# Embedded Control Systems Development with Giotto

Thomas A. Henzinger Benjamin Horowitz Christoph Meyer Kirsch
{tah,bhorowit,cm}@eecs.berkeley.edu
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA 94720-1770

## 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 real-time control systems. Giotto consists of a programming language with a formal semantics, and a retargetable compiler and 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.

## Categories and Subject Descriptors

D.2.2 [**Software**]: Software Engineering—*Design Tools and Techniques, Software Architectures*; D.2.3 [**Software**]: Programming Languages—*Language Constructs and Features*

## General Terms

Design, Languages

## 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. On the other hand, implementing control designs is a 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 and operating system configuration. Platforms, which may be distributed, consist of sensors, actuators, hosts, and networks. Platform-independent issues include application 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*, computational activity is triggered by the tick of a notional global clock. A time-triggered system samples its environment and coordinates communication with respect to such a clock (as implemented by, say, a clock synchronization algorithm). Time-triggered systems contrast with *event-triggered systems*, in which computational activity is triggered by events. The time-triggered architecture (TTA) [9] is an important hardware and protocol realization of the time-triggered paradigm. The TTA has recently gained momentum in safety-critical automotive applications, where timing predictability is essential. Giotto offers the predictability of time-triggered systems, but at a higher and platform-independent level. Thus, Giotto enables the specification and automated generation of timing-predictable code for multiple, even distributed, platforms.

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. Each mode switch of $M$ has a real-time frequency, a predicate that is evaluated at this frequency, and a target mode, say, $N$: if the predicate 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

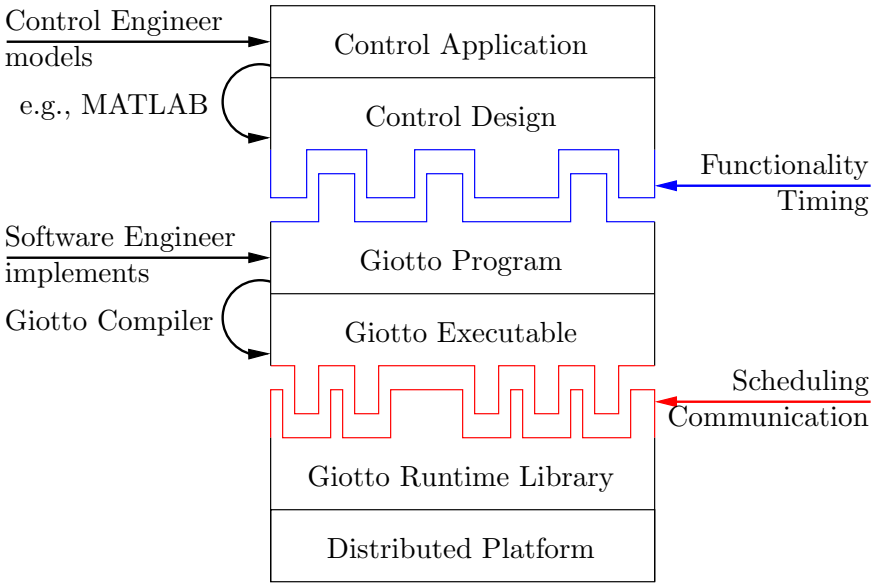


**Figure 1: Embedded control systems development with Giotto**

and actuators. A Giotto program, therefore, 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) or scheduling algorithm. Moreover, Giotto's simplicity allows us to study schedule synthesis and code generation for time-triggered systems.

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 scheduling and communication scheme 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 often difficult, the 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 feasible schedules.

[1]The difficult problem of estimating WCETs is orthogonal to the problems that Giotto addresses. Integer linear programming (ILP) techniques have been proposed to reduce the complexity of WCET prediction [10]. Abstract interpretation can be used to generate integer linear programs for separated cache and path analyses [11].

Figure 1 summarizes the design flow. 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. Most importantly, the Giotto compiler takes over the tedious and error-prone task of generating scheduling directives for computation and communication. In this way, the compiler enables the automation of embedded control systems development. 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 the 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. Because of Giotto's time-triggered semantics, the time-triggered architecture [9] is an interesting target.

Figure 2 shows a more detailed picture of the development flow from a Giotto program to a Giotto executable using Giotto annotations. For non-distributed platforms, the Giotto compiler may be able to automatically generate Giotto executables which obey the timing requirements of the program. However, for distributed platforms, the compiler may be unable to find a feasible schedule. In this case, 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-P annotations specify the platform; Giotto-S annotations map tasks to CPUs and provide constraints for the scheduling of tasks; Giotto-C annotations provide constraints for the scheduling of communication events. It is important to note

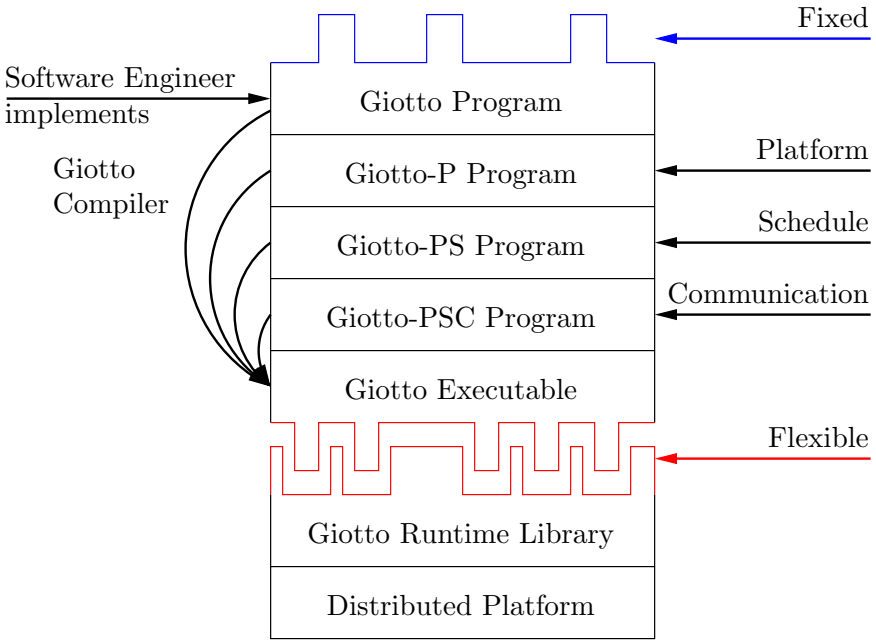


**Figure 2: Automatic compilation with annotated Giotto**

that the original functionality and timing assumptions are not affected by Giotto annotations. Rather, the annotations provide a means of refining a platform-independent Giotto program into an executable for a specific platform.

This paper is a progress report on the development of the Giotto methodology. In the next section we introduce the pure, platform-independent version of the Giotto programming language. Then we illustrate the embedded control system development process with Giotto following an example presented in Section 3. The example asks for 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 syntax and semantics, see the technical report [8].

## 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 hard real-time control system periodically reads sensor information, computes control laws, and writes the results to its actuators. Moreover, such a control system may react to changes in its environment by switching control laws as well as periodicity. The implementation of hard real-time control systems involves complex scheduling problems which, in the case of distributed realizations, become even more advanced.

Giotto's language primitives match the requirements of distributed hard real-time control applications. A *Giotto port* is a physical location in memory, which may be connected to a sensor or actuator, or to an input of a control law. A *Giotto task* is a periodic task that consists of input and output ports as well as some sequential piece of code, with known WCET, written in any programming language. Thus Giotto can be seen as an extension of a standard non-embedded programming language. A Giotto task may also carry state in order to keep track of its past invocations.

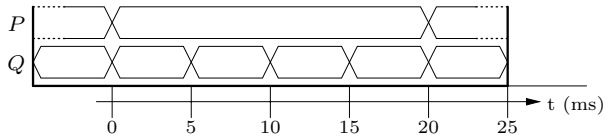


**Figure 3: The timing diagram for the two Giotto tasks $P$ and $Q$**

The Giotto semantics requires that a Giotto task reads the values in its input ports exactly at the time of its invocation and writes its results to its output ports exactly at the end of its period. Consider Figure 3, which depicts the timing diagram of a 20ms Giotto task $P$ and a 5ms Giotto task $Q$. At the 0ms time instant, both $P$ and $Q$ read the values of their input ports and after 5ms, task $Q$ writes its results to its output ports. After 20ms and three more invocations of task $Q$, task $P$ writes its results to its output ports. The Giotto semantics does not specify the physical CPU scheduling of the computation of Giotto tasks. There is only an implicit assumption that the computation of a Giotto task has to be finished within the task's period. Thus it is up to the compiler to use a scheduling mechanism that guarantees the deadlines.

In order to allow data flow between Giotto tasks it is possi-

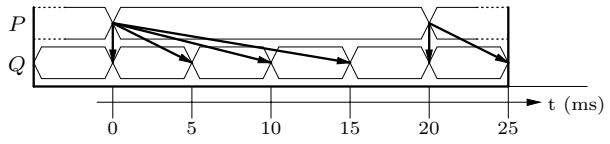


**Figure 4: The data flow over a Giotto connection**

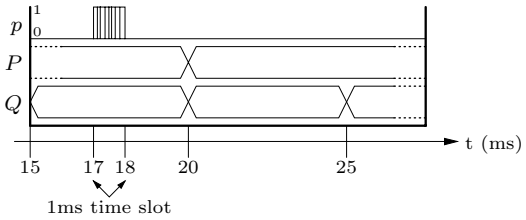


**Figure 5: The timing diagram for a transmission on the output port $p$**

ble to connect input and output ports. A *Giotto connection* associates an input port with an output port. Two ports associated by a Giotto connection are similar to local variables. Since the result of a task's computation is written at the end of the task's period, data may only flow from past task invocations to current invocations, and not between concurrent invocations. Figure 4 shows the data flow over a connection from an output port $p$ of task $P$ to an input port of task $Q$. All four invocations of task $Q$ right after the 0ms time instant will see the result of the last invocation of task $P$ before the 0ms time instant. Independently of when $P$ is finished with its computation after the 0ms time instant, $Q$ will see its result at the 20ms time instant. Giotto's semantical requirements are deterministic and platform-independent, in the sense that these requirements determine the update rate of any Giotto port, regardless of any differences in implementation or performance. An implementation has many different choices to achieve the semantical requirements.

Figure 5 shows a possible timing of the transmission of the output port $p$ which meets Giotto's timing requirements, where $P$ and $Q$ run on different hosts of some distributed platform. Assuming that $P$ will always be finished with its computation within 17ms, we may use the remaining 3ms to deliver the result to the input port of $Q$ on time. In this example, suppose that a 1ms time slot is sufficient to deliver the result of $P$'s computation. Consequently, we may reserve a 1ms time slot somewhere between the 17ms and 20ms time instant to make sure that $p$'s value is available at the 20ms time instant. If $P$ were to finish even earlier, say, before the fourth invocation of $Q$ at the 15ms time instant, then we could deliver its result early but would have to buffer it until after the 15ms time instant in order to guarantee Giotto's semantical requirements. This is because task $Q$ must not see the new value in $p$ before the 20ms time instant. The compiler is in charge of generating not only a suitable computation schedule but also a suitable communication schedule. We will later see how the programmer can guide the compiler with respect to a given platform.

So far, we have seen Giotto ports, connections, and tasks. In order to allow a Giotto implementation to react to changes in its environment we introduce the Giotto mode concept. A *Giotto mode* is a set of concurrent Giotto tasks and Giotto mode switches which fully describes the behavior and timing of a control system. A *Giotto program* is a set of Giotto modes. A *Giotto mode switch*, once enabled, causes the program instantaneously to switch from one Giotto mode to another. The Giotto mode switch has a predicate over ports, which is evaluated periodically. To guarantee determinism, we require that the conjunction of any two mode switch predicates is unsatisfiable. As with Giotto tasks, a Giotto mode switch has a frequency, which determines when to evaluate the mode switch predicate. In Giotto, a task is considered a unit of work, which, once started, must be allowed to complete.[2] A mode switch may cease the periodic invocation of a task if that task finishes at the time the mode switch predicate is evaluated. However, a mode switch may not terminate any running task. For each Giotto mode switch the Giotto semantics requires that the destination mode contains all tasks that may be running when the mode switch occurs. The frequency of the least frequent Giotto task in a Giotto mode determines the *period* of the mode, while the least common multiple of the task and mode switch frequencies determine the *unit* of the mode. The invocations of the Giotto tasks of a Giotto mode within a single period are called a *round*.

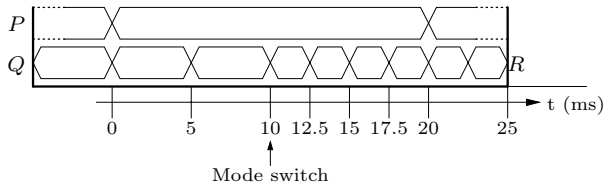


**Figure 6: The timing diagram for the Giotto mode switch $s$**

Suppose we are given a Giotto mode $M$ containing the Giotto tasks $P$ and $Q$ with a 20ms and 5ms period, respectively, and a Giotto mode $N$ containing the Giotto tasks $P$ and $R$ with a 20ms and 2.5ms period, respectively. Then $M$ and $N$ have the same period of 20ms. Suppose there is a Giotto mode switch $s$ from $M$ to $N$ with a 5ms period which conforms to the Giotto requirements. Then $M$ has a unit of 5ms whereas $N$ has a unit of 2.5ms. Figure 6 depicts a timing diagram of the Giotto 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 Giotto task $P$ with a 20ms period, $P$ is not terminated but can finish its computation as if nothing happened. However, $Q$'s invocations are replaced by exactly two times as many invocations of the Giotto task $R$. Since $N$'s round has already been completed half-way at the 10ms time instant, there will be four invocations of $R$ before the end of the round at the 20ms time instant. When a Giotto mode switch occurs within a round, first the current Giotto mode is terminated instantaneously. Then the time $t$ until all currently running tasks finish is

[2]If it is desired that a task end before completion, the work of the task may be divided into a sequence of tasks, with the execution of the members of the sequence occurring only if some condition holds on their input ports.

calculated, and the new mode is entered $t$ seconds before the start of a new period. This ensures that as little time as necessary will elapse before the full functionality of the new mode begins. This functionality includes long-running tasks, and infrequent mode switches.

# 3. A DISTRIBUTED HARD REAL-TIME CONTROL PROBLEM

As an example of a distributed 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 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. 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, 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 robot the new leader. This concludes the behavioral model of the $n$ robot system. In the next section, we will describe a Giotto program that implements the $n$ robot system.

# 4. A GIOTTO PROGRAM

In order to demonstrate Giotto's applicability to distributed and heterogeneous platforms, we implemented a Giotto program for five robots. Two 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 done through a gateway between wireless Ethernet and the infrared link.

```
const int STOP = 0;

// command
int com = STOP;

// mode finished
bool fin = TRUE;

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

int motorL1 = STOP; // robot 1 left motor
int motorR1 = STOP; // robot 1 right motor
int motorL2 = STOP; // robot 2 left motor
int motorR2 = STOP; // robot 2 right motor
```

**Figure 8: The port declarations**

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

start Lead1Follow() {
  mode Lead1Follow() period 400ms entryfreq 1 {
    taskfreq 1 do int com = command1();
      [host bot1 priority p1]
    taskfreq 4 do (int motorL1, int motorR1) =
      motorCtr1(com); [host bot1 priority p0]
    taskfreq 4 do (int motorL2, int motorR2) =
      motorCtr2(com); [host bot2 priority q0]

    exitfreq 2 if (sensor1 && not(sensor2)) then Stop1();
    exitfreq 2 if (sensor2 && not(sensor1)) then Stop2();

    [ net n12 slots s0 (0,20), s1 (20,40),
        s2 (200,220), s3 (220,240), s4(340,360);
      push sensor1 from bot1 to bot2
        in net n12 slots s0, s2;
      push sensor2 from bot2 to bot1
        in net n12 slots s1, s3;
      push com from bot1 to bot2
        in net n12 slots s4; ] }

  mode Stop1() period 400ms entryfreq 2 {
    taskfreq 1 do int com = command1();
      [host bot1 priority p1]
    taskfreq 2 do (int motorL1, int motorR1) =
      motorCtr1(STOP); [host bot1 priority p0]
    taskfreq 2 do (int motorL2, int motorR2) =
      motorCtr2(STOP); [host bot2 priority q0]

    exitfreq 1 if (TRUE) then Evade1Stop(); }

  mode Evade1Stop() period 400ms entryfreq 1 {
    taskfreq 1 do (int com, bool fin) =
      evade1();       [host bot1 priority p1]
    taskfreq 4 do (int motorL1, int motorR1) =
      motorCtr1(com); [host bot1 priority p0]

    exitfreq 1 if (fin) then Lead1Follow();

    [ net n12 slots s0 (340,360);
      push fin from bot1 to bot2
        in net n12 slots s0; ] }

  ...
}
```

**Figure 9: Two-robot Giotto program with annotations**

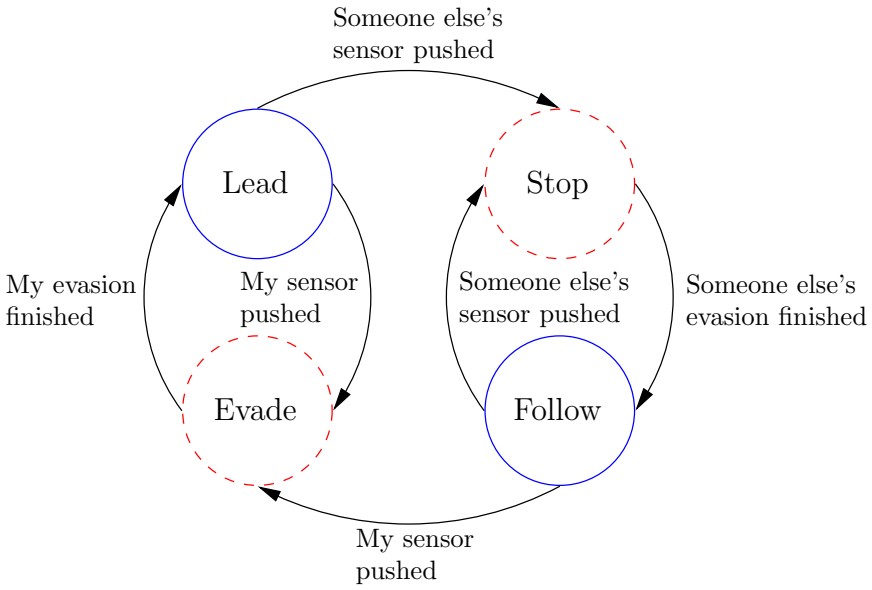


**Figure 7: The behavior of an $n$ robot system**

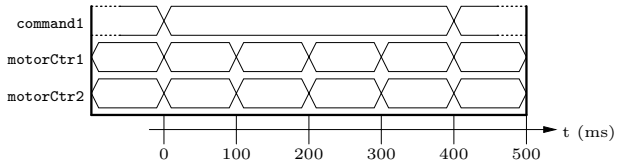


**Figure 10: The timing diagram for a round of the `Lead1Follow` mode**

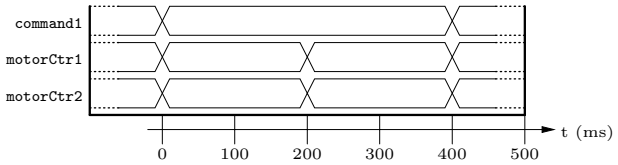


**Figure 11: The timing diagram for a round of the `Stop1` mode**

For the sake of simplicity, we describe the Giotto program for a two-robot system. Figures 8 and 9 depict the Giotto program. Figure 8 shows the declaration and initialization of the required ports. We assume that the port `sensorX` contains `TRUE` whenever the bumper of robot `X` is pushed. The ports `motorLX` and `motorRX` are connected to the left and right motor of robot `X`, respectively.

Figure 9 depicts the Giotto program. It consists of five Giotto modes. The `Lead1Follow` mode is the start mode. 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. Recall that each Giotto mode describes the behavior of the whole system of hosts and nets. Since each robot is either in the lead and follow state or in the evade and stop state, we use a `LeadXFollow` mode and a `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 get $3n$ modes.

All modes run with a period of 400ms with an entry frequency of one, except for the `StopX` modes, which have an entry frequency of two for better performance. Consider the `Lead1Follow` mode in which robot one is the leader. The `command1` task runs once per round and computes a command stored in its only output port `com`. 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 `com`. The higher frequency of these tasks allows for smoother control of the motors. Figure 10 shows the timing diagram for one round in the `Lead1Follow` mode.

The state of the touch sensors is checked twice every round in the three mode switch conditions. If the bumper of robot one is pushed, we switch to the `Stop1` mode in which both robots stop driving. Figure 11 depicts the timing diagram for one round of the `Stop1` mode. After completing one round of the `Stop1` mode the system proceeds to the `Evade1-Stop` mode, in which robot one performs an evasion procedure and robot two does not do anything. Similarly, if the bumper of robot two is pushed 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 `com`, and whether the current mode is finished or not (stored in the output port `fin`). Note that upon entry to the `Evade1Stop` mode `fin` always contains `TRUE`. However, its value will be checked by the mode switch condition no earlier than at the end of

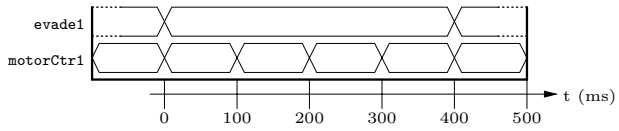


**Figure 12: The timing diagram for a round of the `Evade1Stop` mode**

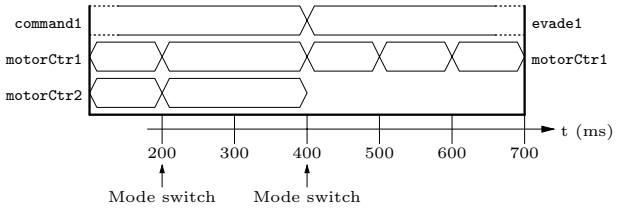


**Figure 13: The timing diagram for Giotto mode switches to the `Stop1` mode and to the `Evade1Stop` mode**

the first round after `evade1` updated `fin`. Figure 12 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 one according to the evasion steps in `com`. Whenever `fin` contains `TRUE` we switch to the `Lead1Follow` mode.

Figure 13 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 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 one. The implementation of the `Lead2Follow`, `Stop2`, and `Evade2Stop` modes, in which robot two is the leader, works similarly as described above. We have introduced a Giotto program implementing a two-robot system. In the next section, we will discuss the compilation of the Giotto program for a given platform.

# 5. AUTOMATIC COMPILATION WITH ANNOTATED GIOTTO

In the previous section, we have discussed the platform-independent aspects of the 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. Usually, the programmer intends a particular mapping of Giotto tasks to hosts as well as a mapping of Giotto ports to networks. A reasonable mapping for the two-robot example may distribute the `commandX`, `evadeX`, and `motorCtrX` tasks on robot `X`, respectively, where the `motorCtrX` task

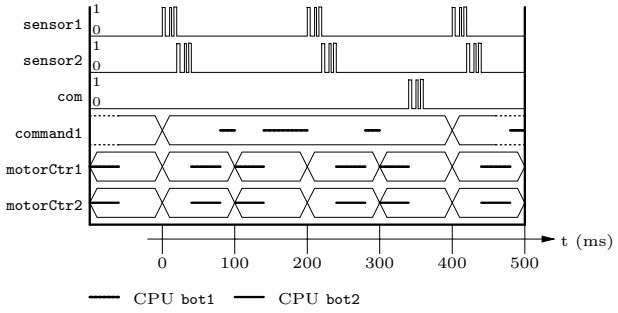


**Figure 14: The timing diagram for a round of the `Lead1Follow` mode with scheduling details**

should get the highest priority because of its shortest deadline. Consequently, the values in `com`, `fin`, and `sensorX` have to be communicated between the two robots.

Starting with a pure Giotto program, which does not contain any platform-related description, the programmer may gradually provide the Giotto compiler, using Giotto annotations, with more and more details about the target platform. A *Giotto annotation* falls in one of three possible categories: (1) a *Giotto-P annotation* (platform) specifies names and unique IP addresses for all hosts and networks as well as a list of priorities for each host in the system; (2) a *Giotto-S annotation* (schedule) specifies for a Giotto mode the Giotto task-to-host mappings and the priorities of the Giotto tasks; (3) a *Giotto-C annotation* (communication) specifies for a Giotto mode the Giotto port-to-network mappings and the time slots of the Giotto ports. These annotations are 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-C annotations would specify time slots for tasks and priorities for communications.

The Giotto program of Figure 9 contains examples of all three types of Giotto annotations. A Giotto-P annotation at the top of the program provides details on the two-robot platform. There are two hosts called `bot1` and `bot2`, which are connected by a network called `net12`. The Giotto compiler may exploit this information to improve code generation. However, more specific information is given by the Giotto-S and Giotto-C annotations in the example. Note that for flexibility we use symbolic names rather than numbers for Giotto task priorities and Giotto port time slots.

For instance, in the `Lead1Follow` mode, the `command1` task is assigned to `bot1` with the priority `p1`, which is lower than the priority `p0` of the `motorCtr1` task running on `bot1` as well. Consider Figure 14, which depicts the timing diagram for a round of the `Lead1Follow` mode with scheduling details. The dotted line shows which Giotto task is running on `bot1`'s CPU. The lower priority `command1` task gets the CPU only when the `motorCtr1` task is finished. In order to allow both robots to evaluate the mode switch predicates, the `sensorX` ports have to be exchanged between the robots twice per

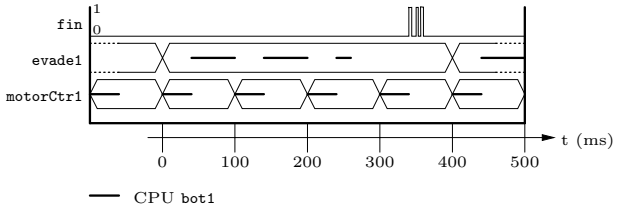


**Figure 15: The timing diagram for a round of the `Evade1Stop` mode with scheduling details**

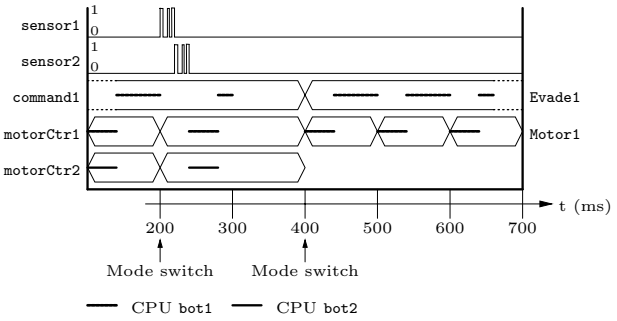


**Figure 16: The timing diagram for Giotto mode switches to the `Stop1` mode and to the `Evade1Stop` mode with scheduling details**

round because of the mode switch period of 200ms. Since the sensors are sampled at the 0ms and 200ms time instant, their values cannot be transmitted early. The `push` command indicates that the output host initiates the communication. The dual `pull` command is also possible, for example, to support less capable distributed sensors.

The signals from the `sensor1` and `sensor2` ports in Figure 14 show the exact timing for the transmission of the `sensor1` and `sensor2` values, respectively. Note that we assume that communication takes CPU cycles. The delay of 40ms of the sensor value update also delays the final decision of performing a mode switch which, however, does not affect the Giotto's semantical requirements as long as all Giotto tasks still meet their deadlines. Finally, the signal for the `com` port in Figure 14 shows that the `com` value is transmitted once per round from `bot1` to `bot2` early enough to be available at the beginning of the next round. The early transmission comes at a cost: the task `command1` has to finish its computation earlier than by the end of its period.

Figure 15 shows the timing for a round of the `Evade1Stop` mode with scheduling details. In this mode, only the `fin` port has to be transmitted from `bot1` to `bot2` once per round, as shown by the top-most line between the 300ms and 400ms instants. The timing and scheduling details for two Giotto mode switches from the `Lead1Follow` to the `Stop1` mode and then to the `Evade1Stop` mode are shown in Figure 16. Although the mode switch to the `Stop1` mode happens logically at the 200ms time instant, it is actually performed 40ms later by starting the `motorCtrX` tasks of the `Stop1` mode rather than of the `Lead1Follow` mode. Note that the transmission of the `com` port is skipped, because its value is not needed by any Giotto task in the `Stop1` 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 and Giotto-C annotations allow even more 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 design methodology for embedded control systems that is based on the the programming language Giotto. In Giotto, the programmer specifies the functionality and timing of a control design, leaving the specification of scheduling 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 scheduling and communication directives. 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. 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 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. A particularly interesting platform is the time-triggered architecture [9], which has already built-in many of the primitives on which the abstract Giotto model is based.

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) [3], particularly MetaH [12]. 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 allow the compilation of

scheduling code to connect tasks written in conventional programming languages. MetaH is designed for real-time, distributed avionics applications. Giotto can be seen 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 [7].

The goal of pure Giotto —to provide a platform-independent programming abstraction for real-time systems— is shared also by the synchronous reactive programming languages [5], such as Esterel [2], Lustre [6], or Signal [1]. 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.

## 7. 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 [4] implementation of Giotto.

This research was supported in part by the DARPA SEC grant F33615-C-98-3614, the DARPA MoBIES grant F33615-00-C-1703, the MARCO GSRC grant 98-DT-660, the AFOSR MURI grant F49620-00-1-0327, and the NSF ITR grant CCR-0085949.

## 8. REFERENCES

[1] A. Benveniste, P. L. Guernic, and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16(2):103–149, September 1991.

[2] G. Berry. The foundations of Esterel. In C. S. G. Plotkin and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[3] P. Clements. A survey of architecture description languages. In *Proc. 8th International Workshop on Software Specification and Design*, 1996.

[4] 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. Technical Report UCB/ERL M99/44, University of California, Berkeley, CA, July 1999.

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

[6] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9), 1991.

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

[8] T. Henzinger, B. Horowitz, and C. Kirsch. Giotto: A time-triggered language for embedded programming. Technical Report UCB//CSD-00-1121, University of California, Berkeley, 2000. Available at: `www.eecs.berkeley.edu/~fresco/giotto`.

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

[10] Y. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. *ACM SIGPLAN Notices*, 30(11):88–98, 1995.

[11] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Time-Critical Computing Systems*, pages 157–179, 2000.

[12] S. Vestal. MetaH support for real-time multi-processor avionics. In *Proc. Joint Workshop on Parallel and Distributed Real-Time Systems*, pages 11–21, 1997.