Giotto Tutorial

M.A.A. Sanvido, Aaron Walburg
Technical Memorandum UCB/ERL M04/30
University of California at Berkeley
giotto@ic.eecs.berkeley.edu
August 2004

Forward:

The Giotto[1] Development Kit (GDK) is a Java implementation of: (1) a compiler for the embedded programming language Giotto, (2) a run-time environment for java, (3) a distributed target machine called the embedded machine, and (4) some ready-to-run examples. Giotto is being developed at the Electrical Engineering and Computer Sciences Department of the University of California at Berkeley.

Installation:

To install the GDK, run the installer for your platform on your machine. If you do not have one, you may request a copy from the Giotto home page at: http://www-cad.eecs.berkeley.edu/~giotto

1 Introduction:

As a tutorial, we will compile a virtual hovercraft program which utilizes Giotto to generate java. The tutorial generates a program whereby a user may use the mouse to click on a location of the screen which then serves as a target for the

---

[2] This work is supported by the National Science Foundation (NSF Award Number CCR-00225610), the Defense Advanced Research Projects Agency (DARPA), and Chess (the Center for Hybrid and Embedded Software Systems), which receives support from NSF and the following companies: Daimler-Chrysler, Hitachi, Honeywell, Toyota and Wind River Systems.

hovercraft, which will move to that selected location under its own power. The program controls the left and right jet engines of this hovercraft. The tutorial will guide us through programming revisions. With each revision more functionality is added. The code revisions are listed below. These example files are located in the "examples" directory and are listed below in order of increasing complexity:

hovercraft01.giotto - No simulation window.
hovercraft02.giotto - Simulation window active, but no control of the hovercraft.
hovercraft03.giotto - Simulation window active, computation made for the current hovercraft location, but no control of the hovercraft is possible.
hovercraft04.giotto – Simulation window active; Hovercraft engines have thrust set at equal power, therefore hovercraft has no turning capacity.
hovercraft05.giotto - Simulation window active; Hovercraft functions completely.

Compiling the Code:

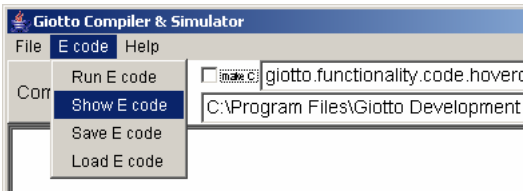
Each of the following steps may be repeated for the above examples.

Step 1: Compiling and running a Giotto program.

Start the Giotto application by clicking on the Start Menu, then selecting "Programs", then "Giotto Development Kit", then "Start." This opens the Giotto window:

After having started the Giotto Development Kit (GDK) open the file `hovercraft01.giotto`. In the Giotto application select “Open” from the “File” menu and open the Giotto program entitled `hovercraft01.giotto` which is located in the “Examples” directory. Once the file has been loaded the program will appear in the program editor.

To compile the program press the “Compile” button. The GDK will compile the text in the program editor (upper GDK window) and print the compilation results in the lower of the GDK window. The E code generated by the compiler can be viewed by selecting “Show E Code” from the E code Menu:



Step 2: Linking the Giotto program with Java functionality code.

The generated E code is not linked with any particular functionality code unless a code option has been checked. There are two code options located next to the Compile button: “Make Java” and “Make C”.

To link E code with external functionality, you need to check one or both of the options in the top of the GDK window and if “C Code” is selected you may make further code refinements. In this tutorial we will only consider java functionality, and therefore you should check the java option to execute code on a java platform.

Check the java option and set the compiler path:

A. Check the box labeled “make Java”.

B. Before compiling, set the compiler path to: “.” Do this by entering “giotto.functionality.code.hovercraft” in the topmost field in the GDK interface. The external java functionality classes will be loaded from the directory specified in this topmost field:

C. Press the “Compile” button.

At this point the compiler loads the external functionality, i.e. the java classes implementing the functionality, and generates E code that is linked with the external functionality.

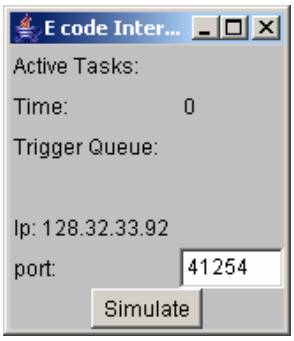
If you look again at the E code window (in the E code menu select “Show E code”) you will see that the E code references to external java classes.

Although we will not be using these features in this tutorial there are also three options available at the far right which relate to the “C Code” functionality: These are “Simulink”, “Annotated”, and “Dynamic”. The Simulink option is used to generate Simulink compatible C code functionality stubs. The Annotated option specifies to the compiler to generate code considering the annotated worst-case execution times, and code distribution. The Dynamic option allows the generation of dynamically loadable E code. Please refer to [1,2,3,4] for more details.

Step 3: Running E code

The linked E code can now be executed by the GDK. The GDK integrates an E code interpreter (called an E machine) that executes the E code. To run compiled E code select “Run” from the E code menu.

This will open a small window:



The window presents basic information about the state of the E machine. To run the simulation press the “Simulate” button and the E Machine will start executing the E code. You can stop the execution at any time by pressing the button again (now labeled “Stop”).

Step 4: Developing the Controller for the Hovercraft.

Having successfully compiled your first Giotto program, `hovercraft01.giotto`, we may now compile code which will increase the functionality of the hovercraft in successive steps which is revealed by running the simulation at each stage. Repeat steps 1-3 using the successive code examples. `hovercraft02.giotto` creates a simulation window, but no control of the hovercraft. This Giotto program is essentially the same as `hovercraft01.giotto` with the addition of the java functionality specification. We will sequentially add and modify the code, at first adding only forward thrust to the hovercraft engines, and thereafter increasing the controller complexity to a fully functional hovercraft simulation. In order to avoid having to code by hand each of the steps you can simply open the files from the example directory. The files are named `hovercraftXX.giotto` where “XX” is a number.

Explanation of `hovercraft02.giotto` code
The program is divided into four different sections: 1) A section defining the actuator ports, 2) A section defining the output ports, 3) A section defining the tasks, and 4) A section defining the drivers.

Defining the **actuator ports**:

```
actuator
    real_port     rightJet uses PutRightJet;
    real_port     leftJet  uses PutLeftJet;
```

An actuator is a port that directly accesses the hardware. In our specific case, the actuators are the two engines connected to the hovercraft, i.e. the `rightJet` and the `leftJet`. In order to access the hardware we need to specify the external functionality to call. In our example are the java class `PutRightJet` and `PutLeftJet` respectively.

Defining the **outputs**:

```
output
    real_port     turn            := real_zero;
    real_port     thrust          := real_zero;
    bool_port     window          := HovercraftWindow;
```

We need also to define storage location for the program. Output ports are memory locations used by tasks in order to store global data. To define a deterministic starting point of the program, each port has to be initialized. In our particular case we define and initialize two ports, i.e. turning capacity and thrust which will be initialized by calling the external java class “real_zero”. The third port window is a port used only to open and close the simulator, i.e. the HovercraftWindow class is responsible for opening the hovercraft simulation window.

Defining the **tasks**:
At this point we need to define the actual task we would like to perform. A task definition specifies the task name and the task input parameter, and the output port bound to the task. In this firs example we will use only a simple idleTask that will schedule an externally linked java class Idle.

```
task idleTask() output (turn, thrust) state () {
    schedule Idle(turn, thrust)
}
```

Defining the **drivers**:
In Giotto any data has to be transported by means of “drivers”. A driver is a function that transports data conditionally from source ports to destination ports, where a port can be an output port, an actuator port, or a sensor port (we will discuss sensor ports later). The transport is conditional and has to be performed by an external Java function. In our first example we had simply called the `constant_true()` conditional, and the `dummy()` functions, that do not perform any functionality. In this particular example we only need two drivers: one to transport data from the task outputs (turn, thrust) to the left engine and one for the right engine.

```
driver leftMotor(turn, thrust) output (real_port left) {
    if constant_true() then dummy()
}

driver rightMotor(turn, thrust) output (real_port right) {
    if constant_true() then dummy()
}
```

At the end of the Giotto program, we still need to specify how the program actually works and connect tasks and ports via the drivers. In Giotto we can have different “modes” and we switch between them. A mode is a particular connection of a task and ports via drivers. We will address this topic in the next few sections. At this point, we only want to define one mode, i.e. the mode “idle” and inform the system that when it starts should begin in the idle mode. In this particular mode, we define three variables: running period, how and when the actuator ports need to be updated by the drivers, and how often (frequency) the idleTask has to be invoked. Note that the idleTask does not have any input, and therefore no driver is needed to transport data to its input parameters.

```
start idle {

    mode idle() period 1000 {
        actfreq 1 do leftJet(leftMotor);
        actfreq 1 do rightJet(rightMotor);
        taskfreq 1 do idleTask();
    }

}
```

(Refer to the file "hovercraft02.giotto" for the above code.)

Step 5: Adding some sensors and a task to compute the hovercraft position and error from the target.

Open, compile, and run `hovercraft03.giotto`. We’ve made the previous program more complex by introducing three sensors that are capable of reading the position of the hovercraft (`positionX`, `positionY`, `angle`), and three sensors that read the input from a console in which an operator will set the desired position and orientation of the hovercraft (`targetX`, `targetY`, `targetAngle`). Furthermore, we will add a task that will compute the distance from the actual position to the destination.

At this point we still need to define the sensor that will provide the actual and target position of the hovercraft to the system. Therefore, we will add the following code to the beginning of the hovercraft program. Note that in the previous code we did not define any sensors, and so we will have to add the sensor definition to the hovercraft02.giotto. The sensors use the external java functionality `GetPos` and `GetTarg` to get the values from the real sensors.

```
sensor
    real_port  positionX   uses GetPosX;
    real_port  positionY   uses GetPosY;
    real_port  angle       uses GetPosA;
    real_port  targetX     uses GetTargX;
    real_port  targetY     uses GetTargY;
    real_port  targetAngle uses GetTargA;
```

The output ports `errorX`, `errorY`, `errorAngle`, `targetDirection` must be added to the output definition of the Giotto program.

```
    real_port     errorX          := real_zero;
    real_port     errorY          := real_zero;
    real_port     errorAngle      := real_zero;
    real_port     targetDirection := real_zero;
```

In `hovercraft03.giotto` we've added a new task called `errorTask`, the task computes the distance of the hovercraft from the target position. The distance is then stored in the output ports `errorX`, `errorY`, `errorAngle`. In addition, the task also computes the direction of the target in targetDirection. The schedule defines the name of the external functionality to be called: in this case the class `Error`.

```
task errorTask(real_port posX, real_port posY, real_port posA,
               real_port tgtX, real_port tgtY, real_port tgtA)
     output (errorX, errorY, errorAngle, targetDirection) state () {
    schedule Error(posX, posY, posA, tgtX, tgtY, tgtA,
                   errorX, errorY, errorAngle, targetDirection)
}
```

In order to complete this step we still need to define how the data has to be transported from the sensors to the `errorTask`. To do so we define a driver `getPos` that gets the sensor values and add to the idle mode the task `errorTask`. Note that the task `errorTask` has a frequency of 2, and that the task will call the `getPos` driver in order to receive its inputs.

```
driver getPos (positionX, positionY, angle,
               targetX, targetY, targetAngle)
       output (real_port posX, real_port posY, real_port posA,
               real_port tgtX, real_port tgtY, real_port tgtA) {
    if constant_true() then
       copy_real_port6(positionX, positionY, angle,
                       targetX, targetY, targetAngle,
                       posX, posY, posA, tgtX, tgtY, tgtA)
}
```

And defined here is the modified `idle` mode:

```
    mode idle() period 1000 {
        actfreq 1 do leftJet(leftMotor);
        actfreq 1 do rightJet(rightMotor);
        taskfreq 1 do idleTask();
        taskfreq 2 do errorTask(getPos);
    }
```

(Refer to the file `hovercraft03.giotto` for the above code.)

Step 6: Introducing mode switches

The previous program `hovercraft03.giotto` did not implement any useful control strategy besides computing the error position of the hovercraft. In this step we want to equip the hovercraft with the capacity to move forward in a fixed direction and be able to advance toward a point in front of it, i.e. we will restrict the movement of the hovercraft to a single direction. To do so we need to introduce a new mode and a mode switch. As explained before, a mode is a collection of tasks running at different periods and a collection of drivers transporting data from and to sensors, actuators, and tasks.

In order to provide this mobility to the hovercraft we add two mode switches to the idle mode. A mode switch calls a mode driver which returns “true” if a switch to another mode has to be completed. In our particular case we add a mode switch to change from the idle mode to the forward mode, while another mode switch remains in the idle mode. Moreover we have to add another mode. The mode forward is very similar to the idle mode, except the `idleTask` is replaced by the `forwardTask`.

```
start idle {

    mode idle() period 1000 {
        actfreq 1 do leftJet(leftMotor);
        actfreq 1 do rightJet(rightMotor);
        exitfreq 1 do idle(goIdle);
        exitfreq 1 do forward(goForward);
        taskfreq 2 do errorTask(getPos);
        taskfreq 1 do idleTask();
    }


    mode forward() period 200 {
        actfreq 1 do leftJet(leftMotor);
        actfreq 1 do rightJet(rightMotor);
        exitfreq 1 do idle(goIdle);
        taskfreq 2 do errorTask(getPos);
        taskfreq 1 do forwardTask(getErr);
    }

}
```

The mode switch drivers `goIdle` returns true if the position of the hovercraft has reached the target, while `goForward` returns true if the target is straight in front of the hovercraft.

```
driver goForward(errorX, errorY, errorAngle, targetDirection)
       output () {
    if Push2Target(errorX, errorY, errorAngle, targetDirection)
    then dummy()
}

driver goIdle(errorX, errorY, errorAngle, targetDirection)
       output () {
    if GoIdle(errorX, errorY, errorAngle, targetDirection) then dummy()
```

```
}
```

We still need to define the `forwardTask`, and the input driver `getErr` responsible for transporting the data from the `errorTask` to the `forwardTask`.

```
task forwardTask(real_port errX, real_port errY, real_port errAngle,
real_port dir)
     output (turn, thrust) state () {
    schedule Forward(errX, errY, errAngle, dir, turn, thrust)
}

driver getErr (errorX, errorY, errorAngle, targetDirection)
       output (real_port errX, real_port errY,
               real_port dir, real_port errA) {
    if constant_true() then
       copy_real_port4(errorX, errorY, errorAngle, targetDirection,
                       errX, errY, errA, dir)
}
```

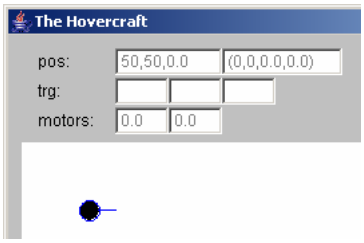
At this point we've implemented a controller capable of moving the hovercraft from an idle position to a target directly in front of the hovercraft. After compiling and running the E code interpreter (E machine), you will be able to play with the hovercraft. In the simulation window, set the target position of the hovercraft by using your mouse or enter the coordinates and orientation of the hovercraft in the text fields. Note,however,that the hovercraft does not yet move.

(Refer to the file `hovercraft04.giotto` for the above code.)

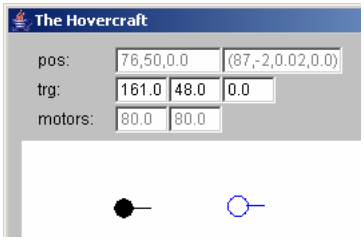
Step 7: Full hovercraft program.

The final step is to implement the fully capable controller. The controller introduces two more modes, one which empowers the hovercraft to turn in place (toward the target), and another mode which properly orients the hovercraft to acquire the target orientation. These two modes are used to move the hovercraft in a straight line from the actual hovercraft position to the target. The first mode turns the hovercraft in the direction of the target, then moves it forward to the target. Once the target is reached, the second mode orients the hovercraft. The full implementation is in the file `hovercraft05.giotto`.

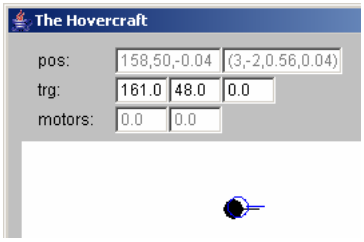
In the full hovercraft program the hovercraft seeks the target at any coordinate. Running the full simulation produces this initial window:



Use your mouse to select target coordinates on the screen (an empty blue circle will appear):



The hovercraft then moves itself to those coordinates and orients itself at that location in approximately the same direction:



Step 8: implementing external functionality in Java

Until now, we have assumed that all external java functionality code was available. However, if you wish to develop your own application you need to be able to develop your own code and make it E code linkable. The java implementation of the E Machine expects all external functions to implement special java interfaces for drivers and tasks. In this step we will briefly present these interfaces. In order to compile and run the system, the compiler needs to know where the classes are located (as was presented in step 3). However, to be able to run, the classes must also be identified by the runtime system. To achieve this you must add the functionality classes to the JVM class path of the system[2]. Note that if you want to be able to save the E Code, all your classes will also have to implement the `Serializable` interface.

Implementing a Port:

[2] Please refer to your particular JVM documentation to do so.

In Giotto all ports are stored in external java objects that implement the `PortVariable` interface. The system predefines the `PortVariable` implementations `real_port`, `int_port`, `bool_port`, but you can implement your own types by implementing the PortVariable interface. The method `copyValueFrom` wuill implement the copy of the port values to the `to` object.

```
public interface PortVariable {
     public void copyValueFrom(PortVariable to);
}
```

Implementing a Task:

In order to implement the functionality of a Giotto task you need to implement a class with the interface `TaskInterface`. `TaskInterface` defines only a single method `run`, which will be invoked by the E Machine during execution. The `run` method accepts an argument of type `Parameter`. The parameters store all the input and output ports defined by the Giotto task. The ports are numbered in the order of appearance in the Giotto task definition, with the inputs coming first. To implement your functionality you need to access the ports, perform your computation and store the values back on the output ports. The system will then automatically copy the output ports to the global ports, as defined by the Giotto language semantic. For example, to access the first port you need to call `p.getPortVariable(0)`. This will return the `PortVariable` referencing to the port. Once you have the reference to the `PortVariable` you can acquire and set its value depending on the `PortVariable` type.

```
public interface TaskInterface {
  public void run(Parameter parameter) throws StopException;
}

public class Parameter implements Serializable {

  public PortVariable getPortVariable(int index);
  public int getNops();

}
```

Now let's observe the implementation of the `Forward` task used in the hovercraft examples. The run methods: read the input ports (0,1, and 3), compute the control law, and write the output to the output ports (4, and 5).

```
public class Forward implements TaskInterface, Serializable {

  public void run(Parameter p) {
    //access the input ports 0,1 and 3
    float  ex = ((real_port)p.getPortVariable(0)).getFloatValue();
    float  ey = ((real_port)p.getPortVariable(1)).getFloatValue();
    float  dir = ((real_port)p.getPortVariable(3)).getFloatValue();
```

```
   //computes the control law
   float r, fp;
   r = (float)Math.sqrt(ex*ex + ey*ey);
   fp = (float)Math.cos(dir)* r*Kp;
   if (fp<0) fp = 0;
   float tp = 0;

   //write back the results to the output ports 4 & 5
   ((real_port)p.getPortVariable(4)).setFloatValue(tp);
   ((real_port)p.getPortVariable(5)).setFloatValue(fp);
  }

}
```

Implementing an input driver:

The implementation of an input driver is similar to the task implementation. The only difference is in the way the E Machine will call the method. In case of a task the task will be wrapped in a Java thread while a driver will be called directly from the E Machine thread. A driver transports data from its input to the outputs. Similar to a task, the input and output ports are stored sequentially in the parameter and accessed via their index.

```
public interface DriverInterface {
     public void run(Parameter parameter);
}
```

The following drivers are predefined and used in the hovercraft examples: `Copy_int, copy_bool, copy_real, copy_real_2, copy_real_4, copy_real_6.`

Implementing a Sensor and an Actuator:

In order to access the sensor directly and drive an actuator directly we need implement sensor and actuator drivers. These are essentially similar to input drivers with the difference that they access external values.

For example, below is the implementation of the `GetPosX` driver used in the hovercraft examples. The run method writes the value of the actual position of the hovercraft (accessed via the Hovercraft.main.getPosX() call) in the first PortVariable of the parameter.

```
public class GetPosX implements DriverInterface, Serializable {

      public void run(Parameter parameter) {
      ((real_port)parameter.getPortVariable(0)).
          setFloatValue(Hovercraft.main.getPosX());
      }
}
```

Here is an example of the `PutLeftJet` driver, that actuates the left jet of the hovercraft.

```
public class PutLeftJet implements DriverInterface, Serializable {

      public void run(Parameter parameter) {
            real_port f = (real_port)parameter.getPortVariable(0);
            float limit = Hovercraft.LimitMotor(f.getFloatValue());
            Hovercraft.main.lm = limit;
    }
}
```

Implementing a Mode Switch:

At last we only need to be able to implement the mode switches. This is done by implementing the interface `ConditionInterface`. Essentially this interface is similar to the `DriverInterface` with the only exception that the method `run` returns a boolean. If the boolean is true the mode switch will be taken, otherwise the next mode switch declared in the mode is tested. If no mode switches return true, the Giotto program will remain in the same mode in the next period.

```
public interface ConditionInterface {
     public boolean run(Parameter parameter);
}
```

And here the implementation of the GoForward mode switch.

```
public class GoForward implements ConditionInterface, Serializable {
  public boolean run(Parameter p) {
    float  ex = ((real_port)p.getPortVariable(0)).getFloatValue();
    float  ey = ((real_port)p.getPortVariable(1)).getFloatValue();
    float  ea = ((real_port)p.getPortVariable(2)).getFloatValue();
    float  ea2 = ((real_port)p.getPortVariable(3)).getFloatValue();
    float ter = (float)Math.sqrt(ex*ex + ey*ey);
    return (ter > 5) & (Math.abs(ea) < 0.1);
  }

}
```

The same interface is used when implementing conditions in the Giotto drivers. For example the definition of the `leftMotor` driver used the `constant_true()` to test if the dummy class, which implemented the `DriverInterface` class) had to be performed

```
driver leftMotor(turn, thrust) output (real_port left) {
    if constant_true() then dummy()
}
```

Copyright

References

1. T.A. Henzinger, B. Horowitz, and C.M. Kirsch. Embedded control systems development with Giotto. In Proceedings of the International Conference on Languages, Compilers, and Tools for Embedded Systems, pages 64–72. ACM Press, 2001.
2. T.A. Henzinger, B. Horowitz, and C.M. Kirsch. Giotto: A time-triggered language for embedded programming. In T.A. Henzinger and C.M. Kirsch, editors, EMSOFT 01: Embedded Software, Lecture Notes in Computer Science 2211, pages 166–184. Springer-Verlag, 2001.
3. T.A. Henzinger and C.M. Kirsch. The Embedded Machine: Predictable, portable real-time code. In Proceedings of the International Conference on Programming Language Design and Implementation, pages 315–326. ACM Press, 2002.
4. T.A. Henzinger, C.M. Kirsch, M.A. Sanvido, and W. Pree. From control models to real-time code using Giotto. IEEE Control Systems Magazine, 23(1):50–64, 2003.
5. C.M. Kirsch, M.A. Sanvido, T.A. Henzinger, and W. Pree. A Giotto-based helicopter control system. In A. Sangiovanni-Vincentelli and J. Sifakis,

editors, EMSOFT 02: Embedded Software, Lecture Notes in Computer Science 2491, pages 46–60. Springer-Verlag, 2002.