous languages: while synchronous programs result from the concurrent composition of sequential (instantaneous) functions, Giotto programs result from the sequential composition (namely, mode switching) of concurrent (non-instantaneous) tasks. In hierarchical Giotto, a task itself can be implemented by a Giotto program, which leads to the arbitrary nesting of concurrent and sequential composition.

For ongoing work on Giotto, see `www.eecs.berkeley.edu/~fresco/giotto`.

Acknowledgments. We thank Rupak Majumdar for implementing a prototype Giotto compiler for Lego Mindstorms robots. We thank Dmitry Derevyanko and Winthrop Williams for building our Intel x86 robots. We thank Edward Lee and Xiaojun Liu for inspiring hierarchical Giotto and for help with a Ptolemy II [DGH$^+$99] implementation of Giotto.

References

[Bau99] D. Baum. *Dave Baum's Definitive Guide to Lego Mindstorms.* A Press, 1999.

[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.* MIT Press, 2000.

[BGP00] J. Berwanger, R. Grießbach, and M. Peller. Byteflight: A new high performance data bus system for safety-related applications. *Automotive Electronics (Sonderausgabe von ATZ und MTZ)*, pages 60–67, January 2000. In German. In English at `www.byteflight.com`.

[Col99] R.P.G. Collinson. Fly-by-wire flight control. *Computing & Control Engineering*, 10(4):141–152, August 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, July 1999.

[FMD+00a] Thomas Führer, Dr. Bernd Müller, Werner Dieterle, Florian Hartwich, and Robert Hugel (Robert Bosch). CAN network with time-triggered communication. In *Proceedings of the 7th International CAN Conference 2000*, 2000.

[FMD+00b] Thomas Führer, Dr. Bernd Müller, Werner Dieterle, Florian Hartwich, and Robert Hugel (Robert Bosch). Time-triggered communication on CAN (time-triggered CAN — TTCAN). In *Proceedings of the 7th International CAN Conference 2000*, 2000.

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

[Int93] International Organization for Standardization. *Road Vehicles — Interchange of Digital Information — Controller Area Network (CAN) for High-speed Communication (ISO 11898)*, 1993.

[Kle93] M. Klein et al. *A Practitioner's Handbook for Real-time Analysis: Guide to Rate Monotonic Analysis for Real-time Systems.* Kluwer, 1993.

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

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