

# Giotto: A Time-triggered Language for Embedded Programming

Thomas A. Henzinger Benjamin Horowitz Christoph M. Kirsch

Electrical Engineering and Computer Sciences  
University of California, Berkeley

**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, actuator updates, 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.

**Keywords**—Programming languages, real-time systems, control systems, embedded software.

## I. INTRODUCTION

Giotto provides a programming abstraction for hard real-time applications that exhibit time-periodic and multimodal 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,

This research was supported in part by the AFOSR MURI grant F49620-00-1-0327, the DARPA SEC grant F33615-C-98-3614, the MARCO GSRC grant 98-DT-660, and the NSF grant CCR-0208875.

A preliminary version of this paper appeared in the *Proceedings of the International Workshop on Embedded Software*, vol. 2211 of Lecture Notes in Computer Science. Springer-Verlag, 2001, pp. 166–184.

and achieving the desired degree of fault tolerance through replication and error correction. While limited automation for these activities is available in the form of code-generation tools, the software engineer has final authority over putting the implementation together through an often iterative process of code integration, testing, and optimization.

Giotto provides an intermediate level of abstraction, which (i) permits the software engineer to communicate more effectively with the control engineer, and (ii) keeps the implementation and its properties more closely aligned with the mathematical model of the control design. 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. 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. 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. Giotto compilation guarantees the preservation of functionality and timing, and thus removes the need for a tedious and error-prone iteration of code evaluation and optimization.

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 [1], [2], which consists of three types of interconnected components (see Figure 2): sensors, CPUs for computing control laws, and actuators. The sensors include an inertial measurement unit (IMU), for measuring linear acceleration and angular velocity; a global positioning system (GPS), for measuring position; an air data measurement system, for measuring such quantities as air pressure; and

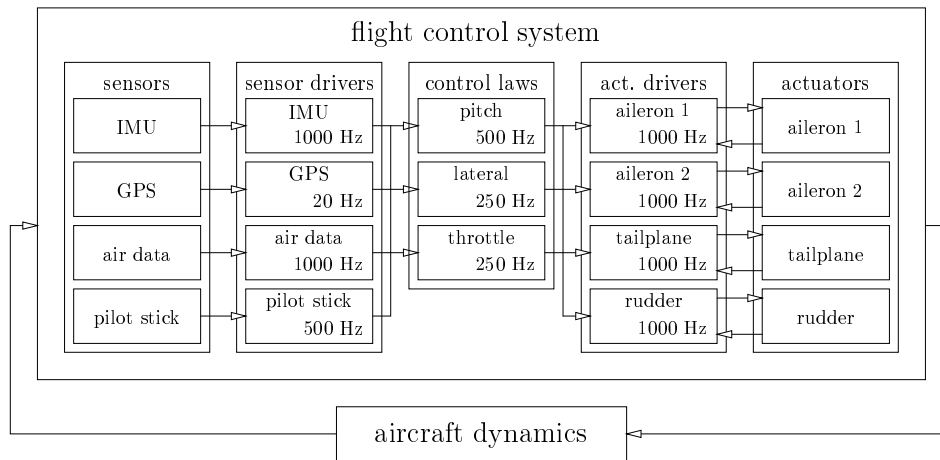


Fig. 2. A fly-by-wire flight control system.

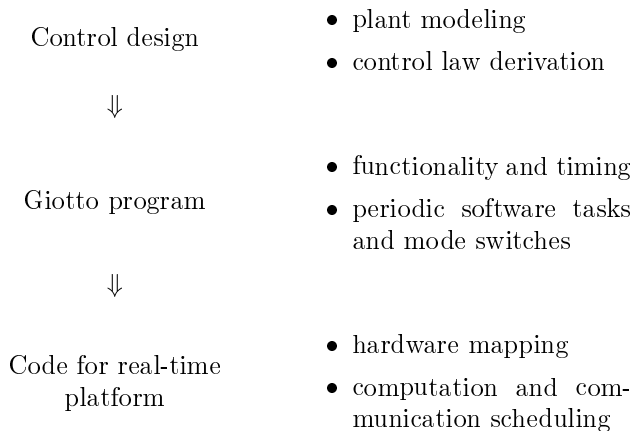


Fig. 1. Giotto-based control-systems development.

the pilot’s controls, such as the pilot’s stick. Each sensor has its own timing properties: the IMU, 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—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 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 take-off, 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 take-off 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 application-level computation that consumes a non-negligible amount of CPU time, a driver is bounded code that can be executed essentially instantaneously on the system level, with environment interrupts disabled (more precisely, drivers satisfy the *synchrony assumption* [3], that they can be executed before the environment state changes<sup>1</sup>). In this way, the Giotto abstraction integrates scheduled computation (tasks) and synchronous communication (drivers). 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;<sup>2</sup> 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 a mathematical (e.g., Matlab) model of the control design.

A Giotto program does not specify where, how, and when

<sup>1</sup>Since drivers cannot depend on each other, no issues of fixed-point semantics arise.

<sup>2</sup>While any choice of time unit is possible, we use milliseconds throughout the paper.

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 real-time operating systems) as well as on truly time-triggered platforms (such as the time-triggered architecture [4]). All the Giotto compiler needs to ensure is that the semantics of the program—i.e., functionality and timing—is preserved. To this end, the compiler needs to solve a possibly distributed scheduling problem. This can be difficult, and to make the job of the compiler easier, a Giotto program can be annotated with compiler directives in the form of *platform constraints*. A platform constraint may map a particular task to a particular CPU, assign a particular priority to a particular task, or schedule a particular communication event between tasks in a particular time slot. Such annotations, however, in no way modify the functionality and timing of a Giotto program; they simply aid the compiler in realizing the semantics of the program.

**Outline of the paper.** We first give an informal introduction to Giotto in Section II, followed by a formal definition of the language in Section III. In Section IV, we define an abstract version of the scheduling problem that needs to be solved by the Giotto compiler, and we illustrate how a program can be annotated to guide distributed code generation. In Section V, we give pointers to current Giotto implementations and relate Giotto to the literature.

## II. 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. Task ports can also be used to transfer data from one mode to the next: task ports can be designated as mode ports of a given mode, and 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  $\text{In}$  of two input ports and a set  $\text{Out}$  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 the tasks 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  $\text{Priv}$ . 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

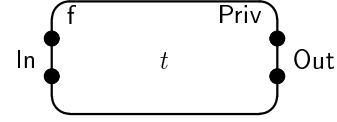


Fig. 3. A task  $t$ .

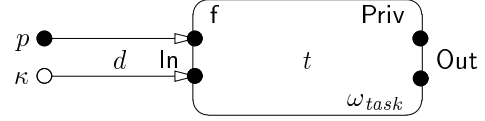


Fig. 4. An invocation of task  $t$ .

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 available CPU.

**Task 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. The task invocation has a frequency  $\omega_{task}$  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  $\omega_{task}$ . The task invocation specifies a driver  $d$  which provides values for the input ports  $\text{In}$ . 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  $\kappa$ . 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  $\tau_{start}$  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 driver  $d$ —is performed in *logically zero time*. In other words, a Giotto driver is an atomic unit of computation that cannot be interrupted. The synchronous communication phase is followed by a scheduled computation phase. The Giotto semantics prescribes that at time  $\tau_{stop}$  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  $\tau_{start}$ . The length of the interval between  $\tau_{start}$  and  $\tau_{stop}$  is determined by the frequency  $\omega_{task}$ . We say that the task  $t$  is *logically running* from time  $\tau_{start}$  to time  $\tau_{stop}$ . The Giotto logical abstraction does not specify when, where, and how the actual computation of  $f$  is physically performed between  $\tau_{start}$  and  $\tau_{stop}$ . However, the time at which the task output ports are updated is determined, and therefore, for any given real-time trace of

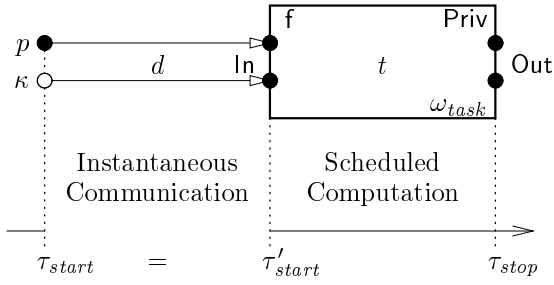


Fig. 5. The time line for an invocation of task  $t$ .

sensor values, all values that are communicated between tasks and to the actuator ports are determined [5]. Instantaneous communication and time-deterministic computation are the two essential ingredients of the Giotto logical abstraction. A compiler must be faithful to this abstraction; for example, task inputs may be loaded after time  $\tau_{start}$ , and the execution of  $f$  may be preempted by other tasks, as long as at time  $\tau_{stop}$  the values of the task output ports are those specified by the Giotto semantics.

**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. Possible transitions from a mode to other modes are specified by mode switches. 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  $m$  which contains invocations of two tasks,  $t_1$  and  $t_2$ . The period  $\pi$  of  $m$  is 10 ms; that 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  $m$ . In addition to  $o_1$ , all output ports of tasks invoked in the mode —  $o_2$ ,  $o_3$ ,  $o_4$ , and  $o_5$  — are, by default, also mode ports; they must be initialized upon entering mode  $m$ . The mode ports 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 invocation of task  $t_1$  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 task  $t_2$  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 the output port  $o_5$  of  $t_2$  to  $i_5$ . The mode  $m$  has one actuator update, which is a driver  $d_3$  that copies the value of the output port  $o_2$  of  $t_1$  to the actuator port  $a$  with the actuator frequency  $\omega_{act} = 1$ ; that is, once every 10 ms.

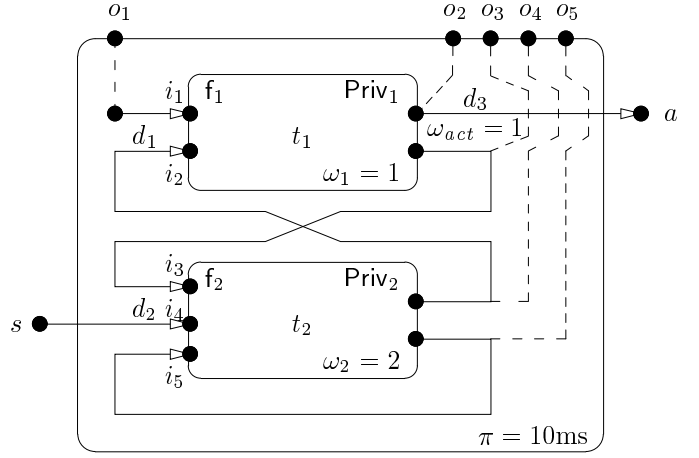


Fig. 6. A mode  $m$ .

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  $\tau_0$  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. The Giotto semantics specifies only that after 5 ms, at time instant  $\tau_1$ , the results of the scheduled computation of  $f_2$  are made available at the output ports of  $t_2$ . The second invocation of  $t_2$  begins with another execution of driver  $d_2$ , still at time  $\tau_1$ , 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 whether or not  $f_1$  finishes its actual computation before  $\tau_1$ . According to the Giotto semantics, the output values of the invocation of  $t_1$  must not be available before  $\tau_2$ . Any implementation 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  $\tau_2$ , 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  $\omega_3$  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 the output port  $o_6$  of  $t_3$  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

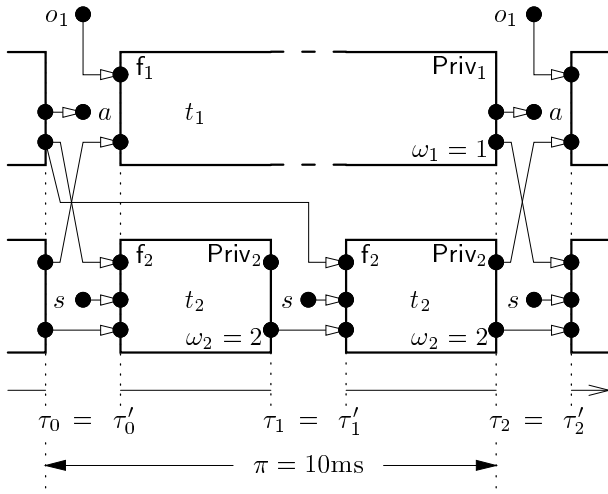


Fig. 7. The time line for a round of mode  $m$ .

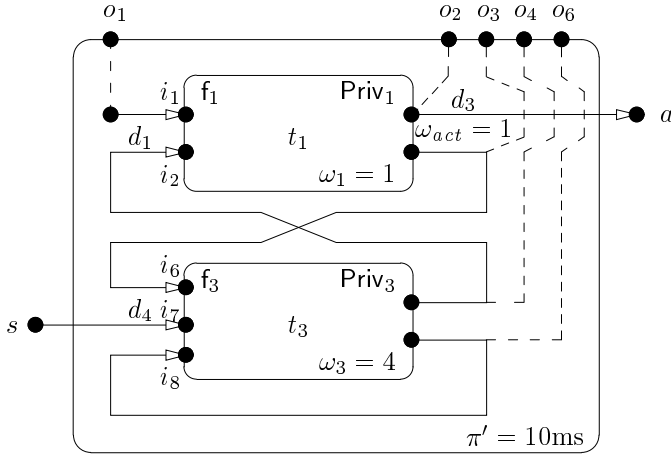


Fig. 8. A mode  $m'$ .

invoked 4 times per round, every 2.5 ms; and the actuator  $a$  is updated once per round, every 10 ms.

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  $\eta$  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 the *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  $\kappa$  into  $o_1$ , the value of the mode port  $o_5$  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). Like all drivers, mode switches are performed in logically zero

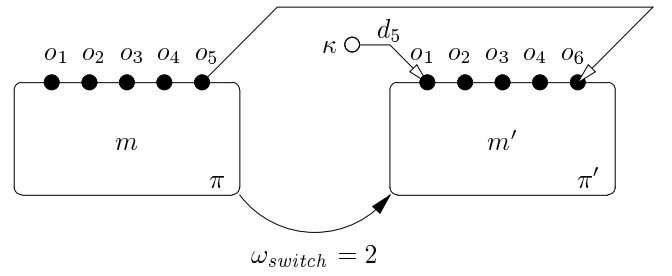


Fig. 9. A mode switch  $\eta$  from mode  $m$  to mode  $m'$ .

time.

Figure 10 shows the time line for the mode switch  $\eta$  performed at time  $\tau_1$ . The program is in mode  $m$  until  $\tau_1$  and then enters mode  $m'$ . Note that until time  $\tau_1$  the time line corresponds to the time line shown in Figure 7. At time  $\tau_1$ , 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  $\tau_1$ . Note that the output port  $o_6$  of  $t_3$  has just received the value of the output port  $o_5$  from task  $t_2$  by the mode driver  $d_5$ . At time  $\tau_2$ , task  $t_3$  is invoked a second time, and at time  $\tau_3$ , 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  $\kappa$  from the mode port  $o_1$ . In this way, task  $t_1$  can detect that a mode switch occurred.

A mode switch may occur while a task is logically running; in this case we say that the mode switch *logically interrupts* the task invocation. 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 continued in the target mode. In our example, the mode switch  $\eta$  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.

### III. FORMAL DEFINITION OF GIOTTO

#### A. Syntax

Rather than specifying a concrete syntax for Giotto, we formally define the components of a Giotto program in a more abstract way. In practice, Giotto programs can be written in a concrete, C like syntax.

A *Giotto program* consists of the following components:

- 1) A set of *port declarations*. A port declaration  $(p, \text{Type}, \text{init})$  consists of a port name  $p$ , a type  $\text{Type}$ , and an initial value  $\text{init} \in \text{Type}$ . We require that all port names are uniquely declared; that is, if  $(p, \cdot, \cdot)$  and  $(p', \cdot, \cdot)$  are distinct port declarations, then  $p \neq p'$ .



then there exists a task invocation  $(\omega'_{task}, t, \cdot) \in \text{Invokes}[m']$  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 is logically running, then the same task must be present also in the target mode.

### B. Semantics

A *program configuration*  $C = (m, \delta, v, \sigma_{active}, \tau)$  consists of a mode  $m \in \text{Modes}$ , a *mode time*  $\delta \in \mathbb{Q}$ , a valuation  $v \in \text{Vals}[\text{Ports}]$  for all ports, a set  $\sigma_{active} \subseteq \text{Tasks}$  of *active tasks*, and a *time stamp*  $\tau \in \mathbb{Q}$ . The set  $\sigma_{active} \subseteq \text{Tasks}$  contains all tasks that are logically running, whether or not they are physically running by expending CPU time. The number  $\delta \geq 0$  measures the amount of time that has elapsed since the last mode switch, unless some tasks were logically running at the time of the last mode switch, in which case  $\delta$  “dates back” the mode switch to the closest time instant before the mode switch when the current mode could have started from its beginning with all its tasks. For a program configuration  $C$  and a set  $P \subseteq \text{Ports}$ , we write  $C[P]$  for the valuation in  $\text{Vals}[P]$  that agrees with  $C$  on the values of all ports in  $P$ .

The *mode frequencies* of a mode  $m \in \text{Modes}$  include (i) the task frequencies  $\omega_{task}$  for all task invocations  $(\omega_{task}, \cdot, \cdot) \in \text{Invokes}[m]$ , (ii) the actuator frequencies  $\omega_{act}$  for all actuator updates  $(\omega_{act}, \cdot) \in \text{Updates}[m]$ , and (iii) the mode-switch frequencies  $\omega_{switch}$  for all mode switches  $(\omega_{switch}, \cdot, \cdot) \in \text{Switches}[m]$ . Let  $\omega_{max}[m]$  be the least common multiple of the mode frequencies of  $m$ . During an execution, as long as the program is in mode  $m$ , the program configuration is updated every  $\pi[m]/\omega_{max}[m]$  time units. Each update results from a sequence of five types of events: first, some tasks are completed (i.e., removed from the active set); second, some actuators are updated; third, some sensors are read; fourth, a mode switch may occur; fifth, some new tasks are activated.

Let us be more precise. Consider a program configuration  $C = (m, \delta, v, \sigma_{active}, \tau)$ . We need the following auxiliary definitions:

- A task invocation  $(\omega_{task}, t, \cdot) \in \text{Invokes}[m]$  is *completed* at configuration  $C$  if  $t \in \sigma_{active}$ , and  $\delta$  is an integer multiple of  $\pi[m]/\omega_{task}$ .
- An actuator update  $(\omega_{act}, d) \in \text{Updates}[m]$  is *evaluated* at configuration  $C$  if  $\delta$  is an integer multiple of  $\pi[m]/\omega_{act}$ .
- A mode switch  $(\omega_{switch}, \cdot, d) \in \text{Switches}[m]$  is *evaluated* at configuration  $C$  if  $\delta$  is an integer multiple of  $\pi[m]/\omega_{switch}$ .
- A task invocation  $(\omega_{task}, \cdot, d) \in \text{Invokes}[m]$  is *evaluated* at configuration  $C$  if  $\delta$  is an integer multiple of  $\pi[m]/\omega_{task}$ .

The actuator update  $(\omega_{act}, d)$ , mode switch  $(\omega_{switch}, \cdot, d)$ , or task invocation  $(\omega_{task}, \cdot, d)$  is *enabled* at configuration  $C$  if it is evaluated at  $C$  and  $g[d](v) = \text{true}$ .

The program configuration  $C_{succ}$  is a *successor configuration* of  $C$  if  $C_{succ}$  results from  $C$  by the following nine steps, called *Giotto micro steps*. These are the steps a Giotto program performs whenever it is invoked, initially with  $\delta = 0$ ,  $\sigma_{active} = \emptyset$ , and  $\tau = 0$ :

- 1) **[Update task output and private ports]** Let  $\sigma_{completed}$  be the set of tasks  $t$  such that a task invocation of the form  $(\cdot, t, \cdot) \in \text{Invokes}[m]$  is completed at configuration  $C$ . Consider a port  $p \in \text{OutPorts} \cup \text{PrivPorts}$ . If  $p \in \text{Out}[t] \cup \text{Priv}[t]$  for some task  $t \in \sigma_{completed}$ , then define  $v_{task}(p) = f[t](C[\text{In}[t] \cup \text{Priv}[t]])(p)$ ; 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  $\text{OutPorts} \cup \text{PrivPorts}$ , and otherwise agrees with  $C$ .
- 2) **[Update actuator ports]** Consider a port  $p \in \text{ActPorts}$ . If  $p \in \text{Dst}[d]$  for some actuator update  $(\cdot, d) \in \text{Updates}[m]$  that is enabled at configuration  $C_{task}$ , then define  $v_{act}(p) = h[d](C_{task}[\text{Src}[d]])(p)$ ; 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  $\text{ActPorts}$ , and otherwise agrees with  $C_{task}$ .
- 3) **[Update sensor ports]** Consider a port  $p \in \text{SensePorts}$ . Let  $v_{sense}(p)$  be any value in  $\text{Type}[p]$ ; 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  $\text{SensePorts}$ , and otherwise agrees with  $C_{act}$ .
- 4) **[Update mode]** If a mode switch  $(\cdot, m_{target}, \cdot) \in \text{Switches}[m]$  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) **[Update mode ports]** Consider a port  $p \in \text{OutPorts}$ . If  $p \in \text{Dst}[d]$  for some mode switch  $(\cdot, \cdot, d) \in \text{Switches}[m]$  that is enabled at configuration  $C_{sense}$ , then define  $v_{mode}(p) = h[d](C_{target}[\text{Src}[d]])(p)$ ; otherwise, define  $v_{mode}(p) = C_{target}[\text{OutPorts}](p)$ . 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  $\text{OutPorts}$ , and otherwise agrees with  $C_{target}$ .
- 6) **[Update mode time]** If no mode switch in  $\text{Switches}[m]$  is enabled at configuration  $C_{sense}$ , then define  $\delta' = \delta$ . 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  $\delta' = 0$ . Otherwise, let  $\gamma$  be the least common multiple of the set  $\{\pi[m]/\omega_{task} \mid (\omega_{task}, t, \cdot) \in \text{Invokes}[m] \text{ for some } t \in \sigma_{running}\}$  of task periods for running tasks; then  $\gamma$  is the time it takes during a round of mode  $m$  to complete all running tasks simultaneously. Let  $\varepsilon$  be the least integer multiple of  $\gamma$  such that  $\varepsilon \geq \delta$ ; then  $\varepsilon - \delta$  is the time

```

sensor
  port s1 type ℝ
  port s2 type {0, 1}
actuator
  port a type ℝ init 0
input
  port i1 type ℝ
  port i2 type ℝ
  port i3 type ℝ
output
  port o1 type ℝ init 0
  port o2 type ℝ init 0
private
  port p1 type ℝ init 0
  port p2 type ℝ init 0
  port p3 type ℝ init 0

task t1 input i1 output o1 private p1 function f1
task t2 input i2 output o2 private p2 function f2
task t3 input i3 output o2 private p3 function f3

driver d1
  source o2 guard g1 destination i1 function h1
driver d2
  source s1 guard g1 destination i2 function h2
driver d3
  source s1 guard g1 destination i3 function h3
driver d4
  source o1 guard g1 destination a function h4
driver d5
  source s2 guard g5 destination o1, o2 function h5

mode m1 period 6 ports o1, o2
  frequency 1 invoke t1 driver d1
  frequency 2 invoke t2 driver d2
  frequency 1 update d4
  frequency 2 switch m2 driver d5
mode m2 period 12 ports o1, o2
  frequency 2 invoke t1 driver d1
  frequency 3 invoke t3 driver d3
  frequency 2 update d4
  frequency 3 switch m1 driver d5

start m1

```

Fig. 11. The abstract syntax of a Giotto program with two modes.

until the next simultaneous completion point. Define  $\delta' = \pi[m'] - (\varepsilon - \delta)$ . Thus a mode switch always jumps as close as possible to the end of a round of the target mode. Let  $C_{local}$  be the configuration with mode time  $\delta'$  that otherwise agrees with  $C_{mode}$ .

- 7) **[Update task input ports]** Consider a port  $p \in \text{InPorts}$ . If  $p \in \text{Dst}[d]$  for some task invocation  $(\cdot, \cdot, d) \in \text{Invokes}[m']$  that is enabled at configuration  $C_{local}$ , then define  $v_{input}(p) = h[d](C_{local}[\text{Src}[d]])(p)$ ; 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  $\text{InPorts}$ , and otherwise agrees with  $C_{local}$ .
- 8) **[Update active tasks]** Let  $\sigma_{enabled}$  be the set of tasks  $t$  such that a task invocation of the form  $(\cdot, \cdot, \cdot) \in \text{Invokes}[m']$  is enabled at configuration  $C_{local}$ . 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  $\sigma'_{active}$  of active tasks that otherwise agrees with  $C_{input}$ .
- 9) **[Advance time]** Let  $\delta_{succ}$  be the least integer multiple of  $\pi[m']/\omega_{max}[m']$  such that  $\delta_{succ} > \delta'$ ; this is the time of the next event (task invocation, actuator update, or mode switch) in mode  $m'$ . The next time instant at which the Giotto program is invoked is  $\delta_{succ} - \delta'$  time units in the future; an implementation may use a timer interrupt for this. Let  $\tau_{succ} = \tau + \delta_{succ} - \delta'$ . Let  $C_{succ}$  be the configuration with mode time  $\delta_{succ}$  and time stamp  $\tau_{succ}$  that otherwise agrees with  $C_{active}$ .

An *execution* of a Giotto program is an infinite sequence  $C_0, C_1, C_2, \dots$  of program configurations  $C_i$  such that (i)  $C_0 = (\text{start}, 0, v, \emptyset, 0)$  with  $v(p) = \text{init}[p]$  for all ports  $p \in \text{Ports}$ , 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 the

start time of the program, but there can never be two mode switches in a row without any time passing.

### C. Example

We use the simple Giotto program from Figure 11 to illustrate Giotto's semantics. This program contains two modes,  $m_1$  and  $m_2$ . Mode  $m_1$  has a period of 6 ms, and invokes two tasks,  $t_1$  and  $t_2$ , with frequencies of 1 and 2, respectively. Mode  $m_2$  has a period of 12 ms, and invokes  $t_1$  and the task  $t_3$ , with frequencies of 2 and 3, respectively. The tasks  $t_2$  and  $t_3$  both read the sensor port  $s_1$  and write to the same output port  $o_2$ . This is possible because  $t_2$  and  $t_3$  are invoked in different modes. The task  $t_1$  reads  $o_2$  and writes to the output port  $o_1$ , which is read by the actuator driver  $d_4$  to write the actuator port  $a$ . In both modes the actuator update occurs every 6 ms. Mode  $m_1$  evaluates a possible mode switch to mode  $m_2$  every 3 ms;  $m_2$  contemplates switching back to  $m_1$  every 4 ms. These mode switches are controlled by the driver  $d_5$ , which reads the sensor port  $s_2$ . A mode change occurs if  $s_2$  contains the value 1. Both mode switches, when enabled, write the ports  $o_1$  and  $o_2$ . We assume that with the exception of  $g_5$ , the guards  $g_1$  of all other drivers are always true. The initial values of the sensor and input ports are omitted from the figure, as they are written before being read.

To illustrate the semantics of this program, consider an execution  $\hat{E} = C_0, C_1, C_2, \dots$  that begins with the following

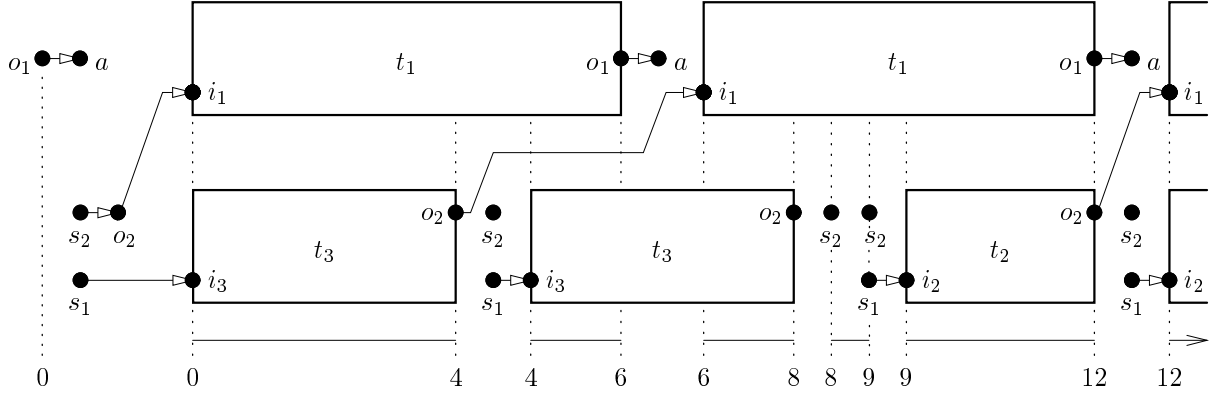


Fig. 12. The time line for an execution of the program from Figure 11.

program configurations:

$$\begin{aligned}
 C_0 &= (m_1, 0, \cdot, \emptyset, 0) \\
 C_1 &= (m_2, 2, \cdot, \{t_1, t_3\}, 2) \\
 C_2 &= (m_2, 4, \cdot, \{t_1, t_3\}, 4) \\
 C_3 &= (m_2, 6, \cdot, \{t_1, t_3\}, 6) \\
 C_4 &= (m_2, 8, \cdot, \{t_1, t_3\}, 8) \\
 C_5 &= (m_1, 3, \cdot, \{t_1\}, 9) \\
 C_6 &= (m_1, 6, \cdot, \{t_1, t_2\}, 12) \\
 C_7 &= (m_1, 9, \cdot, \{t_1, t_2\}, 15)
 \end{aligned}$$

The execution  $\hat{E}$  starts in mode  $m_1$ , but switches immediately to mode  $m_2$ . At configuration  $C_4$ , the execution switches back to mode  $m_1$  (note that a mode switch in a configuration  $C_i$  is reflected only in the successor configuration  $C_{i+1}$ ). The execution remains in mode  $m_1$  until configuration  $C_7$ .

Figure 12 shows an initial segment of the time line for  $\hat{E}$ . At 0 ms, the port  $o_1$  is used to update the actuator port  $a$ . The sensor port  $s_2$  is read by the mode driver  $d_5$ . The guard of  $d_5$  evaluates to true, indicating a mode change, and thus the port  $o_2$  is updated (port  $o_1$  is also updated but not used, and therefore omitted in the figure). Port  $o_2$  provides the input to task  $t_1$ . The sensor  $s_1$  provides the input to task  $t_3$ . At 4 ms, task  $t_3$  completes; the sensor port  $s_2$  is updated, but no mode change occurs; and the sensor  $s_1$  provides input to a new invocation of task  $t_3$ . At 6 ms, task  $t_1$  completes; the actuator port  $a$  is updated using the output of  $t_1$ ; and a new invocation of task  $t_1$  starts. At 8 ms, task  $t_3$  completes; sensor  $s_2$  is updated; and the guard of the mode driver  $d_5$  evaluates to true, indicating a switch to mode  $m_1$  and updating ports  $o_1$  and  $o_2$  (neither port is used, and therefore both are omitted in the figure). At 9 ms, sensor  $s_2$  is updated again, but no mode change occurs; sensor  $s_1$  is updated; and an invocation of task  $t_2$  begins. At 12 ms, both tasks  $t_1$  and  $t_2$  complete;  $a$ ,  $s_2$ , and  $s_1$  are all updated; and new invocations of  $t_1$  and  $t_2$  start. Note that at the time of the mode switch at 8 ms, the mode time of the target mode  $m_1$  is 2 ms, because task  $t_1$  has been logically running for 2 ms. For the duration of 1 ms, task  $t_1$  is the only running task, until at mode time 3 ms (real time 9 ms), an invocation of task  $t_2$  is added. At mode time 6 ms (real time 12 ms), the partial round of mode  $m_1$  is finished,

and a new round begins.

#### IV. PLATFORM CONSTRAINTS FOR GIOTTO

In order to compile a Giotto program, the compiler needs two additional pieces of information: (i) a *platform specification*, which defines the number and topology of hosts (CPUs), and worst-case execution times for all Giotto activities (tasks, drivers, and sensor readings); and (ii) a *jitter tolerance*, which specifies how much the actual timing can deviate from the Giotto semantics. The jitter tolerance is needed because it may be impossible to implement the Giotto semantics exactly. For example, if according to Giotto semantics, several actuators are written at the same point  $\tau$  in time, and there is only one host, then the actual writes cannot all occur exactly at time  $\tau$ . The Giotto compiler takes a Giotto program, a platform specification, and a jitter tolerance, and if possible, generates platform code that lies within the jitter tolerance of Giotto semantics. Specifically, for every program execution, the compiler must attempt to produce a schedule that indicates when and where the Giotto activities are performed. Such a schedule may not exist, because the scheduling problem can be overconstrained. An overconstrained scheduling problem may become solvable without changing the Giotto program, by a combination of the following: increase the number of hosts, decrease the worst-case execution times, or increase the jitter tolerance.

##### A. Scheduling Giotto

We define an *abstract* Giotto scheduling problem. The problem is abstract, as we include only scheduling constraints that need to be met by all Giotto implementations. Any particular, *concrete* implementation may have to take into account additional scheduling constraints.

**Jobs.** Let  $G$  be a Giotto program. A *job* of  $G$  is a pair  $j[k]$  consisting of a *job action*  $j$  and a *job instance*  $k \in \mathcal{I}$ , chosen from some index set  $\mathcal{I}$ . We distinguish between *computation jobs* and *communication jobs*. The action of a computation job is either a task  $t \in \text{Tasks}$ , or  $\text{true}(d)$  or  $\text{false}(d)$  for a driver  $d \in \text{Drivers}$ , or  $\text{read}(s)$  for a sensor port  $s \in \text{SensePorts}$ . The

action  $t$  executes the task  $t$ ; the actions  $true(d)$  and  $false(d)$  represent the execution of driver  $d$  in cases where the outcome of the driver guard is true or false, respectively; the action  $read(s)$  loads a new sensor value into the port  $s$ . We write

$$\begin{aligned} \text{Jobs} &= \text{Tasks} \cup \\ &\quad \{true(d), false(d) \mid d \in \text{Drivers}\} \cup \\ &\quad \{read(s) \mid s \in \text{SensePorts}\} \end{aligned}$$

for the set of computation actions. For every computation action  $j \in \text{Jobs}$ , the set  $r(j) \subseteq \text{Ports}$  of *read ports* and the set  $w(j) \subseteq \text{Ports}$  of *written ports* are defined as follows:

- If  $j = t$  for  $t \in \text{Tasks}$ , then  $r(j) = \text{In}[t] \cup \text{Priv}[t]$  and  $w(j) = \text{Out}[t] \cup \text{Priv}[t]$ .
- If  $j = true(d)$ , then  $r(j) = \text{Src}[d]$  and  $w(j) = \text{Dst}[d]$ .
- If  $j = false(d)$ , then  $r(j) = \text{Src}[d]$  and  $w(j) = \emptyset$ .
- If  $j = read(s)$ , then  $r(j) = \emptyset$  and  $w(j) = \{s\}$ .

The action of a communication job has the form  $send(p)$ , for a port  $p \in \text{Ports}$ , and its purpose is to broadcast the value of  $p$  over a network to all hosts of the platform. Other models of communication are possible, but not addressed here.

Let  $E = C_0, C_1, C_2, \dots$  be an execution of  $G$ . For each position  $i \geq 0$  and  $1 \leq \ell \leq 9$ , we write  $C_{i,\ell}$  for the program configuration obtained from  $C_i$  by performing the Giotto micro steps 1 through  $\ell$ , as defined in Section III-B. The execution  $E$  gives rise to a set  $\mathcal{J}_E$  of computation jobs. For these jobs we use the index set  $\mathcal{I}_E = (\mathbb{N} \cup \{0\}) \times \{1, \dots, 9\}$ , where the index  $(i, \ell)$  refers to the program configuration  $C_{i,\ell}$ . We write  $<$  for the lexicographic order on  $\mathcal{I}_E$ ; that is,  $(i_1, \ell_1) < (i_2, \ell_2)$  if either  $i_1 < i_2$ , or both  $i_1 = i_2$  and  $\ell_1 < \ell_2$ . The set  $\mathcal{J}_E$  is the smallest set of jobs containing the following:

- [Task jobs] If  $(\cdot, t, \cdot)$  is a task invocation that is completed at configuration  $C_i$ , then  $t[i, 1] \in \mathcal{J}_E$ .
- [Actuator jobs] If  $(\cdot, d)$  is an actuator update that is enabled at configuration  $C_{i,1}$ , then  $true(d)[i, 2] \in \mathcal{J}_E$ . If  $(\cdot, d)$  is evaluated but not enabled at  $C_{i,1}$ , then  $false(d)[i, 2] \in \mathcal{J}_E$ .
- [Sensor jobs] If  $(\cdot, \cdot, d)$  is a mode switch that is evaluated at configuration  $C_{i,3}$ , or  $(\cdot, \cdot, d)$  is a task invocation that is evaluated at configuration  $C_{i,6}$ , and  $s \in \text{Src}[d]$  for a sensor port  $s \in \text{SensePorts}$ , then  $read(s)[i, 3] \in \mathcal{J}_E$ .
- [Mode-driver jobs] If  $(\cdot, \cdot, d)$  is a mode switch that is enabled at configuration  $C_{i,3}$ , then  $true(d)[i, 4] \in \mathcal{J}_E$ . If  $(\cdot, \cdot, d)$  is evaluated but not enabled at configuration  $C_{i,3}$ , then  $false(d)[i, 4] \in \mathcal{J}_E$ .
- [Task-driver jobs] If  $(\cdot, \cdot, d)$  is a task invocation that is enabled at configuration  $C_{i,6}$ , then  $true(d)[i, 7] \in \mathcal{J}_E$ . If  $(\cdot, \cdot, d)$  is evaluated but not enabled at  $C_{i,6}$ , then  $false(d)[i, 7] \in \mathcal{J}_E$ .

The jobs in  $\mathcal{J}_E$  are called the *computation jobs induced by the execution  $E$*  of the program  $G$ .

The interaction between the jobs in  $\mathcal{J}_E$  constrains the order in which these jobs can be performed: if job  $J_1$  supplies a value to job  $J_2$  via a port, then  $J_1$  must finish before  $J_2$  can begin. For two jobs  $J_1 = j_1[k_1]$  and  $J_2 = j_2[k_2]$  in  $\mathcal{J}_E$  and a port  $p \in \text{Ports}$ , we say that  $J_1$  *writes  $p$  to  $J_2$*  (in symbols,  $J_1 \prec_E^p J_2$ ) if (i)  $p \in w(j_1) \cap r(j_2)$  and  $k_1 < k_2$ , and (ii) there is

no job  $J_3 = j_3[k_3]$  in  $\mathcal{J}_E$  such that  $p \in w(j_3)$  and  $k_1 < k_3 < k_2$ . We write  $J_1 \prec_E J_2$  if there is some port  $p$  such that  $J_1 \prec_E^p J_2$ . Note from the definition of  $\mathcal{J}_E$  that a task job  $t[i, 1]$  is added to  $\mathcal{J}_E$  with  $i$  set to the configuration number of the job's completion in order to make the relation  $\prec_E$  capture the fact that the output ports of  $t$  are written when the task completes. Figure 13 shows the precedence constraints between the jobs in  $\mathcal{J}_E$ .

**Platform specifications.** 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. We therefore consider distributed platforms. For simplicity, we restrict our attention to platforms that connect a set of hosts through a broadcast channel, called the *network*; for example, all hosts may be on a common bus. A *platform specification* for the program  $G$  is a triple  $H = (\text{Hosts}, \text{wcet}, \text{wcct})$ :

- $\text{Hosts}$  is a finite set of *hosts*, which represent the processing elements on which computation jobs may execute. We write  $\text{Hosts}_N = \text{Hosts} \cup \{N\}$  for the set of hosts together with the network, which is denoted  $N$ .
- $\text{wcet}: \text{Jobs} \times \text{Hosts} \rightarrow \mathbb{Q}^+$  is a function that assigns to each pair  $(j, h)$ , where  $j$  is a computation action and  $h$  is a host, a *worst-case execution time*, which represents an upper bound on the time required for processing a job of the form  $j[\cdot]$  on host  $h$ . For driver jobs of the form  $true(d)[\cdot, \cdot]$ , the worst-case execution time takes into account both the guard and the function of driver  $d$ ; for driver jobs of the form  $false(d)[\cdot, \cdot]$ , only the driver guard. Methods for obtaining worst-case execution times can be found, for example, in [6], [7].
- $\text{wcct}: \text{Ports} \rightarrow \mathbb{Q}^+$  is a function that assigns to each port  $p$  a *worst-case communication time*, which represents an upper bound on the time required for broadcasting the value of  $p$  over the network.

**Jitter tolerance.** A *jitter tolerance*  $\varepsilon \in \mathbb{Q}^+$  is a positive rational number. Intuitively,  $\varepsilon$  represents the maximal tolerable difference between the actual time of an actuator write (or sensor read), and the time at which the write (or read) is supposed to occur according to the Giotto semantics. In particular, if Giotto specifies an actuator write at 12 ms, then an implementation that conforms with the jitter tolerance  $\varepsilon$  must write the actuator in the interval  $[12 - \varepsilon, 12]$ ; and if Giotto specifies a sensor read at 12 ms, then a conforming implementation must read the sensor in the interval  $[12, 12 + \varepsilon]$  (cf. Figure 13).

**Schedules.** A schedule specifies a possible timing for the jobs that are induced by a program execution. Formally, a *schedule* of the program  $G$  on the set  $\text{Hosts}$  is a function  $S: \mathbb{R} \times \text{Hosts}_N \rightarrow \mathcal{J}$  that maps every time  $\tau \in \mathbb{R}$  and host  $h \in \text{Hosts}_N$  (including the network) to a job in some set  $\mathcal{J}$ . An element in  $\mathcal{J}$  may represent a computation or communication job of  $G$ , or a non-Giotto activity. We require that jobs do not migrate between hosts: if  $S(\tau, h) = S(\tau', h')$ , then  $h = h'$ . We also require that schedules are finitely

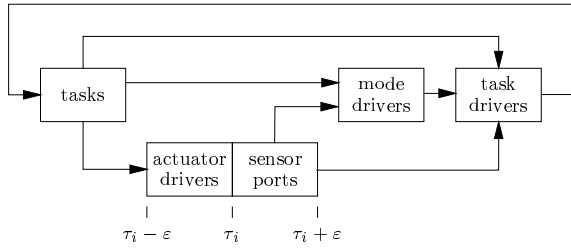


Fig. 13. The precedence and timing constraints for computation jobs.

varying: for all  $h \in \text{Hosts}_N$ , there is no bounded infinite sequence  $\tau_1 < \tau_2 < \tau_3 < \dots$  of reals such that  $S(\tau_1, h) \neq S(\tau_2, h) \neq S(\tau_3, h) \neq \dots$ .

Given a schedule  $S$  and job  $J \in \mathcal{J}$ , we say that  $J$  occurs in  $S$  if there exist  $\tau \in \mathbb{R}$  and  $h \in \text{Hosts}_N$  such that  $S(\tau, h) = J$ ; in this case, we define  $\text{host}_S(J) = h$  and the following:

- The *start time*  $\text{start}_S(J)$  of the job  $J$  in the schedule  $S$  is  $\inf \{\tau \in \mathbb{R} \mid S(\tau, \cdot) = J\}$ . The start time may be  $-\infty$ .
- The *finish time*  $\text{fin}_S(J)$  of the job  $J$  in the schedule  $S$  is  $\sup \{\tau \in \mathbb{R} \mid S(\tau, \cdot) = J\}$ . The finish time may be  $+\infty$ .
- The *execution time*  $\text{exec}_S(J)$  of the job  $J$  in the schedule  $S$  is  $\int_{t \in \{\tau \in \mathbb{R} \mid S(\tau, \cdot) = J\}} 1$ . The execution time may be infinite.

Let  $E$  be an execution of the program  $G$ , and let  $H$  be a platform specification for  $G$ . The schedule  $S$  realizes the program execution  $E$  on a platform specified by  $H$  if the following conditions hold:

- [*Computation jobs*] Every job  $J \in \mathcal{J}_E$  occurs in  $S$  and  $\text{host}_S(J) \neq N$ . Second, if  $J = j[\cdot]$  and  $\text{host}_S(J) = h$ , then  $\text{exec}_S(J) = \text{wcet}(j, h)$ . Third, for all jobs  $J_1, J_2 \in \mathcal{J}_E$ , if  $J_1 \prec_E J_2$  and  $\text{host}_S(J_1) = \text{host}_S(J_2)$ , then  $\text{fin}_S(J_1) \leq \text{start}_S(J_2)$ .
- [*Communication jobs*] For all jobs  $J_1, J_2 \in \mathcal{J}_E$ , if  $J_1 \prec_E^p J_2$  and  $\text{host}_S(J_1) \neq \text{host}_S(J_2)$ , then there exists a communication job  $J = \text{send}(p)[J_1]$  such that (i)  $J$  occurs in  $S$  and  $\text{host}_S(J) = N$ , (ii)  $\text{exec}_S(J) = \text{wcet}(p)$ , and (iii)  $\text{fin}_S(J_1) \leq \text{start}_S(J)$  and  $\text{fin}_S(J) \leq \text{start}_S(J_2)$ . In this case, we say that  $J$  is a *communication predecessor* of  $J_2$ .

Note that because  $S$  is a schedule, rather than an actual run of the Giotto program, it allocates the worst-case execution time for each computation job, and the worst-case communication time for each communication job. The schedule  $S$  conforms to the jitter tolerance  $\epsilon$  if the following conditions hold:

- [*Actuator timing*] For every actuator job  $J = \text{true}(d)[i, 2]$  or  $J = \text{false}(d)[i, 2]$  in  $\mathcal{J}_E$ , where  $d$  is an actuator driver, we have  $\tau_i - \epsilon \leq \text{start}_S(J)$  and  $\text{fin}_S(J) \leq \tau_i$ . Here  $\tau_i$  is the time stamp of the  $i$ -th configuration of the program execution  $E$ .
- [*Sensor timing*] For every sensor job  $J = \text{read}(s)[i, 3]$  in  $\mathcal{J}_E$ , where  $s$  is a sensor port, we have  $\tau_i \leq \text{start}_S(J)$  and  $\text{fin}_S(J) \leq \tau_i + \epsilon$ .

Given a Giotto program  $G$ , a platform specification  $H$ , and a jitter tolerance  $\epsilon$ , a *scheduling function*  $S$  maps every execution  $E$  of  $G$  to a schedule  $S_E$  that realizes  $E$  on  $H$  in

conformance with  $\epsilon$ . The scheduling function  $S$  is *feasible* if for any two executions  $E$  and  $E'$  that agree on the values of all sensor ports up to time  $\tau$ , the schedules  $S_E$  and  $S_{E'}$  are identical up to time  $\tau$ ; more precisely, if  $E = C_0, C_1, C_2, \dots$  and  $E' = C'_0, C'_1, C'_2, \dots$  and  $C_i[\text{SensePorts}] = C'_i[\text{SensePorts}]$  for all  $i \leq k$ , then  $S_E(\tau, h) = S_{E'}(\tau, h)$  for all  $\tau \leq \tau_k$  and all  $h \in \text{Hosts}$ , where  $\tau_k$  is the time stamp of configuration  $C_k$ . Feasibility rules out clairvoyant scheduling functions, which can predict future sensor values. The *abstract Giotto scheduling problem* asks, given  $G$ ,  $H$ , and  $\epsilon$ , if there exists a feasible scheduling function. If not, then the scheduling problem  $(G, H, \epsilon)$  is *overconstrained*.

The scheduling constraints presented in this section are intended to capture a minimal set of constraints: precedences, sensor and actuator timing, and execution and communication times. These constraints are necessary for any implementation of Giotto, but they may not be sufficient. For example, a particular implementation may restrict the amount of information on which a scheduler can base its decisions (according to our definitions, a scheduling decision may depend on all past sensor values), or it may bound the buffer size for storing previous values of a port (according to our definitions, a schedule may send any number of values of a port over the network before any of the values is used), or it may require the transmission of mode-change messages between hosts, etc. By considering the constraints of concrete implementations, the abstract Giotto scheduling problem can be refined into a number of different concrete scheduling problems.

### B. Giotto annotations

An ideal compiler must solve a Giotto scheduling problem by producing a feasible scheduling function or determining that the given problem instance is overconstrained. However, for distributed platforms, the abstract Giotto scheduling problem is NP-hard (it is a generalization of *multiprocessor scheduling* [8]). Algorithms and heuristics for solving similar distributed scheduling problems can be found, for example, in [9], [10], [11], [12]. In practice, a compiler will have a third outcome, namely, that it succeeds neither in generating code nor in proving non-schedulability. In order to aid the compiler in finding a feasible scheduling function in difficult situations, we introduce the concept of Giotto annotations.

The most basic Giotto annotation is the mapping annotation. A particular application may require that tasks be located on specific hosts, e.g., close to the physical processes that the tasks control, or on processors particularly suited for the operations of the tasks. A mapping annotation can be used to express such constraints, and also to reduce the size of the space in which the compiler must look for a feasible scheduling function. Let  $G$  be a Giotto program, and let  $H$  be a platform specification for  $G$ . A *mapping annotation* for  $G$  on  $H$  is a partial function  $\text{host}: \text{Jobs} \rightarrow \text{Hosts}$  that assigns a host of  $H$  to some computation actions of  $G$ . The mapping annotation is *complete* if the function  $\text{host}$  is total. Consider a schedule  $S$  that realizes an execution  $E$  of  $G$  on  $H$ . The schedule  $S$  conforms to the mapping annotation  $\text{host}$  if for all jobs  $J \in \mathcal{J}_E$ , if  $J = j[\cdot]$  and  $\text{host}(j)$  is defined, then  $\text{host}_S(J) = \text{host}(j)$ .

A more detailed Giotto annotation is the scheduling annotation. The exact form of scheduling annotations depends on the platform: a scheduling annotation specifies task priorities, relative deadlines, or time slots, depending on whether the underlying real-time operating system uses a priority-driven, deadline-driven, or time-triggered scheduler. We choose an uncomplicated platform—with preemptive priority scheduling of tasks, and round-robin time-slice scheduling of messages on the network—in order to demonstrate that a precise definition of scheduling annotations is possible; more elaborate annotations would require longer definitions, but not a fundamental change in approach. One can define partial scheduling annotations, which leave some decisions to the system scheduler, but for simplicity, we define only a complete form of scheduling annotation. To be precise, a *scheduling annotation* for the program  $G$  on a platform specified by  $H$  is a tuple (host, priority, slot,  $\delta$ ):

- [Mapping] The function  $\text{host}: \text{Jobs} \rightarrow \text{Hosts}$  is a complete mapping annotation for  $G$  on  $H$ .
- [Task priorities] The function  $\text{priority}: \text{Tasks} \rightarrow \mathbb{N}$  assigns a priority to every task.
- [Communication times] For simplicity, we assume that all communication proceeds in rounds, with each round providing a time slot to every port. The value of a port  $p$  can be broadcast once per round, in the slot provided to  $p$ . Let  $P = |\text{Ports}|$  be the number of ports. The function  $\text{slot}: \text{Ports} \rightarrow \{0, 1, \dots, P-1\}$  is a bijection that assigns a slot number to every port. The positive rational  $\delta \in \mathbb{Q}^+$  is the duration of each time slot. We assume that only one broadcast is possible per time slot; that is,  $\text{wcet}(p) = \delta$  for all ports  $p \in \text{Ports}$ .

Consider a schedule  $S$  that realizes an execution  $E$  of  $G$  on  $H$ . The schedule  $S$  *conforms* to the scheduling annotation (host, priority, slot,  $\delta$ ) if  $S$  conforms to the mapping annotation host and the following conditions hold:

- [Task priorities] Consider a job  $J$  that occurs in the schedule  $S$ . The job  $J$  is *completed* in  $S$  at time  $\tau$  if  $\text{fin}_S(J) \leq \tau$ . The job  $J$  is *enabled* in  $S$  at time  $\tau$  if for all jobs  $J'$  that occur in  $S$ , if  $J' \prec_E J$  or  $J'$  is a communication predecessor of  $J$ , then  $J'$  is completed at  $\tau$ . For all times  $\tau \in \mathbb{R}$ , all hosts  $h \in \text{Hosts}$ , and all task jobs  $J_1 = t_1[i_1, 1]$  and  $J_2 = t_2[i_2, 1]$  in  $\mathcal{J}_E$ , if  $S(\tau, h) = J_1$  and  $\text{host}(t_2) = h$  and  $J_2$  is enabled in  $S$  at time  $\tau$ , then  $\text{priority}(t_1) \geq \text{priority}(t_2)$ .
- [Communication times] For every communication job  $J = \text{send}(p)[\cdot]$  that occurs in  $S$ , there exists a round number  $n \in \mathbb{N} \cup \{0\}$  such that  $\delta \cdot (n \cdot P + \text{slot}(p)) \leq \text{start}_S(J)$  and  $\text{fin}_S(J) \leq \delta \cdot (n \cdot P + \text{slot}(p) + 1)$ .

A Giotto program with annotations is a formal refinement of the program: the Giotto semantics, as defined in Section III-B, is not changed by the annotations, but the number of feasible scheduling functions may be reduced. The *annotated Giotto scheduling problem* asks, given a Giotto program  $G$ , a platform specification  $H$ , a jitter tolerance  $\varepsilon$ , and a (mapping or scheduling) annotation  $A$ , if there is a feasible scheduling function  $S$  such that for every execution  $E$  of  $G$ , the schedule  $S_E$  conforms to the annotation  $A$ . If the abstract Giotto scheduling

problem  $(G, H, \varepsilon)$  has a solution, but the annotated problem  $(G, H, \varepsilon, A)$  does not, then the annotation  $A$  is *invalid*. Invalid annotations constrain the program in a way that rules out all feasible scheduling functions.

Mapping and scheduling annotations, as defined above, provide only one example of how a Giotto program can be mapped onto a particular kind of platform. According to the definitions, mapping annotations occur strictly prior to scheduling annotations. In general, we believe that it is advantageous to arrange Giotto annotations in multiple levels. Such a structured view supports the incremental refinement of a Giotto program into an executable image. The multilayered approach suggests a modular architecture for the Giotto compiler with separate modules for, say, mapping and scheduling. The compiler may attempt to solve the scheduling problem on any annotation level, and if it fails to do so, it may ask for more detailed annotations at a lower level. At every level, the annotation must be checked for validity, that is, for consistency with the annotations at the higher levels and with the Giotto semantics. Such a compiler can be evaluated along several dimensions: (i) how many annotations it requires to generate 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.

### C. Example

To illustrate the flexibility afforded to the Giotto compiler, we present several possible schedules for an execution of the Giotto program from Figure 11. The platform specification  $\hat{H} = (\text{Hosts}, \text{wcet}, \text{wcet})$  consists of a single host ( $\text{Hosts} = \{h\}$ ) and the following worst-case execution times:

$$\begin{aligned} \text{wcet}(\text{read}(s_1), h) &= 0.25 \\ \text{wcet}(\text{read}(s_2), h) &= 0.25 \\ \text{wcet}(t_1, h) &= 0.25 \\ \text{wcet}(t_2, h) &= 0.5 \\ \text{wcet}(t_3, h) &= 1.0 \\ \text{wcet}(\text{true}(d_1), h) &= 0.25 \\ \text{wcet}(\text{true}(d_2), h) &= 0.5 \\ \text{wcet}(\text{true}(d_3), h) &= 1 \\ \text{wcet}(\text{true}(d_4), h) &= 1 \\ \text{wcet}(\text{true}(d_5), h) &= \text{wcet}(\text{false}(d_5), h) = 0.5 \end{aligned}$$

Since  $\text{Hosts}$  is a singleton set, we need not define  $\text{wcet}$ . The jitter tolerance is  $\hat{\varepsilon} = 1$ .

Consider the sample execution  $\hat{E}$  pictured in Figure 12. In  $\hat{E}$ , sensors are read and actuators are written at precisely the time instants specified by the Giotto semantics. This precision is clearly impossible to attain if sensor reads and actuator writes take a non-negligible amount of time. Further, in  $\hat{E}$ , the second invocation of task  $t_1$  executes between 6 and 12 ms. This requirement may be too strict, and if insisted upon would prevent some Giotto programs from being schedulable. Instead, what is required is that the second invocation of  $t_1$  executes after all its input port values are available, and before any job that needs its output port values.

Figure 14 shows the constraints on timing and precedences for the computation jobs that are induced by the execution  $\hat{E}$ ;

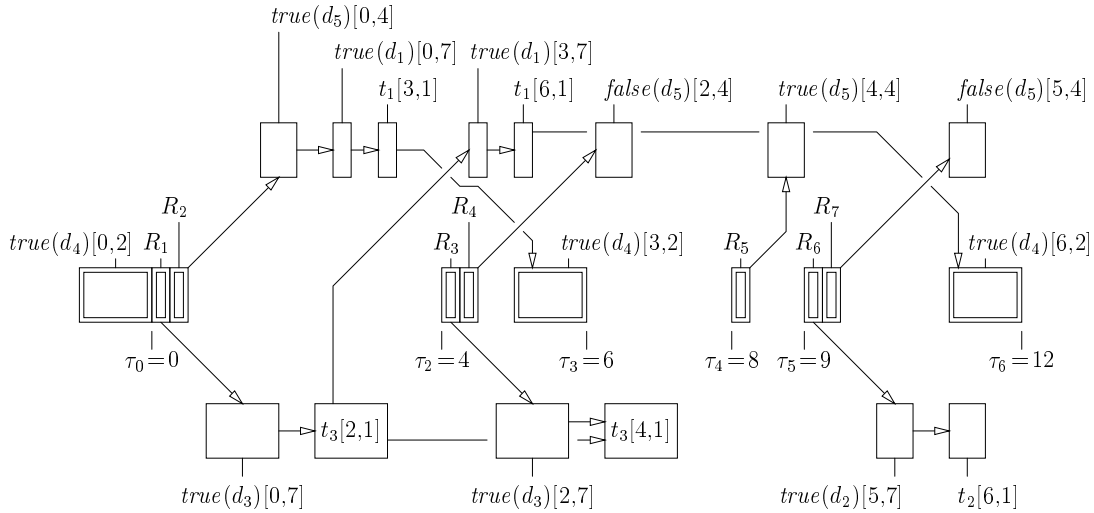


Fig. 14. The precedence and timing constraints for the program execution of Figure 12. Here  $R_1$  is an abbreviation for  $read(s_1)[0, 3]$ . Similarly,  $R_2, \dots, R_7$  are, respectively, abbreviations for  $read(s_2)[0, 3]$ ,  $read(s_1)[2, 3]$ ,  $read(s_2)[2, 3]$ ,  $read(s_2)[4, 3]$ ,  $read(s_1)[5, 3]$ , and  $read(s_2)[5, 3]$ .

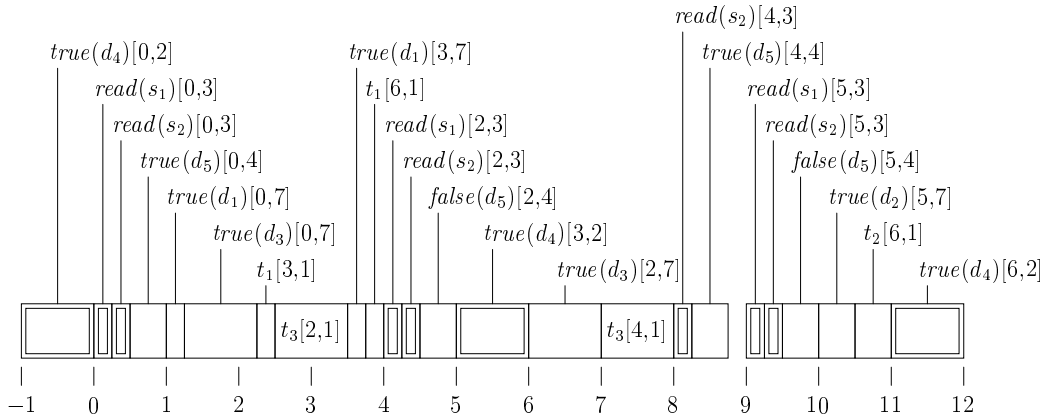


Fig. 15. A schedule for the program execution of Figure 12.

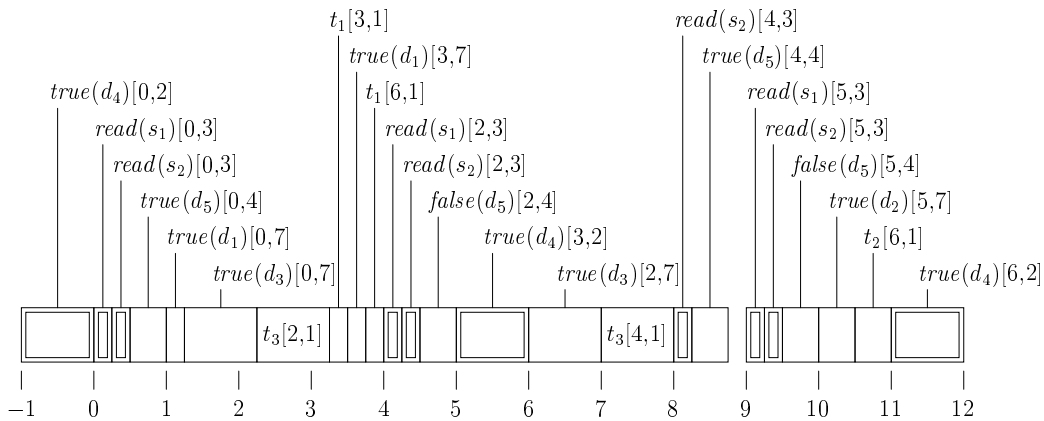


Fig. 16. A second schedule for the program execution of Figure 12.

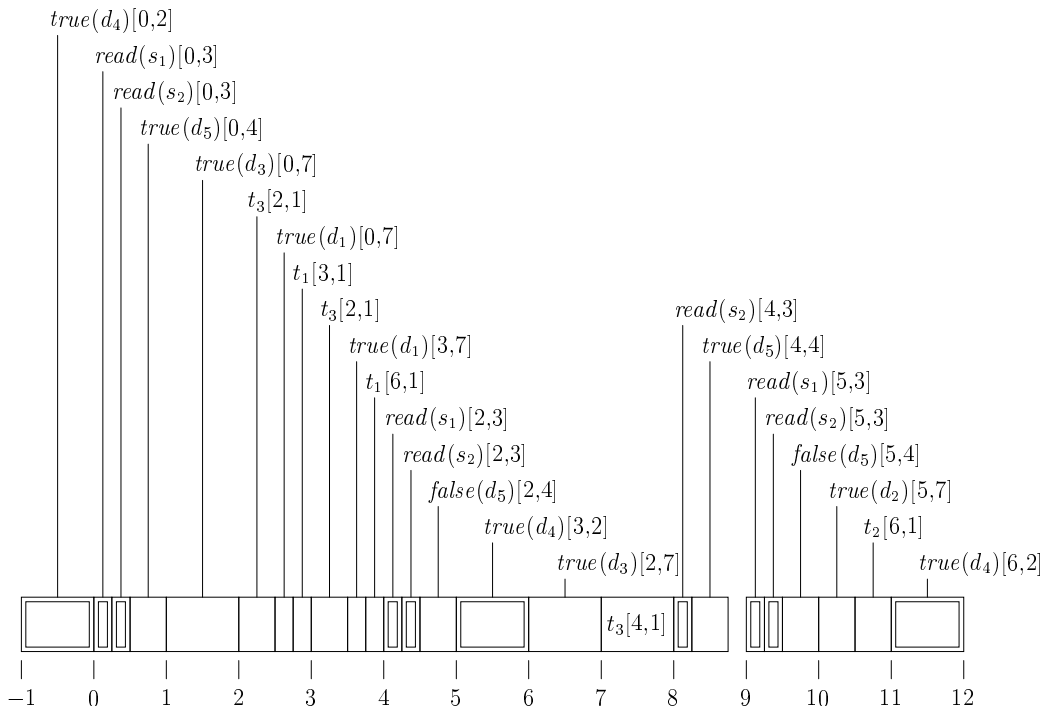


Fig. 17. A third schedule for the program execution of Figure 12.

these are the constraints that appear in the definition of the realization of an execution, and in the definition of conformance with the jitter tolerance. Boxes with a double border represent sensor and actuator jobs. These jobs are special, because their execution is constrained to happen at specific times. The remaining boxes are jobs that execute tasks, mode drivers, and task drivers. These jobs may execute at any time, provided they meet all precedence constraints. For example,  $read(s_1)[0, 3]$  precedes  $true(d_3)[0, 7]$ , because the sensor job  $read(s_1)[0, 3]$  provides the sensor value to the task-driver job  $true(d_3)[0, 7]$ . Not also that in Figure 14 jobs of the form  $false(d_i)[\cdot, \cdot]$  do not precede other jobs, as a driver does not write any ports if its guard evaluates to false.

Figure 15 shows a schedule that realizes the execution  $\hat{E}$  on a platform specified by  $\hat{H}$  and conforms to the jitter tolerance  $\hat{\epsilon} = 1$ . To understand what Figure 15 represents, consider the interval from  $-1$  to  $3.5$ . First the actuator job  $true(d_4)[0, 2]$  executes; this job updates the actuator port  $a$ . Then the sensor jobs  $read(s_1)[0, 3]$  and  $read(s_2)[0, 3]$  execute; these jobs update the sensor ports  $s_1$  and  $s_2$ . Next, the mode-driver job  $true(d_5)[0, 4]$  executes, indicating a mode change, followed by the task-driver jobs  $true(d_1)[0, 7]$  and  $true(d_3)[0, 7]$ . Finally, the task jobs  $t_1[3, 1]$  and  $t_3[2, 1]$ , corresponding to the first invocations of tasks  $t_1$  and  $t_3$ , execute. Note that the driver job for the second invocation of task  $t_1$ , namely  $true(d_1)[3, 7]$ , as well as the task itself,  $t_1[6, 1]$ , execute in advance of 6 ms. This is permissible, because  $true(d_1)[3, 7]$  needs only the value of port  $o_2$  produced by the first invocation  $t_3[2, 1]$  of task  $t_3$ , which is complete at 3.5 ms. The schedule of Figure 15 conforms to a scheduling

annotation with  $priority(t_1) > priority(t_3)$ ; for example, at 2.25 ms  $t_1[3, 1]$  and  $t_3[2, 1]$  are both enabled, but  $t_1[3, 1]$  executes.

Figure 16 shows a second schedule that realizes the execution  $\hat{E}$  on a platform specified by  $\hat{H}$  and conforms to the jitter tolerance  $\hat{\epsilon} = 1$ . The schedule of Figure 16 conforms to a scheduling annotation with  $priority(t_3) > priority(t_1)$ . Figure 17 shows a third schedule for the same execution, conforming to a scheduling annotation with  $priority(t_1) > priority(t_3)$ . In this schedule, task  $t_3$  is preempted at 2.5 ms by the driver for task  $t_1$  and then by task  $t_1$  itself.

## V. 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 step towards separating *reactivity* from *schedulability*. The term reactivity expresses what we mean by control-systems aspects: the system's functionality, in particular, the control laws, and the system's timing requirements. The term schedulability expresses what we mean by platform-dependent aspects, such as platform performance, platform utilization (scheduling), and fault tolerance. 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 [5]. The separation of reactivity from schedulability has several important ramifications. First, the 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 [13]. Second, a Giotto program specifies reactivity in a modular fashion, which facilitates the exchange and addition of functionality. For example, functionality code (i.e., tasks and driver functions) can be packaged as software components and reused. Third, as increasingly powerful Giotto compilers become available, the embedded-software development effort is significantly reduced. The tedious programming of scheduling code is replaced by compilation, which eliminates a common source of errors. Fourth, Giotto is compatible with any scheduling strategy, which therefore becomes a parameter of the Giotto compiler. There are essentially two reasons why even the best Giotto compiler may fail to generate executable code: (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, and performance can be improved by faster or more parallel hardware or leaner functionality code.

#### A. Current 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 Mindstorms robots. The robots used 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 used 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 was done by an infrared-Ethernet bridge implemented on a PC. For an informal discussion of these implementations, and embedded control-systems development with Giotto in general, can be found in [14].

In collaboration with Marco Sanvido and Walter Schaufelberger at ETH Zürich, we built a high-performance implementation of a Giotto system on a single StrongARM SA-110 processor that controls an autonomously flying model helicopter [15]. We started from an existing implementation of the helicopter control system [16], which included a custom-designed real-time operating system called HelyOS and control software written in a subset of Oberon [17] suited for embedded real-time systems. We reimplemented the existing software as a combination of a Giotto program and Oberon code that implements the task and driver functions. Much of the existing functionality code could be reused. The Giotto program for the helicopter consists of six Giotto modes such as “take-off” and “hover.” The hover mode, for example, contains a 40 Hz controller task and a 200 Hz data-fusion task.

For this project, we developed a Giotto compiler that targets a virtual real-time machine, called the *Embedded Machine* [5]. Embedded Machine code, also called *E code*, supervises the timing of functionality code, which can be written in any conventional programming language such as C. An Embedded Machine-based Giotto run-time system consists of an implementation of the Embedded Machine together with the scheduler of a real-time operating system. While E code is interpreted by the Embedded Machine, functionality code is native code that is scheduled for execution by the system scheduler. For E code that is generated from a Giotto source program, the scheduling problem is more constrained than the abstract Giotto scheduling problem defined in Section IV-A, but still independent of any particular system scheduler; it is only required that the scheduler be compatible with the schedulability test of the Giotto compiler [18]. E code produced by the compiler can be executed on any platform for which an Embedded Machine implementation is available. For the helicopter project, we implemented the Embedded Machine on top of HelyOS.

We also implemented a Giotto-based electronic throttle controller on a single Motorola MPC-555 processor running the real-time operating system OSEKWorks. For this purpose, we ported the Embedded Machine to OSEKWorks, which is widely used in the automotive industry. In addition to these real-time versions of the Embedded Machine, non-real-time implementations of the Embedded Machine are available for Linux and Windows.

#### B. Related work

Giotto is inspired by the time-triggered architecture (TTA) [4], 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) [19]. 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 for the Mars system, a predecessor of the TTA, distinguishes in a similar way “programming in the large” from “programming in the small” [20]. The inter-task communication semantics of Giotto is particularly similar to the MetaH language [21], [22], which is designed for real-time, distributed avionics applications. MetaH supports periodic real-time tasks, multimodal control, and distributed implementations. Giotto can be viewed as capturing a time-triggered fragment of MetaH in an abstract and formal way. Unlike MetaH, Giotto 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 [3], such as Esterel [23], Lustre [24], and Signal [25]. 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 scheduling constraints (in Giotto all values are, semantically, always well-defined). Thus, Giotto can be seen as identifying a class of synchronous reactive programs that support typical real-time control applications and efficient code generation [5].

**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 [26]. We thank Marco Sanvido for his suggestions on the design of the Giotto drivers. We thank Paul Griffiths for implementing the functionality code of the electronic throttle controller.

## REFERENCES

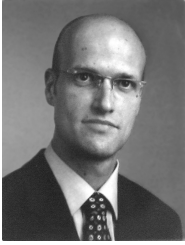
- [1] D. Langer, J. Rauch, and M. Röbber, “Fly-by-wire systems for military high-performance aircraft,” in *Real-time Systems: Engineering and Applications*. Kluwer, 1992, pp. 369–395.
- [2] R. Collinson, “Fly-by-wire flight control,” *Computing and Control Engineering*, vol. 10, no. 4, pp. 141–152, 1999.
- [3] N. Halbwachs, *Synchronous Programming of Reactive Systems*. Kluwer, 1993.
- [4] H. Kopetz, *Real-time Systems: Design Principles for Distributed Embedded Applications*. Kluwer, 1997.
- [5] T. Henzinger and C. Kirsch, “The Embedded Machine: Predictable, portable real-time code,” in *Proc. of the Conference on Programming Language Design and Implementation*. ACM, 2002, pp. 315–326.
- [6] S. Malik and Y.-T. Li, *Performance Analysis of Real-Time Embedded Software*. Kluwer, 1999.
- [7] H. Theiling, C. Ferdinand, and R. Wilhelm, “Fast and precise WCET prediction by separated cache and path analyses,” *Real-Time Systems*, vol. 18, no. 2–3, pp. 157–179, 2000.
- [8] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979.
- [9] K. Tindell and J. Clark, “Holistic schedulability for distributed hard real-time systems,” *Microprocessing and Microprogramming*, vol. 40, pp. 117–134, 1994.
- [10] P. Eles, K. Kuchcinski, Z. Peng, A. Doboli, and P. Pop, “Process scheduling for performance estimation and synthesis of hardware/software systems,” in *Proc. of the EUROMICRO Conference*. IEEE, 1998, pp. 168–175.
- [11] P. Brucker, *Scheduling Algorithms*. Springer-Verlag, 2001.
- [12] M. Pinedo, *Scheduling: Theory, Algorithms, and Systems*. Prentice-Hall, 2002.
- [13] T. Henzinger, “Masaccio: A formal model for embedded components,” in *Proc. of the IFIP Conference on Theoretical Computer Science*, vol. 1872 of Lecture Notes in Computer Science. Springer-Verlag, 2000, pp. 549–563.
- [14] T. Henzinger, B. Horowitz, and C. Kirsch, “Embedded control systems development with Giotto,” in *Proc. of the Conference on Languages, Compilers, and Tools for Embedded Systems*. ACM, 2001, pp. 64–72.
- [15] C. Kirsch, M. Sanvido, T. Henzinger, and W. Pree, “A Giotto-based helicopter control system,” in *Proc. of the Intl. Workshop on Embedded Software*, vol. ??? of Lecture Notes in Computer Science. Springer-Verlag, 2002.
- [16] J. Chapuis, C. Eck, M. Kottmann, M. Sanvido, and O. Tanner, “Control of helicopters,” *Control of Complex Systems*, pp. 359–392, 1999.
- [17] N. Wirth and J. Gutknecht, *Project Oberon: The Design of an Operating System and Compiler*. ACM, 1992.
- [18] T. Henzinger, C. Kirsch, R. Majumdar, and S. Matic, “Time-safety checking for embedded programs,” in *Proc. of the Intl. Workshop on Embedded Software*, vol. ??? of Lecture Notes in Computer Science. Springer-Verlag, 2002.
- [19] P. Clements, “A survey of architecture description languages,” in *Proc. of the Intl. Workshop on Software Specification and Design*. IEEE, 1996, pp. 16–25.
- [20] 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*, vol. 6, no. 3, pp. 72–82, 1991.
- [21] S. Vestal and P. Binns, “Scheduling and communication in MetaH,” in *Proc. of the Real-time Systems Symposium*. IEEE, 1993, pp. 194–200.
- [22] S. Vestal, “MetaH support for real-time multi-processor avionics,” in *Proc. of the Joint Workshop on Parallel, Distributed, and Object-Oriented Real-Time Systems*. IEEE, 1997, pp. 11–21.
- [23] G. Berry, “The foundations of Esterel,” in *Proof, Language and Interaction: Essays in Honour of Robin Milner*, G. Plotkin, C. Stirling, and M. Tofte, Eds. MIT Press, 2000, pp. 425–454.
- [24] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, “The synchronous dataflow programming language Lustre,” *Proc. of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.
- [25] A. Benveniste, P. L. Guernic, and C. Jacquemot, “Synchronous programming with events and relations: The Signal language and its semantics,” *Science of Computer Programming*, vol. 16, no. 2, pp. 103–149, 1991.
- [26] J. Davis, M. Goel, C. Hylands, B. Kienhuis, E. 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,” University of California, Berkeley, Tech. Rep. UCB/ERL M99/44, 1999.



Thomas A. Henzinger is a Professor of Electrical Engineering and Computer Sciences at the University of California, Berkeley. He holds a Dipl.-Ing. degree in Computer Science from Kepler University in Linz, Austria, an M.S. degree in Computer and Information Sciences from the University of Delaware, and a Ph.D. degree in Computer Science from Stanford University (1991). He was an Assistant Professor of Computer Science at Cornell University (1992–95), and a Director of the Max-Planck Institute for Computer Science in Saarbrücken, Germany (1999). His research focuses on formalisms and tools for the design, implementation, and verification of reactive, real-time, and hybrid systems.



Benjamin Horowitz received his B.A. in Philosophy from Wesleyan University in 1994, studied computer science at the University of Massachusetts, Amherst from 1995 to 1997, and since 1997 has been a Ph.D. student in the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley. His research interests include real-time programming languages, scheduling theory, and the design of embedded systems.



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