

Embedded Control Systems Development with Giotto^{*†}

Thomas A. Henzinger Benjamin Horowitz Christoph Meyer Kirsch

Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA 94720-1770
{tah,bhorowit,cm}@eecs.berkeley.edu

Abstract. Giotto is a principled, tool-supported design methodology for implementing embedded control systems on platforms of possibly distributed sensors, actuators, CPUs, and networks. Giotto is based on the principle that time-triggered task invocations plus time-triggered mode switches can form the abstract essence of programming control systems. Giotto consists of a programming language with a formal semantics, a retargetable compiler, and a runtime library. Giotto supports the automation of control system design by strictly separating platform-independent functionality and timing concerns from platform-dependent scheduling and communication issues. The time-triggered predictability of Giotto makes it particularly suitable for safety-critical applications with hard real-time constraints. We illustrate the platform independence and time-triggered execution of Giotto by coordinating a heterogeneous flock of Intel x86 robots and Lego Mindstorms robots.

1 Introduction

Embedded software development for control applications consists of two phases: first modeling, then implementation. Modeling control applications is usually done by control engineers with support from tools such as Matlab or MatrixX. While these tools offer limited code-generation facilities, the efficient implementation of control designs remains a challenging subdiscipline of software engineering. Control designs impose hard real-time requirements, which software engineers traditionally meet by tightly coupling model, code, and platform. We advocate a decoupling of these domains.

Throughout this paper, the term *platform* denotes a hardware configuration, operating system, and communication protocol. Platforms, which may be distributed, consist of sensors, actuators, CPUs, and networks. Platform-independent issues include application

^{*}This research was supported in part by the DARPA SEC grant F33615-C-98-3614, the MARCO GSRC grant 98-DT-660, the AFOSR MURI grant F49620-00-1-0327, and the NSF ITR grant CCR-0085949.

[†]A preliminary version of this paper appeared in the *Proceedings of the ACM Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2001.

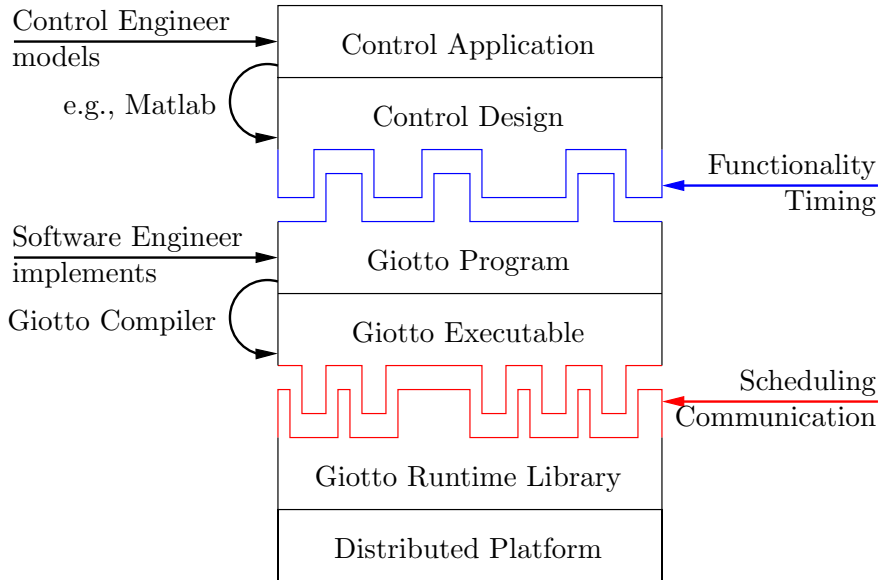


Figure 1: Embedded control systems development with Giotto

functionality and timing. In contrast, platform-dependent issues include scheduling, communication, and physical performance. The key to automating embedded software development is to understand the interface between platform-independent and platform-dependent issues. Such an interface —i.e., an abstract programmer’s model for embedded systems— enables decoupling software design from implementation, even for distributed platforms and even in the presence of hard real-time requirements.

Giotto provides an abstract programmer’s model based on the time-triggered paradigm. In a *time-triggered system*, communication —from sensors, between CPUs, to actuators— is triggered by the tick of a notional global clock. In a distributed system, such a clock can be provided by a clock synchronization service. The time-triggered architecture (TTA) [Kop97] offers a hardware and protocol realization of the time-triggered paradigm, and therefore a natural platform for a Giotto implementation. The TTA has recently gained momentum in safety-critical automotive applications, where timing predictability is essential. While Giotto offers the predictability of time-triggered systems, it also offers the flexibility of platform independence. In particular, it is possible to use a clock synchronization service on top of any real-time communication protocol to implement Giotto. For example, recently the event-triggered CAN standard has been extended to TTCAN [FMD⁺00], which includes a clock synchronization service and thus offers another attractive platform for implementing Giotto.

The two central ingredients of Giotto are periodic task invocations and time-triggered mode switches. More precisely, a *Giotto program* specifies a set of *modes*. Each mode determines a set of *tasks* and a set of *mode switches*. At every time instant, the program execution is in one specific mode, say, M . Each task of M has a real-time frequency and is invoked at this frequency as long as the mode M remains unchanged. A task invocation typically causes non-trivial computational activity, such as the calculation of desired actua-

tor values. Each mode switch of M has a real-time frequency, a condition that is evaluated at this frequency, and a target mode, say, N : if the condition evaluates to true, then the new mode is N . In the new mode, some tasks may be removed, and others added.

Giotto has a formal semantics that specifies the meaning of mode switches, of intertask communication, and of communication with the program environment. The environment consists of sensors and actuators. A Giotto program determines the functionality (input and output behavior) of concurrent periodic tasks, and the timing of the program's interaction with its environment. Functionality and timing are the key elements of the interface between control design and implementation. A Giotto program does not specify platform-dependent aspects such as priorities and other scheduling and communication directives. Giotto's strength is its simplicity: Giotto is compatible with any choice of real-time operating system (RTOS), scheduling algorithm, and real-time communication protocol. Moreover, Giotto's simplicity allows us to automate schedule and code generation.

The *Giotto compiler* is an essential part of the methodology. A Giotto program is a platform-independent specification of a control software design, from which the Giotto compiler synthesizes embedded software for a given platform. The Giotto tasks are given, say, as C code. The tasks' worst-case execution times (WCETs) are known by the Giotto compiler.¹ Given a platform, the compiler maps tasks to CPUs. The compiler then computes a task and communication schedule that guarantees the timing requirements of the Giotto program. Compilation of the same program for platforms with different resource and performance characteristics will result in different task mappings, and different task and communication schedules. Since the synthesis problem is difficult for distributed platforms, a Giotto compiler may fail to find a feasible schedule, even if such a schedule exists. For this case, we propose *Giotto annotations*, which allow the programmer to give directives that aid the compiler in finding a feasible schedule. A Giotto annotation constrains the compiler to a nonempty subset of the permissible schedules.

Figure 1 summarizes the design flow of embedded control systems development with Giotto. First the control and software engineers agree on the functionality and timing of a design, specified as a Giotto program. Then the software engineer uses the Giotto compiler to map the program to a given platform. The Giotto compiler produces an executable, which can then be linked against the Giotto runtime library. The Giotto runtime library provides a layer of scheduling and communication primitives. This layer defines the interface between the Giotto executable and a platform. We have developed a Giotto runtime library for Wind River's VxWorks RTOS on Intel x86 targets. We are currently in the process of porting the library to other platforms.

Figure 2 shows a more detailed picture of the design flow from a Giotto program to a Giotto executable using Giotto annotations. For distributed platforms, if the compiler is unable to find a permissible schedule, the programmer may use Giotto annotations to give directives to the compiler on how to map tasks to hosts and how to schedule resources. A Giotto program can be gradually refined with more and more specific annotations, until the compiler is able to generate a mapping and a schedule that meet the timing requirements. Giotto annotations fall into three increasingly specific classes of directives: Giotto-H anno-

¹The difficult problem of estimating WCETs is orthogonal to the problems that Giotto addresses. For example, for complex processor architectures with cache and pipeline mechanisms, abstract interpretation can be used to generate integer linear programs for WCET prediction [TFW00].

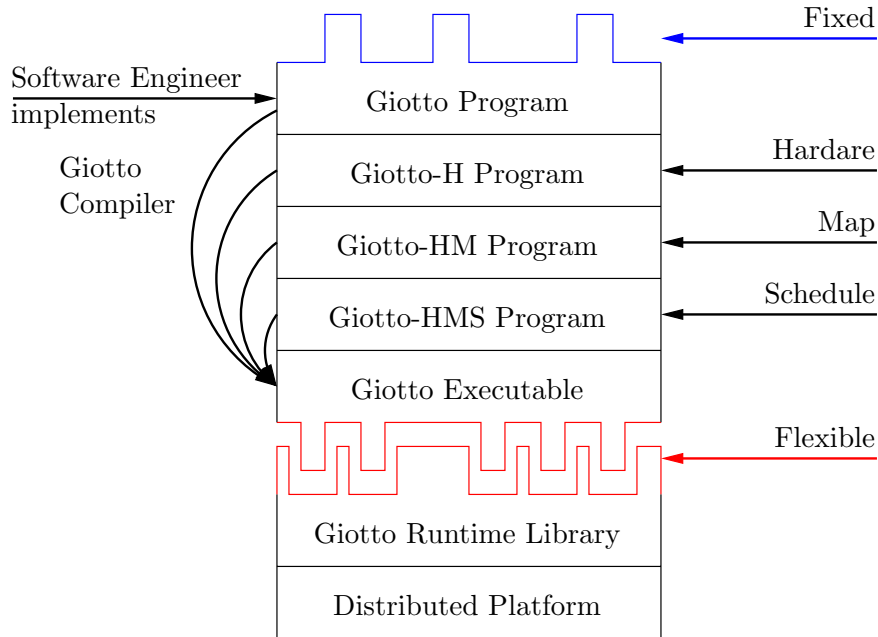


Figure 2: Semi-automatic compilation with annotated Giotto

tations specify the hardware and its performance; Giotto-M annotations map tasks to CPUs and communications to networks; and Giotto-S annotations schedule tasks and communications. It is important to note that the functionality and timing properties of a Giotto program are not affected by Giotto annotations. Rather, the annotations provide a means of incrementally refining a platform-independent Giotto program into an executable for a specific platform.

In the next section we introduce the pure, platform-independent version of the Giotto programming language. In Section 3, we use an example to illustrate embedded control systems development using Giotto. The example requires the coordination of a heterogeneous flock of Intel x86 robots and Lego Mindstorms robots. The example is implemented by a Giotto program first discussed in Section 4, and then refined in Section 5 with Giotto annotations. We relate Giotto to existing work and conclude in Section 6. For a complete and detailed exposition of Giotto, including formal definitions of abstract syntax and semantics, see [HHK01].

2 The Giotto Programming Language

Giotto is a programming language that aims at distributed hard real-time applications with periodic behavior, such as control systems. A typical control system periodically reads sensor information, computes control laws, and writes the results to actuators. Moreover, such a control system may react to changes in its environment by switching control laws as well as periodicity. Giotto’s language primitives match these requirements of distributed hard real-time control applications.

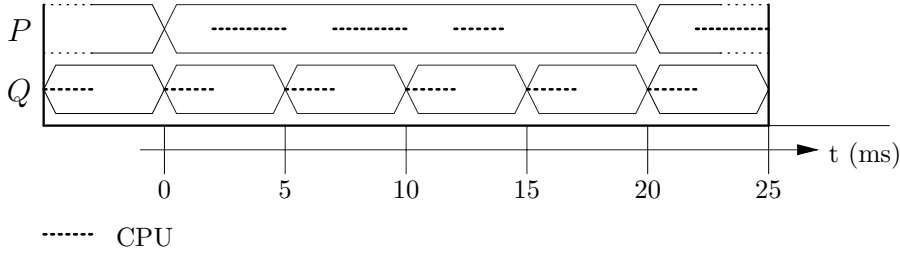


Figure 3: The timing diagram for the two Giotto tasks P and Q

A *Giotto port* is a physical location in memory. A Giotto port may either be associated with a sensor or actuator, or else may be used for intertask communication. A *Giotto task* is a periodic job, say, computing a control law. A Giotto task has input, private, and output ports, and an implementation with known WCET, written in any programming language. Thus Giotto can be seen as a real-time extension of a standard non-embedded programming language. A Giotto task is invoked with a certain *period*, given in milliseconds (ms). A Giotto task may carry state in its private ports in order to keep track of its past invocations. Thus, at each invocation, the task reads its input and private ports, and computes new values for its output and private ports.

The Giotto semantics requires that the input and output ports of a Giotto task are updated logically at the beginning and the end of the task's period, respectively. However, a Giotto task does not have to be started at the beginning of its period. A Giotto task only has to be started and be finished sometime during its period. Consider Figure 3, which shows the timing diagram of a 20ms Giotto task P and a 5ms Giotto task Q running on a single CPU. The dotted lines give one possible scenario indicating which task currently runs on the CPU. At the 0ms time instant, both P and Q read the values of their input ports. At 5ms, the result of the computation of task Q is written to its output ports although, in this scenario, Q finished its execution earlier (as indicated by the dotted line). After 20ms and three more invocations of task Q , the result of the computation of task P is written to its output ports. Note that, in this scenario, P finished its execution already before the fourth invocation of Q . The Giotto semantics does not specify the physical CPU scheduling of the execution of Giotto tasks; there is only the requirement that the execution of a Giotto task is finished within the task's period. It is up to the compiler to use a scheduling mechanism that guarantees the deadlines (or declare the program to be invalid, if no permissible schedule exists; for example, if the WCET exceeds the period of a task). In our scenario, the compiler used rate-monotonic scheduling and thus assigned a higher priority to the more frequent task Q .

For processing sensor values, sensor ports can be connected to task input ports; for driving actuators, task output ports can be connected to actuator ports; and for communication between Giotto tasks, output ports of one task can be connected to input ports of another task. Connections are established by drivers. A *Giotto driver* computes a function on its source ports, and passes the result to its destination ports. If the destination ports are actuator ports, we refer to the driver as *actuator driver*; if the destination ports are the input ports of a task, as *task driver*. Every Giotto driver has a *guard*, which is a

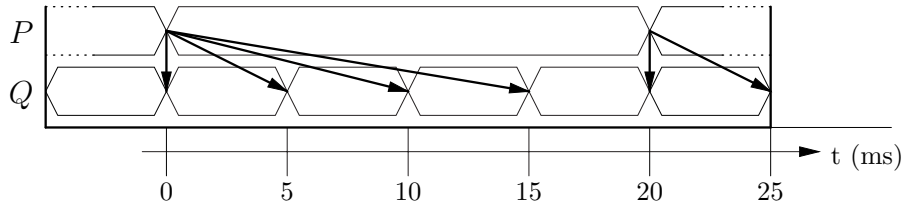


Figure 4: The data flow using a Giotto driver

predicate on its source ports. If the guard does not hold, then the driver is not executed, and in the case of a task driver, neither is the associated system task. There is an important difference between drivers and tasks: drivers represent system-level code for preprocessing and transmitting data between local ports; tasks, on the other hand, perform application-level computation. In Giotto’s programming model, driver execution is logically instantaneous, while task execution takes time up to the length of the task’s period.

Since the result of a task computation is written at the end of the task’s period, task drivers only cause data flow from past task invocations to current invocations, and not between concurrent invocations. Figure 4 shows the data flow of a driver for task Q that reads an output port p of task P . The first four invocations of task Q after the 0ms time instant see the result of the last invocation of task P before the 0ms time instant. No matter when P is finished with its computation after the 0ms time instant, Q sees the result only at the 20ms time instant. The Giotto semantics is deterministic and platform-independent, in the sense that a Giotto program uniquely determines the update rate of every Giotto port, regardless of any differences in implementation or performance. Furthermore, the values passed between tasks and to actuators depend only on the sensor readings, not on the scheduling scheme. In general, a compiler has many different choices to conform with the Giotto semantics: the execution of task P , the driver for Q , and task Q can be scheduled in any way that guarantees the data flow shown in Figure 4.

The tasks P and Q may run on different CPUs of a distributed platform. In this case, the output port p of P needs to be transmitted over the network to the driver of Q . Figure 5 shows one possible timing of the transmission of the output port p . Assuming that P always finishes its computation by 17ms, we may use the remaining 3ms to deliver the result to the driver of Q on time. If 1ms is an upper bound on network transmission, then the compiler may allocate a 1ms time slot somewhere between the 17ms and 20ms time instants for the communication from P to the driver of Q . This ensures that p ’s value is available at Q ’s CPU at the 20ms time instant. If P were to finish before the 15ms time instant, then the compiler might reserve an earlier time slot. However, it would have to buffer the result until after the 15ms time instant in order to guarantee the Giotto semantics. This is because the task Q must not see the new value of p before the 20ms time instant. The compiler is in charge of generating not only a permissible computation schedule but also a permissible communication schedule. We will later see how the programmer can also guide the compiler by giving directives for a specific platform.

So far, we have seen Giotto ports, drivers, and tasks. To allow a Giotto program to react to changes in its environment, we introduce modes. A *Giotto mode* consists of a set

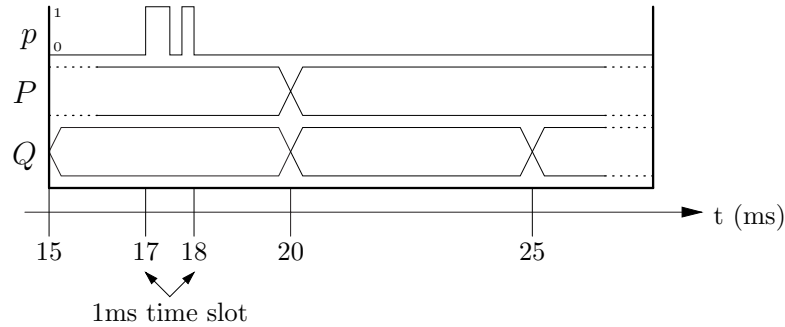


Figure 5: The timing diagram for a transmission from the task output port p

of concurrent Giotto tasks with invocation frequencies (which determine the task periods²) and task drivers, a set of actuator updates, and a set of mode switches. An *actuator update* has a frequency and an actuator driver, which is executed with the given frequency to update the actuator values. A Giotto mode fully describes the behavior and timing of a control system at a particular point in time. A *Giotto program* is a set of Giotto modes. A *mode switch*, when enabled, causes the program instantaneously to switch from one Giotto mode to another. The mode switch has a frequency and a *mode driver*, whose guard is evaluated with the given frequency. When the guard evaluates to true, the mode switch is enabled and the mode driver is executed. The source ports of the mode driver are task output ports of the current mode, and the destination ports are task output ports of the next mode. In this way, mode drivers can pass values from one Giotto mode to the next. To guarantee determinism, we require that, for every Giotto mode, the conjunction of any two mode driver guards is unsatisfiable; that is, at most one mode switch can be enabled at any given time.

In Giotto, a task is considered a unit of work, which, once started, must be allowed to complete. A mode switch may cease the periodic invocation of a task if that task's period ends at the time the mode switch guard is evaluated. However, a mode switch may not terminate any task whose period has not ended. If a task may be running when a mode switch occurs, the Giotto semantics requires that the next mode again contains this task. The least common multiple of all task invocation periods, actuator update periods, and mode switch periods of a Giotto mode determines the *period* of the mode. The execution of a Giotto mode for a single period is called a *round*. When a mode switch occurs in the middle of a round, first the current mode is terminated instantaneously. If t is the time until the periods of all current task invocations end, then the next mode is entered t milli-seconds before the start of a new round. This ensures that as little time as necessary elapses before the full functionality of the new mode begins.

Suppose we are given a Giotto mode M containing the Giotto tasks P and Q invoked with 20ms and 5ms periods, respectively, and a Giotto mode N containing the Giotto tasks P and R invoked with 20ms and 2.5ms period, respectively. Suppose there is a mode switch s from M to N with a 5ms period. Then M and N have the same period of 20ms.

²For an event with frequency f , the period is $1/f$, and vice versa.

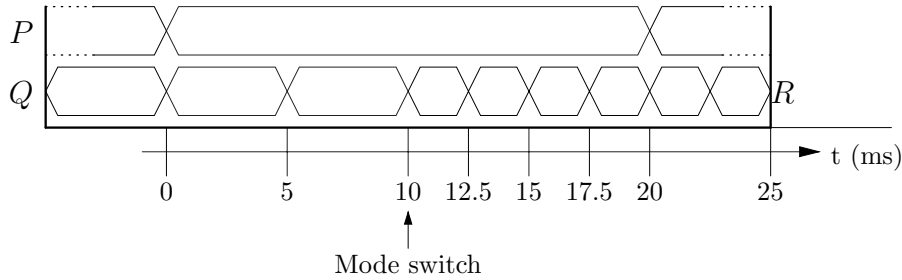


Figure 6: The timing diagram for the Giotto mode switch s

Figure 6 shows the timing diagram of the mode switch s enabled at the 10ms time instant in the middle of a round of M . Since both modes M and N contain the task P , it is not terminated but can continue its computation as if nothing happened. However, Q 's invocations are replaced by two times as many invocations of the task R . Since N 's round has already been completed half-way at the 10ms time instant, there will be only four invocations of R before a new round of N starts at the 20ms time instant.

3 A Distributed Hard Real-time Control Problem

As an example of a distributed hard real-time control problem consider a set of n robots. Each robot has a CPU, two motors, and a touch sensor. The motors drive wheels and allow the robot to move forward and backward, and to rotate. The touch sensor is connected to a bumper. The n robots share a broadcast communication medium.

Figure 7 shows the behavior of the n robot system, where a circle depicts the state of a robot and an arc is a transition from one state to another. Note that here state is a behavioral concept rather than, say, a Giotto mode. A robot that is either in the lead or evade state is called a *leader*. A robot that is either in the follow or stop state is called a *follower*. We require that at all times there is only a single robot that is a leader, while the $n - 1$ remaining robots are followers. Upon initialization the leader robot (which is chosen at random) is in the lead state and determines the movements taken by all n robots. For simplicity, the leader tells everyone to move in the same way, resulting in a synchronized "dance." The $n - 1$ followers are in the follow state and listen to the commands of the leader.

Now, there are two possible scenarios. Either the leader's bumper or the bumper of one of the followers is pushed, presumably by an obstacle. Again for simplicity, we assume that no more than a single bumper can be 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 short amount of time, to avoid the obstacle, whereas a robot in the stop state simply stops. When the leader is finished with the evasion procedure it goes into the lead state, while the $n - 1$ followers go 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. Pushing a bumper of a follower makes this

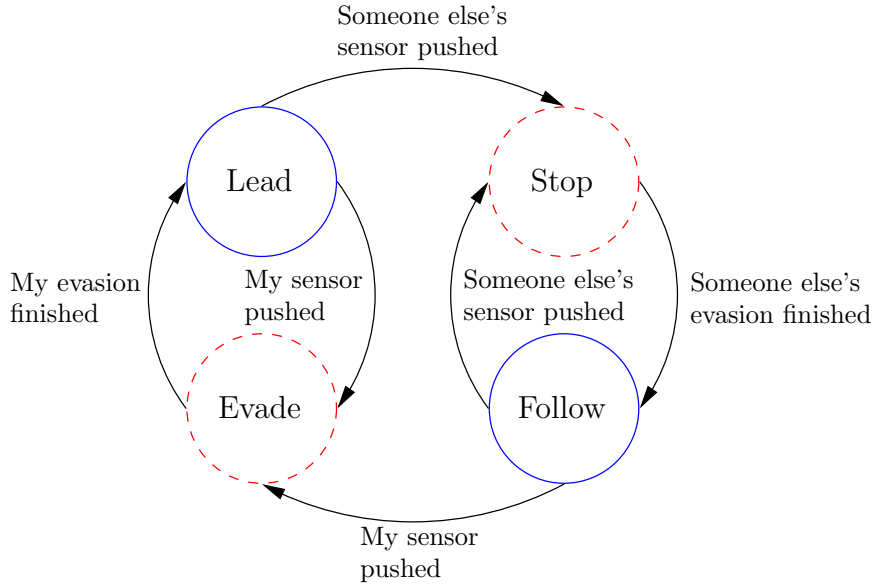


Figure 7: The behavior of an n robot system

robot the new leader. This concludes the description of the n robot system. In the next section, we describe a Giotto implementation.

4 A Giotto Program

In order to demonstrate Giotto’s applicability for distributed and heterogeneous platforms, we implemented a Giotto program for coordinating five robots in the way described above. Two of the robots feature a credit card form-factor single-board computer with an Intel 80486 processor and a Lucent WaveLAN wireless Ethernet card. The single board computers run Wind River’s VxWorks RTOS. Both robots use Lego Mindstorms motors and touch sensors. The three other robots are pure Lego Mindstorms robots equipped with Hitachi microcontrollers and infrared transceivers. The microcontrollers run Lego’s original firmware. Communication between the different platforms is established through a gateway—a notebook PC—between wireless Ethernet and the infrared link.

For the sake of simplicity, we describe the Giotto program for a two-robot system. The appendix contains the Giotto program. In this section, any program code in brackets can be disregarded; it belongs to the annotated version of Giotto. We will discuss annotated Giotto in the next section. The Giotto program begins with the global declaration of ports. The sensor port `sensorX` of robot `X` contains `true` whenever the bumper of robot `X` is pushed. These ports use the method `c_sensorget`, written in the host programming language, to read the status of the bumper device. We call `c_sensorget` a *device driver*. A device driver like `c_sensorget` implements the association of a sensor port to a device. Every time a sensor port is accessed by a task or mode driver, its associated device driver is called. The implementation of the device driver may, e.g., poll the device or read the latest value of an asynchronous update. Similarly to sensor ports, the actuator ports `motorLX` and `motorRX`

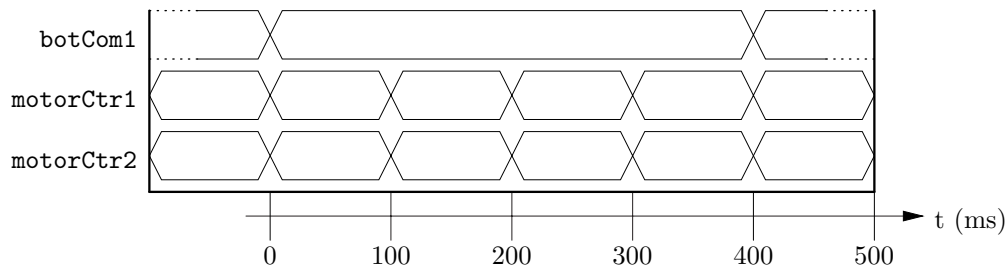


Figure 8: The timing diagram for a round of the Lead1Follow mode

are associated to the left and right motor of robot X using the device drivers `c_motorLput` and `c_motorRput`, respectively. The remaining ports —`command`, `motor1`, `motor2`, and `finished`— are task output ports, which are not associated to any devices.

The port declarations in the Giotto program are followed by task and driver declarations. Consider, for example, the task declaration `motorCtr1`. This task has a single input port called `com`, which is a formal parameter, and a single output port, which is the globally declared output port `motor1`. In general, the input ports are declared by formal parameters with local scope whereas the output ports must be globally declared output ports. In the body of the declaration, we specify the name `c_botCom1` of the procedure that implements the task. We use ANSI C to implement all tasks and drivers (not shown). Now, consider the driver declaration `stop1Drv`. This driver has a single source port, which is the globally declared sensor port `sensor1`, and two destination ports given by the formal parameters `who` and `com`. In general, the source ports must be globally declared sensor or task output ports whereas the destination ports are declared by formal parameters with local scope. In the body of the declaration, we specify the name `c_true_port` of the procedure that implements the guard of the driver, and the name `c_stop1Drv` of the procedure that implements the actual driver.

Finally, the Giotto program contains six mode declarations. The `Lead1Follow` mode is the start mode. Recall that each Giotto mode describes the behavior of the whole system of CPUs and networks. Since one robot is in the lead or evade state and the rest are in the follow or stop state, we use a `LeadXFollow` mode and an `EvadeXStop` mode for each leader X . To improve responsiveness of the implementation we also introduce for each robot X a Giotto mode `StopX`, which allows the robots to stop quickly. In general, for n robots we obtain $3n$ modes.

All modes run with a period of 400ms. Consider the `Lead1Follow` mode in which robot 1 is the leader. Actuator updates are indicated by the keyword `actfreq`; mode switches, by `exitfreq`; and task invocations, by `taskfreq`. The `botCom1` task runs once per round and computes a command stored in the output port `command`. There are two more Giotto tasks `motorCtr1` and `motorCtr2` running with a period of 100ms four times per round. The two tasks control the motors of both robots according to the command in `command`. The task driver `motorDrv` delivers the value in `command` to the input ports of the tasks `motorCtr1` and `motorCtr2`. The guard `motorDrv` is always `true`, so the tasks `motorCtr1` and `motorCtr2` are always executed. The higher frequency of these tasks allows for smoother control of

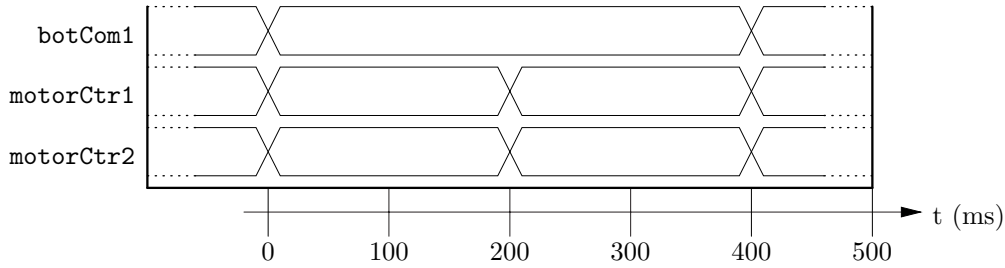


Figure 9: The timing diagram for a round of the `Stop1` mode

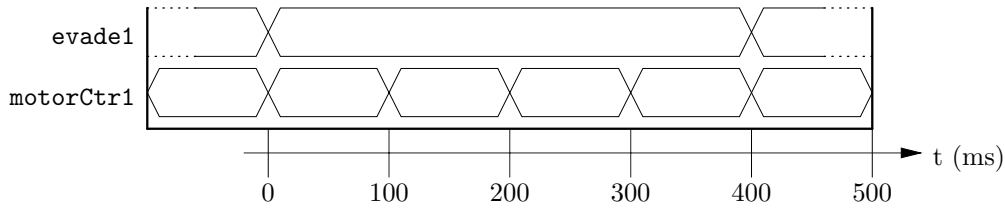


Figure 10: The timing diagram for a round of the `Evade1Stop` mode

the motors. For example, the left motor of robot 1 is controlled by the actuator port `motorL1`, which is updated by the actuator driver `motorActL1` at the end of every period of the task `motorCtr1`. The actuator driver `motorActL1` is invoked at the same frequency as `motorCtr1`; it extracts the necessary information for the left motor from the output port `motor1`. Figure 8 shows the timing diagram for one round in the `Lead1Follow` mode.

The state of the touch sensors is checked twice every round by the two mode switches. The mode switches employ the mode drivers `stop1Drv` and `stop2Drv`, which perform the work of checking the sensors of robot 1 and 2, respectively. We assume that both bumpers cannot be pressed simultaneously. If the bumper of robot 1 is pushed, we switch to the `Stop1` mode, in which both robots stop driving. Figure 9 shows the timing diagram for one round of the `Stop1` mode. After completing one round of the `Stop1` mode the system proceeds to the `Evade1Stop` mode, in which robot 1 performs an evasion procedure and robot 2 does nothing. Similarly, if the bumper of robot 2 is pushed and the bumper of robot 1 is not, we switch to the `Evade2Stop` mode via one round in the `Stop1` mode.

In the `Evade1Stop` mode, the `evade1` task computes once per round the next evasion step (stored in the output port `command`), and whether the evasion maneuver is finished or not (stored in the output port `finished`). Figure 10 shows the timing diagram for the `Evade1Stop` mode. There is also a Giotto task `motorCtr1` running with a period of 100ms four times per round, which controls the motors of robot 1 according to the evasion steps in `command`. Once `finished` contains `true`, we switch to the `Lead1Follow` mode.

Figure 11 shows the timing diagram for two mode switches from the `Lead1Follow` mode to the `Stop1` mode and then to the `Evade1Stop` mode. The mode switch to the `Stop1` mode happens in the middle of the round of the `Lead1Follow` mode at the 200ms time

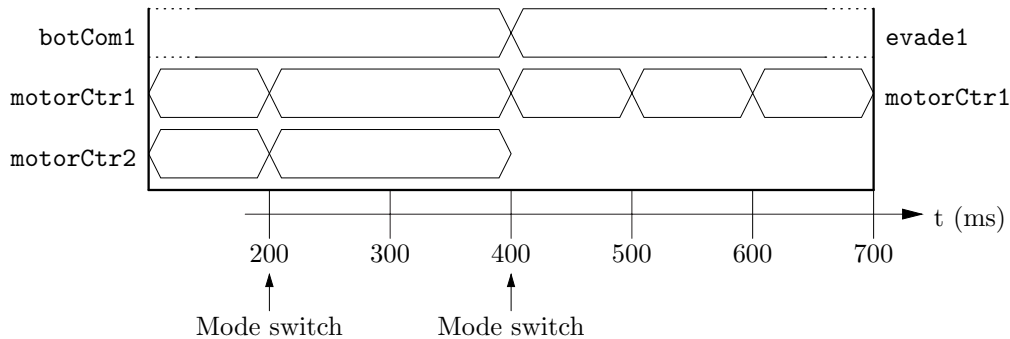


Figure 11: The timing diagram for mode switches to the `Stop1` mode and to the `Evade1Stop` mode

instant. Then the robots stop 200ms later at the 400ms time instant after both motor control tasks have been invoked once. At the 400ms time instant, another mode switch is performed to the `Evade1Stop` mode, which performs an evasion maneuver with robot 1. The implementation of the `Lead2Follow`, `Stop2`, and `Evade2Stop` modes, in which robot 2 is the leader, works similarly. This completes the Giotto program implementing a two-robot system. In the next section, we discuss the compilation of the Giotto program for a given platform.

5 Semi-automatic Compilation with Annotated Giotto

In the previous section, we discussed the platform-independent aspects of a Giotto program which implements the two-robot system. For a non-distributed platform, this level of detail is sufficient to allow the Giotto compiler to generate code that guarantees the timing requirements of Giotto. However, code generation for distributed platforms is more complex and may require user interaction.

Often the programmer intends a particular mapping of Giotto tasks to hosts (CPUs), as well as a mapping of Giotto ports to networks. A reasonable mapping for the two-robot example may allocate the `botComX`, `evadeX`, and `motorCtrX` tasks to robot X , for $X=1,2$. The `motorCtrX` task should get the highest priority because of its shortest deadline. Also, a reasonable mapping may allocate the mode switches of the modes `LeadXFollow`, `StopX`, and `EvadeXStop` to robot X . Consequently, the values in `com`, `sensorX`, and the next mode have to be communicated between the two robots.

Starting with a pure Giotto program, which contains no platform-related information, the programmer may use Giotto annotations to provide the Giotto compiler with details about the target platform. A *Giotto annotation* falls into one of three possible categories: (1) a *Giotto-H annotation* (H for “hardware”) specifies a set of hosts, a set of networks, and WCET information; (2) a *Giotto-M annotation* (M for “map”) specifies for a Giotto mode the Giotto task-to-host and communication-to-network mappings; (3) a *Giotto-S annotation* (S for “schedule”) specifies scheduling information for each host and network. In our exam-

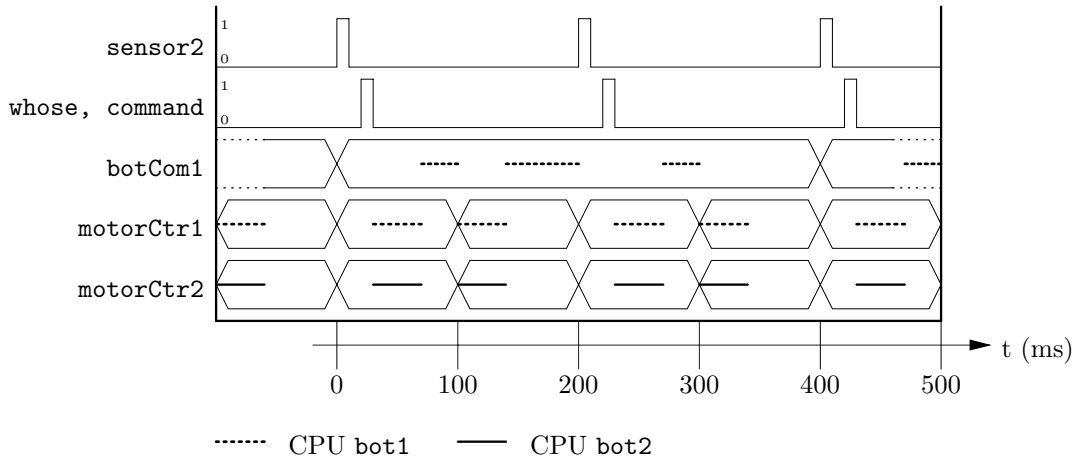


Figure 12: The timing diagram for a round of the `Lead1Follow` mode with scheduling details

ple, the Giotto-S annotations will specify a priority for each task and a time slot for each communication. This style of Giotto-S annotations is suited for static priority RTOS hosts and TDMA networks. For targeting platforms with different scheduling primitives, different annotations can be developed. For example, for platforms consisting of non-preemptive RTOS hosts and CAN networks, Giotto-S annotations would specify time slots for tasks and priorities for communications.

The Giotto program of in the appendix contains examples of all three types of Giotto annotations. A Giotto-H annotation at the top of the program provides details on the two-robot platform. There are two hosts called `bot1` and `bot2`, representing the two robot CPUs, which are connected by a network called `net12`. Specific mapping and scheduling information is given by the Giotto-M and Giotto-S annotations in the example. Note that for flexibility we use symbolic names rather than numbers for task priorities and communication time slots.

For instance, in the `Lead1Follow` mode, the `botCom1` task is assigned to `bot1` with the priority `p1`, which is lower than the priority `p0` of the `motorCtr1` task assigned to `bot1` as well. Consider Figure 12, which shows the timing diagram for a round of the `Lead1Follow` mode with scheduling details. The dotted line shows which Giotto task is running on `bot1`. The lower priority `botCom1` task gets the CPU only when the `motorCtr1` task is finished. In order to allow `bot1` to evaluate the mode switch guards, the value of the `sensor2` port has to be transmitted from `bot2`. This communication from `bot2` to `bot1` has to occur twice per round because of the mode switch period of 200ms. Since the sensors are sampled at the 0ms and 200ms time instant, the value of `sensor2` cannot be transmitted early. The `push` annotation indicates that the output host `bot2` initiates the communication. The dual `pull` annotation is also available, for example, to support less capable distributed sensors.

The signal at the `sensor2` port in Figure 12 shows the timing for the transmission of the `sensor2` values to `bot1`. We assume that communication consumes CPU cycles. The 0ms to

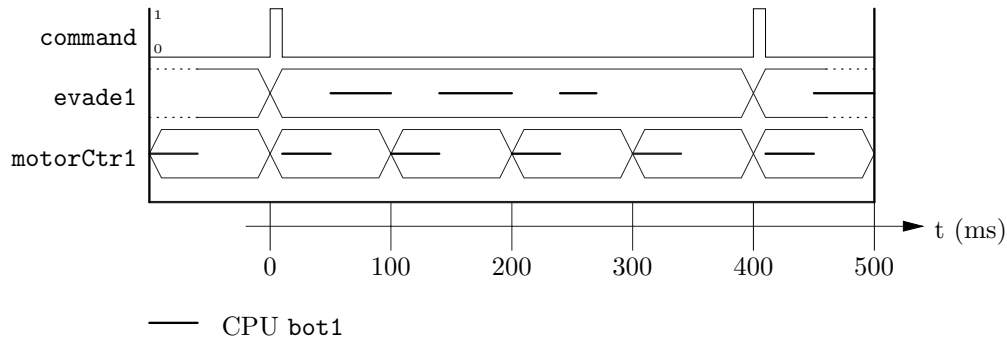


Figure 13: The timing diagram for a round of the `Evade1Stop` mode with scheduling details

10ms time slot for the `sensor2` value transmission delays the decision of performing a mode switch. Such delays do not affect the Giotto semantics as long as all Giotto tasks still meet their deadlines. The mode switch decision is taken by `bot1` between 10ms and 20ms. Then, between 20ms and 30ms, `bot1` sends its decision to `bot2`. The communication between 20ms and 30ms from `bot1` to `bot2` also transmits the value of the output port `command` computed in the previous round, and the port `whose`, which indicates whose bumper was pressed.

Figure 13 shows the timing for a round of the `Evade1Stop` mode with scheduling details. In this mode, `command` has to be transmitted from `bot1` to `bot2` once per round, as shown by the top-most line between the 0ms and 10ms time instants. The timing and scheduling details for two mode switches from the `Lead1Follow` to the `Stop1` mode and then to the `Evade1Stop` mode are shown in Figure 14. Although the mode switch to the `Stop1` mode happens logically at the 200ms time instant, it is actually performed 30ms later by starting the `motorCtrlX` tasks of the `Stop1` mode rather than of the `Lead1Follow` mode.

With annotated Giotto, the programmer is able to give directives to the Giotto compiler on how to map Giotto tasks and Giotto ports to a given platform of hosts and networks. Giotto-S annotations allow guidance on how to schedule computation and communication resources. Most importantly, Giotto annotations refine a given non-annotated Giotto program without affecting the functionality and timing specification. In this way, Giotto separates compilation from the use of a particular scheduling or communication scheme: if incomplete directives are given, then different compilers may use different scheduling schemes. Indeed, one compiler, using one particular scheduling scheme, may fail, whereas another, “smarter” compiler may succeed in compiling a given Giotto program on a given distributed platform.

6 Summary and Related Work

We have presented a tool-supported development methodology for embedded control software which is based on the programming language Giotto. In Giotto, the programmer specifies the functionality and timing of a control design, leaving the specification of scheduling

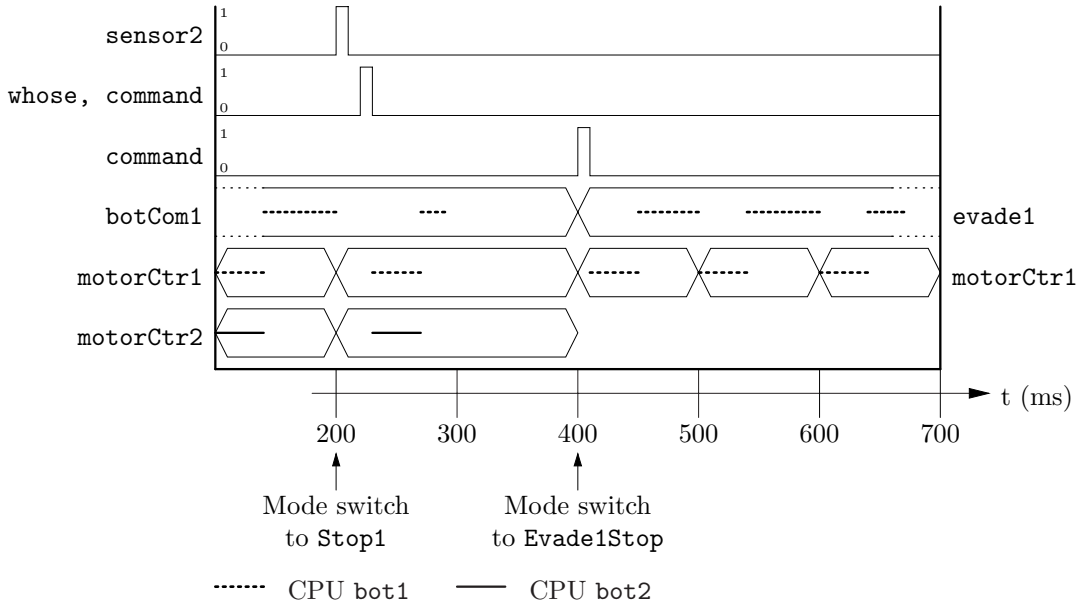


Figure 14: The timing diagram for mode switches from the `Lead1Follow` mode to the `Stop1` mode and to the `Evade1Stop` mode with scheduling details

schemes to the Giotto compiler. The Giotto compiler automates the implementation of embedded control systems, by taking over the tedious and error-prone task of producing computation and communication schedules. Given a Giotto program and a particular platform, the Giotto compiler may (or may not) be able to generate Giotto executables that obey the timing requirements of the program. When targeting complex distributed platforms, the programmer may also give explicit scheduling directives to the compiler using Giotto annotations. Giotto has a time-triggered semantics. Task invocations as well as observations of the environment in a Giotto system are triggered by the tick of a notional global clock. Consequently, the timing behavior of a Giotto system is highly predictable, which makes Giotto particularly well-suited for safety-critical applications with hard real-time constraints.

We have implemented a compiler for fully annotated Giotto, with drivers and tasks that are given as C functions, as well as a runtime library for Wind River’s VxWorks RTOS. The Giotto executables are generated as C source code that is compiled and linked against the runtime library. In the near future we hope to develop a Giotto compiler that makes scheduling decisions, rather than relying on full annotations, and we hope to develop runtime systems for additional platforms.

Many of the individual elements of Giotto are derived from the literature. However, we believe that the use of time-triggered task invocation plus time-triggered mode switching for platform-independent real-time programming is novel. Giotto is similar to architecture description languages (ADLs) [Cle96]. 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. ADLs also allow the compilation of scheduling code to connect tasks written in conventional programming languages. The design methodology for the MARS system, a precursor of the TTA, similarly distinguishes *programming-in-the-large* from *programming-in-the-small* [KZF⁺91]. Giotto’s inter-task communication semantics is particularly similar to the MetaH language [Ves97], which is designed for real-time, distributed avionics applications. MetaH supports periodic tasks, multi-modal control, and distributed implementations. Giotto can be viewed as capturing a time-triggered fragment of MetaH in an abstract and formal way. Unlike MetaH, the Giotto abstraction does not constrain the implementation to a particular scheduling paradigm as long as the timing requirements of a Giotto program are guaranteed. Since the semantics of Giotto is defined formally, the behavioral properties of a Giotto program may be subject to formal verification [Hen00].

The goal of Giotto —to provide a platform-independent programming abstraction for real-time systems— is shared also by the family of synchronous reactive programming languages [Hal93], such as Esterel [Ber00], Lustre [HCRP91], and Signal [BGJ91]. While the synchronous reactive languages are designed around zero-delay computation, Giotto is based on the formally weaker notion of unit-delay computation, because the execution of a Giotto task has a positive duration. This avoids the complications involved with fixed-point semantics and shifts the emphasis to code generation under WCET constraints. Giotto can be seen as identifying a class of synchronous reactive programs that support both typical real-time control applications and distributed code generation.

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 help with a Ptolemy II [DGH⁺99] implementation of Giotto.

Appendix: A Giotto Program with Annotations

```
[host bot1 address 192.168.0.1 priorities p0 > p1;
host bot2 address 192.168.0.2 priorities q0 > q1;
network n12 address 192.168.0.0 connects bot1, bot2]

// Sensor ports

sensor

// True means pushed
c_bool sensor1 uses c_sensorget [host bot1]; // bot1 touch sensor
c_bool sensor2 uses c_sensorget [host bot2]; // bot2 touch sensor

// Actuator ports

actuator

c_int motorL1 uses c_motorLput [host bot1]; // bot1 left motor
c_int motorR1 uses c_motorRput [host bot1]; // bot1 right motor
c_int motorL2 uses c_motorLput [host bot2]; // bot2 left motor
c_int motorR2 uses c_motorRput [host bot2]; // bot2 right motor

// Output ports

output

// command port
c_int command := c_zero;

// bot1 motor control port
c_int motor1 := c_zero;

// bot2 motor control port
c_int motor2 := c_zero;

// Evade maneuver finished
c_bool finished := false;

// Task declarations

task botCom1() output (command) {
  schedule c_botCom1(command)
}

task botCom2() output (command) {
  schedule c_botCom2(command)
}

task motorCtr1(c_int com) output (motor1) {
  schedule c_motorCtr(com, motor1)
}

task motorCtr2(c_int com) output (motor2) {
  schedule c_motorCtr(com, motor2)
}

task evade1() output (command, finished) {
```

```

    schedule c_evade1(command, finished)
}

task evade2() output (command, finished) {
    schedule c_evade2(command, finished)
}

// Driver declarations

driver motorDrv(command) output (c_int com) {
    if c_true() then c_motorDrv(command, com)
}

driver motorActL1(motor1) output (c_int mot) {
    if c_true() then c_motorActL(motor1, mot)
}

driver motorActR1(motor1) output (c_int mot) {
    if c_true() then c_motorActR(motor1, mot)
}

driver motorActL2(motor2) output (c_int mot) {
    if c_true() then c_motorActL(motor2, mot)
}

driver motorActR2(motor2) output (c_int mot) {
    if c_true() then c_motorActR(motor2, mot)
}

driver stop1Drv(sensor1) output (c_int who, c_int com) {
    if c_true_port(sensor1) then c_stop1Drv(who, com)
}

driver stop2Drv(sensor2) output (c_int who, c_int com) {
    if c_true_port(sensor2) then c_stop2Drv(who, com)
}

driver motorStopDrv() output (c_int com) {
    if c_true() then c_motorStopDrv(com)
}

driver evade1StopDrv(whose) output (c_int com) {
    if c_evade1StopGrd(whose) then c_evadeStopDrv(com)
}

driver evade2StopDrv(whose) output (c_int com) {
    if c_evade2StopGrd(whose) then c_evadeStopDrv(com)
}

driver leadFollowDrv(finished) output (c_int com) {
    if c_true_port(finished) then c_leadFollowDrv(com)
}

start Lead1Follow(c_zero) {
    mode Lead1Follow(command) period 400
    [network n12 slots s0 (0,10), s1 (20,30),
     s2 (200,210), s3 (220,230)] {

        actfreq 4 do motorL1(motorActL1);
        actfreq 4 do motorR1(motorActR1);
        actfreq 4 do motorL2(motorActL2);
        actfreq 4 do motorR2(motorActR2);

        [push (sensor2) from bot2 to (bot1) in n12 slots s0,s2]

        exitfreq 2 do Stop1(stop1Drv)
        [host bot1; push (whose,command) to (bot2) in n12 slots s1,s3];
        exitfreq 2 do Stop2(stop2Drv)
        [host bot1; push (whose,command) to (bot2) in n12 slots s1,s3];

        taskfreq 1 do botCom1() [host bot1 priority p1];
        taskfreq 4 do motorCtr1(motorDrv) [host bot1 priority p0];
        taskfreq 4 do motorCtr2(motorDrv) [host bot1 priority q0];
    }

    mode Stop1(whose, command) period 400
    [network n12 slots s0 (0,10)] {

        actfreq 2 do motorL1(motorActL1);
        actfreq 2 do motorR1(motorActR1);
        actfreq 2 do motorL2(motorActL2);
        actfreq 2 do motorR2(motorActR2);

        exitfreq 1 do Evade1Stop(evade1StopDrv)
        [host bot2; push (command) to (bot1) in n12 slots s0];
        exitfreq 1 do Evade2Stop(evade2StopDrv)
        [host bot2; push (command) to (bot1) in n12 slots s0];

        taskfreq 1 do botCom2() [host bot2 priority q1];
        taskfreq 2 do motorCtr1(motorStopDrv) [host bot1 priority p0];
        taskfreq 2 do motorCtr2(motorStopDrv) [host bot2 priority q0];
    }

    mode Evade2Stop(command) period 400
    [network n12 slots s0 (0,10)] {

        actfreq 4 do motorL2(motorActL2);
        actfreq 4 do motorR2(motorActR2);

        exitfreq 1 do Lead2Follow(leadFollowDrv)
        [host bot2; push (command) to (bot1) in n12 slots s0];

        taskfreq 1 do evade2() [host bot2 priority q1];
        taskfreq 4 do motorCtr2(motorDrv) [host bot2 priority q0];
    }
}

actfreq 2 do motorL1(motorActL1);
actfreq 2 do motorL2(motorActL2);
actfreq 2 do motorR2(motorActR2);

exitfreq 1 do Evade1Stop(evade1StopDrv)
[host bot1; push (command) to (bot2) in n12 slots s0];
exitfreq 1 do Evade2Stop(evade2StopDrv)
[host bot1; push (command) to (bot2) in n12 slots s0];

taskfreq 1 do botCom1() [host bot1 priority p1];
taskfreq 2 do motorCtr1(motorStopDrv) [host bot1 priority p0];
taskfreq 2 do motorCtr2(motorStopDrv) [host bot2 priority q0];
}

mode Evade1Stop(command) period 400
[network n12 slots s0 (0,10)] {

    actfreq 4 do motorL1(motorActL1);
    actfreq 4 do motorR1(motorActR1);

    exitfreq 1 do Lead1Follow(leadFollowDrv)
    [host bot1; push (command) to (bot2) in n12 slots s0];

    taskfreq 1 do evade1() [host bot1 priority p1];
    taskfreq 4 do motorCtr1(motorDrv) [host bot1 priority p0];
}

mode Lead2Follow(command) period 400
[network n12 slots s0 (0,10), s1 (20,30),
 s2 (200,210), s3 (220,230)] {

    actfreq 4 do motorL1(motorActL1);
    actfreq 4 do motorR1(motorActR1);
    actfreq 4 do motorL2(motorActL2);
    actfreq 4 do motorR2(motorActR2);

    [push (sensor1) from bot1 to (bot2) in n12 slots s0,s2]

    exitfreq 2 do Stop1(stop1Drv)
    [host bot2; push (whose,command) to (bot1) in n12 slots s1,s3];
    exitfreq 2 do Stop2(stop2Drv)
    [host bot2; push (whose,command) to (bot1) in n12 slots s1,s3];

    taskfreq 1 do botCom2() [host bot2 priority q1];
    taskfreq 4 do motorCtr1(motorDrv) [host bot1 priority p0];
    taskfreq 4 do motorCtr2(motorDrv) [host bot2 priority q0];
}

mode Stop2(whose, command) period 400
[network n12 slots s0 (0,10)] {

    actfreq 2 do motorL1(motorActL1);
    actfreq 2 do motorR1(motorActR1);
    actfreq 2 do motorL2(motorActL2);
    actfreq 2 do motorR2(motorActR2);

    exitfreq 1 do Evade1Stop(evade1StopDrv)
    [host bot2; push (command) to (bot1) in n12 slots s0];
    exitfreq 1 do Evade2Stop(evade2StopDrv)
    [host bot2; push (command) to (bot1) in n12 slots s0];

    taskfreq 1 do botCom2() [host bot2 priority q1];
    taskfreq 2 do motorCtr1(motorStopDrv) [host bot1 priority p0];
    taskfreq 2 do motorCtr2(motorStopDrv) [host bot2 priority q0];
}

mode Evade2Stop(command) period 400
[network n12 slots s0 (0,10)] {

    actfreq 4 do motorL2(motorActL2);
    actfreq 4 do motorR2(motorActR2);

    exitfreq 1 do Lead2Follow(leadFollowDrv)
    [host bot2; push (command) to (bot1) in n12 slots s0];

    taskfreq 1 do evade2() [host bot2 priority q1];
    taskfreq 4 do motorCtr2(motorDrv) [host bot2 priority q0];
}
}

```

References

- [Ber00] G. Berry. The foundations of Esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 425–454. MIT Press, 2000.
- [BGJ91] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: The Signal language and its semantics. *Science of Computer Programming*, 16(2):103–149, 1991.
- [Cle96] P. Clements. A survey of architecture description languages. In *Proceedings of the 8th International Workshop on Software Specification and Design*, pages 16–25. IEEE Computer Society Press, 1996.
- [DGH⁺99] J. Davis, M. Goel, C. Hylands, B. Kienhuis, E.A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, and Y. Xiong. *Ptolemy II: Heterogeneous Concurrent Modeling and Design in Java*. Technical Report UCB/ERL-M99/44, University of California, Berkeley, 1999.
- [FMD⁺00] T. Führer, B. Müller, W. Dieterle, F. Hartwich, and R. Hugel. Time-triggered communication on CAN (TTCAN). In *Proceedings of the 7th International CAN Conference*, 2000.
- [Hal93] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer, 1993.
- [HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.
- [Hen00] T.A. Henzinger. Masaccio: A formal model for embedded components. In *Proceedings of the First IFIP International Conference on Theoretical Computer Science*, LNCS 1872, pages 549–563. Springer, 2000.
- [HHK01] T.A. Henzinger, B. Horowitz, and C.M. Kirsch. Giotto: A time-triggered language for embedded programming. In *Proceedings of the First International Workshop on Embedded Software*, LNCS 2211, pages 175–194. Springer, 2001.
- [Kop97] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer, 1997.
- [KZF⁺91] H. Kopetz, R. Zainlinger, G. Fohler, H. Kantz, P. Puschner, and W. Schütz. The design of real-time systems: From specification to implementation and verification. *IEE/BCS Software Engineering Journal*, 6(3):72–82, 1991.
- [LM95] Y.T.S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. *ACM SIGPLAN Notices*, 30(11):88–98, 1995.
- [TFW00] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, 18(2/3):157–179, 2000.
- [Ves97] S. Vestal. MetaH support for real-time multi-processor avionics. In *Proceedings of the 5th International Workshop on Parallel and Distributed Real-Time Systems*, pages 11–21. IEEE Computer Society Press, 1997.