Chapter 5 from: C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, H. Zheng "Heterogeneous Concurrent Modeling and Design in Java (Volume 3: Ptolemy II Domains)," Technical Memorandum UCB/ERL M04/17, University of California, Berkeley, CA USA 94720, June 24, 2004.

5

Giotto Domain

Authors: *Haiyang Zheng*
Edward Lee
Christoph Kirsch

5.1 Introduction

The Giotto model is a semantic model that describes the communication between periodic time triggered components. It was developed by Thomas Henzinger and his group. It was designed for deterministic and safety critical applications.

The main points about the Giotto model are:

1. A Giotto model is composed of one or more *modes* and each mode is composed of several *tasks*.

2. For every task, the design specifies a worst case execution time (WCET) which constrains the execution time of that task in the model.

3. Tasks are concurrent and preemptable.

4. Each task may consume some tokens and produce some tokens for other actors or itself, the produced tokens are not available until the end of the task's deadline.

5. Mode switching includes invoking or terminating some tasks.

6. There are constraints on mode switching, e.g., the states consistency of tasks.

More details of the Giotto model may be found at http://www-cad.eecs.berkeley.edu/~giotto.

5.2 Using Giotto

The execution time of an actor in the Giotto model is defined as the *period* (a parameter of the Giotto Director) divided by the *frequency* (a parameter associated with the actor). To configure the period of a Giotto model, modify the value of the period parameter. The default value of period is 0.1 sec. To configure the frequency of a task, add a parameter called *frequency* (the value has to be an integer). Without the explicit frequency parameter, the director assigns a default frequency 1 to the actor.

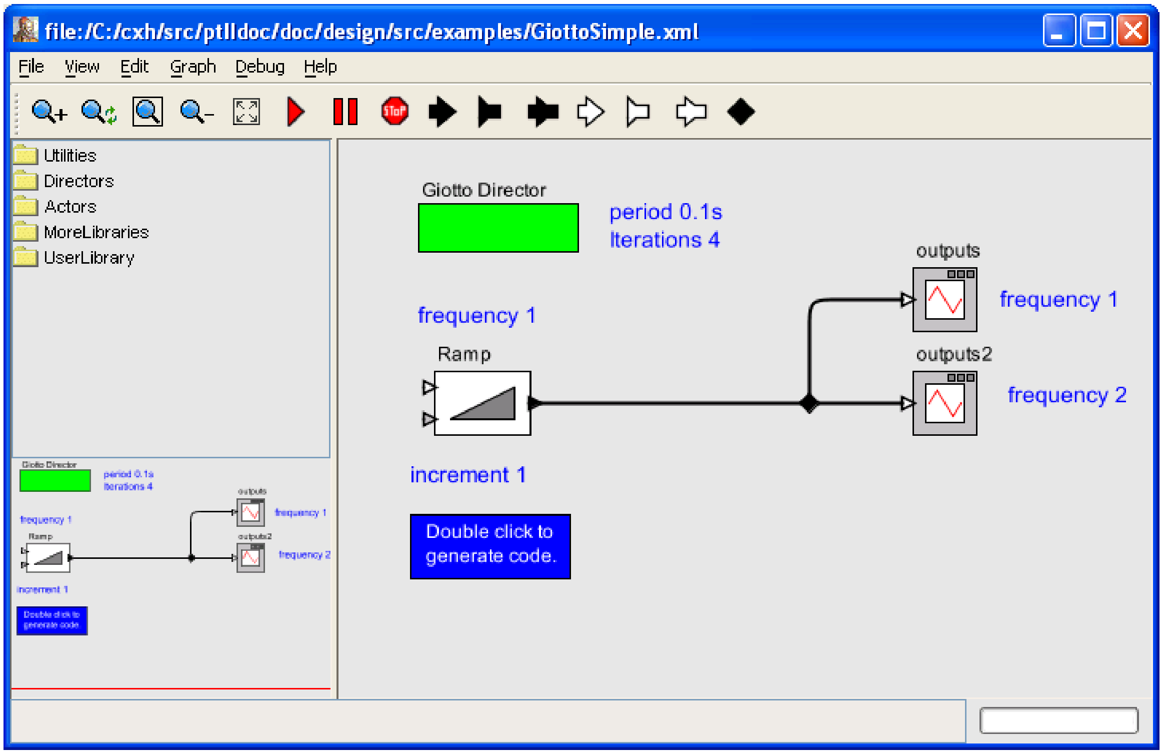

FIGURE 5.1. A Simple Giotto model with only one mode.

There is also an *iterations* parameter associated with the director, which is used to control the number of iterations of the model, or the total execution time of the model. The default value is 0, which means that the model executes forever.

There is one constraint when constructing models: each channel of an input port must have exactly one source. This ensures the determinacy of the model.

Figure 5.1 is a simple Giotto model. The simulation result of this model is shown in Figure 5.2. The blue box in Figure 5.1 is GiottoCodeGenerator. It is used to generate Giotto code for the E-Compiler for schedulability analysis. To use the GiottoCodeGenerator, drag the *CodeGenerator* into the graph editor from the tools on the left side under the directory *more libraries/experimental domains/ Giotto*. Double clicking this icon will pop up a text window with the generated code. The generate code for Figure 5.1 is shown in Figure 5.3.

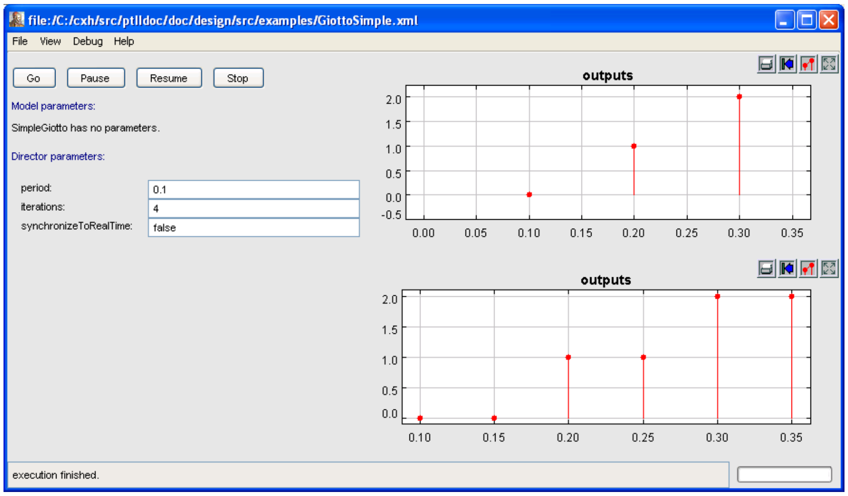

FIGURE 5.2. Simulation results for the model in Figure 5.1

```
sensor
actuator
output
  Token_port Ramp_output := CGinit_Ramp_output;
task Ramp (Token_port Ramp_trigger,Token_port
Ramp_step)
        output (Ramp_output)
        state ()
{
        schedule
CGRamp_Task(Ramp_trigger,Ramp_step,Ramp_output)
}
task outputs (Token_port outputs_input)
        output ()
        state ()
{
        schedule CGoutputs_Task(outputs_input,)
}
task outputs2 (Token_port outputs2_input)
        output ()
        state ()
{
        schedule
CGoutputs2_Task(outputs2_input,)
}
driver Ramp_driver ()
        output (Token_port
Ramp_trigger,Token_port Ramp_step)
{
}
driver outputs_driver (Ramp_output)
        output (Token_port outputs_input)
{
          if constant_true() then
copy_Token_port( Ramp_output, outputs_input)
}
driver outputs2_driver (Ramp_output)
        output (Token_port outputs2_input)
{
          if constant_true() then
copy_Token_port( Ramp_output, outputs2_input)
}
start SimpleGiotto {
  mode SimpleGiotto () period 100 {
    taskfreq 1 do Ramp(Ramp_driver);
    taskfreq 1 do outputs(outputs_driver);
    taskfreq 2 do outputs2(outputs2_driver);
  }
}
```

FIGURE 5.3. Generated Giotto code for the model in Figure 5.1

5.3 Interacting with Other Domains

During the design of real applications, big models are often decomposed into smaller models, each having their own model of computation. So, it is important to study the interactions between Giotto models and other models. A few discussions and examples are given in the following paragraphs.

5.3.1 Giotto Embedded in DE and CT

The interface between DE model and Giotto model is well defined. Embedded inside DE model, the Giotto model could easily be invoked to meet design requirements. The composite model gives a paradigm of asynchronous Giotto model triggered by discrete events compared with the normal Giotto model triggered by periodic time.

Figure 5.4 shows a Giotto model composed inside a DE model, which can be found at $PTII/ptolemy/domains/giotto/demo/Composite/Composite.xml. The details of the DE domain are in Chapter 14. The Giotto model runs with period 0.2 sec. and iterates twice each time it is invoked. There are two triggering events: one happens at time 0.0 sec. and the other at time 1.0 sec. The result is shown in Figure 5.5. The results in the *State* plot have a delay of 0.2 sec. with respect to the triggering events in the *Events* plot.

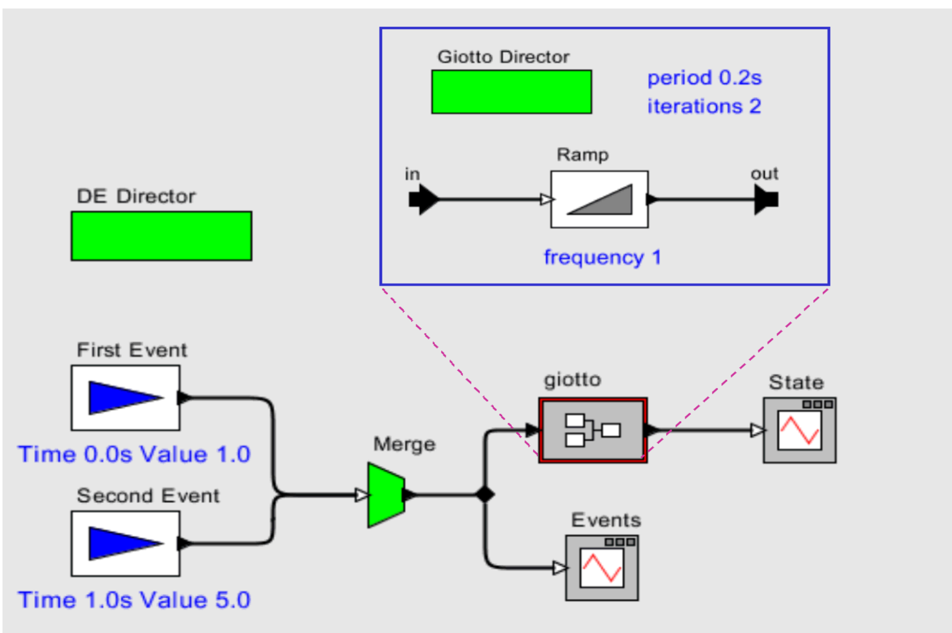

FIGURE 5.4. Giotto model embedded in DE model.

There are a few important issues:

i. The results in states plot has 0.2 sec. delay according to the Giotto semantics.

ii. For each input to the Giotto model, two outputs are generated since the value of the iterations parameter is 2.

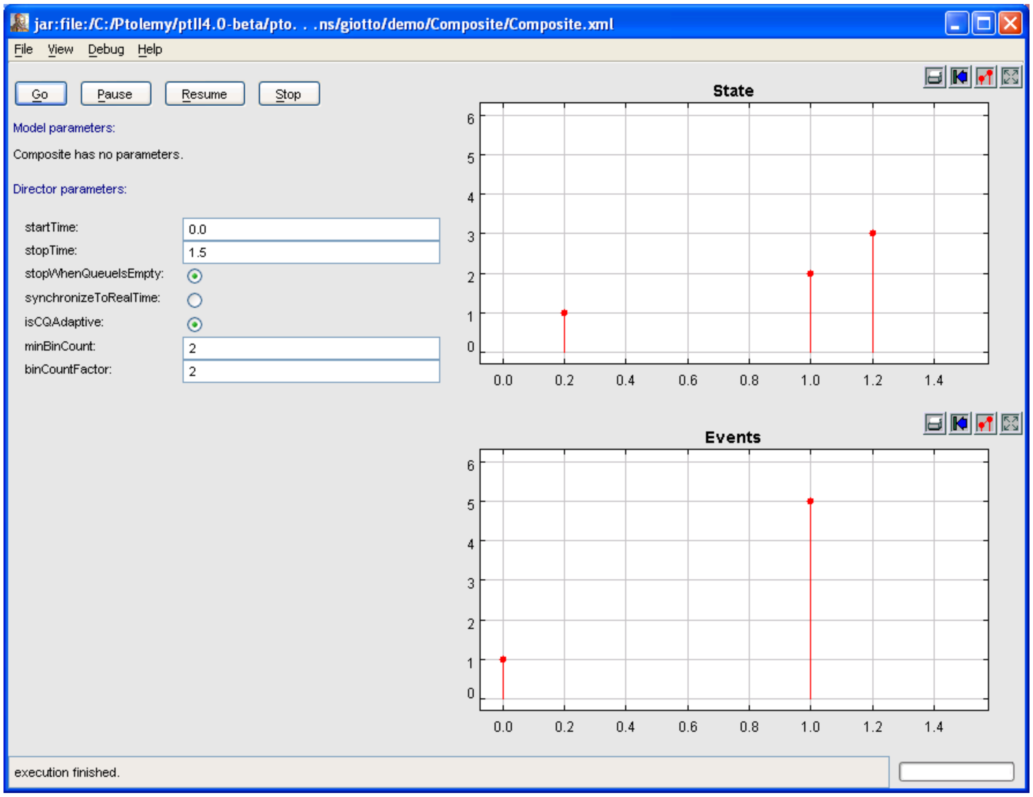

FIGURE 5.5. Simulation results of model of Figure 5.4

When a Giotto model is composed inside a CT model, the Giotto model is always invoked. So, the iterations parameter does not has effect.

5.3.2 FSM and SDF embedded inside Giotto

A Giotto model may be composed of several modes. To realize mode switching, we employed the modal model. A modal model is basically a FSM with the states which may be refined into other models of computations. The details of the modal model is in Chapter 16. In our example, the states are refined into the SDF models. The details of the SDF domain is in Chapter 15.

The model shown in Figure 5.6 can be found at $PTII/ptolemy/domains/giotto/demo/Multimode/Multimode.xml. This model is a simple implementation of mode switching where each mode has only one task, (implemented as a SDF model). The modal model has three states, init, mode1 and mode2. The default state is init and it is never reached again after the execution starts. The states mode1 and mode2 are refined into the tasks doing *addition* and *subtraction* respectively.

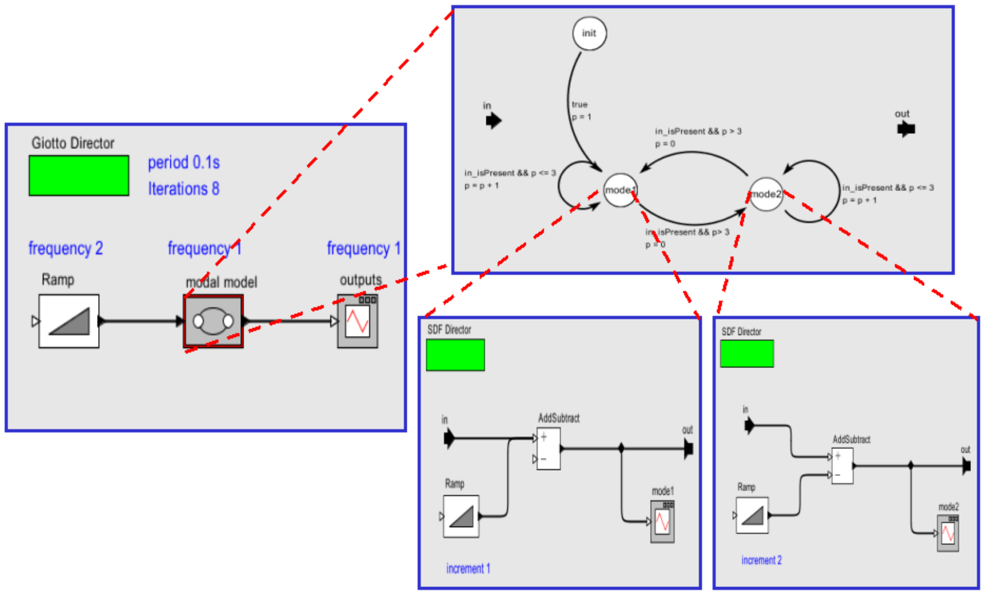

FIGURE 5.6. Modal model embedded in Giotto model.

The simulation result is shown in Figure 5.7. The *outputs* plotter resides in the Giotto model. *Mode1* plotter and *mode2* plotter reside in states of mode1 and mode2.

The *outputs* plot shows the results have 0.1 sec. delay according to the Giotto semantics. At time 0.4 sec., the *mode1* plot shows a mode switching (from mode1 to mode2) happens. However, the mode switching does not show on the outputs plot until 0.5 sec.

Note that in the *mode2* plot, the last result at 0.7 sec. does not show up in the outputs plot. The reason is that although the result of mode2 is available at 0.7 sec., it is not transferred to the outputs actor until 0.8 sec. Thus, the *outputs* plotter could not show the result until 0.8 sec., which exceeds the iterations limit.

5.4 Software structure of the Giotto Domain and implementation

The Giotto kernel package implements the Giotto model of computation. It's composed of three classes: *GiottoScheduler*, *GiottoDirector* and *GiottoReceiver*. Also, a code generation tool the E-complier is provided as *GiottoCodeGenerator*. The structure of classes is shown the Figure 5.8.

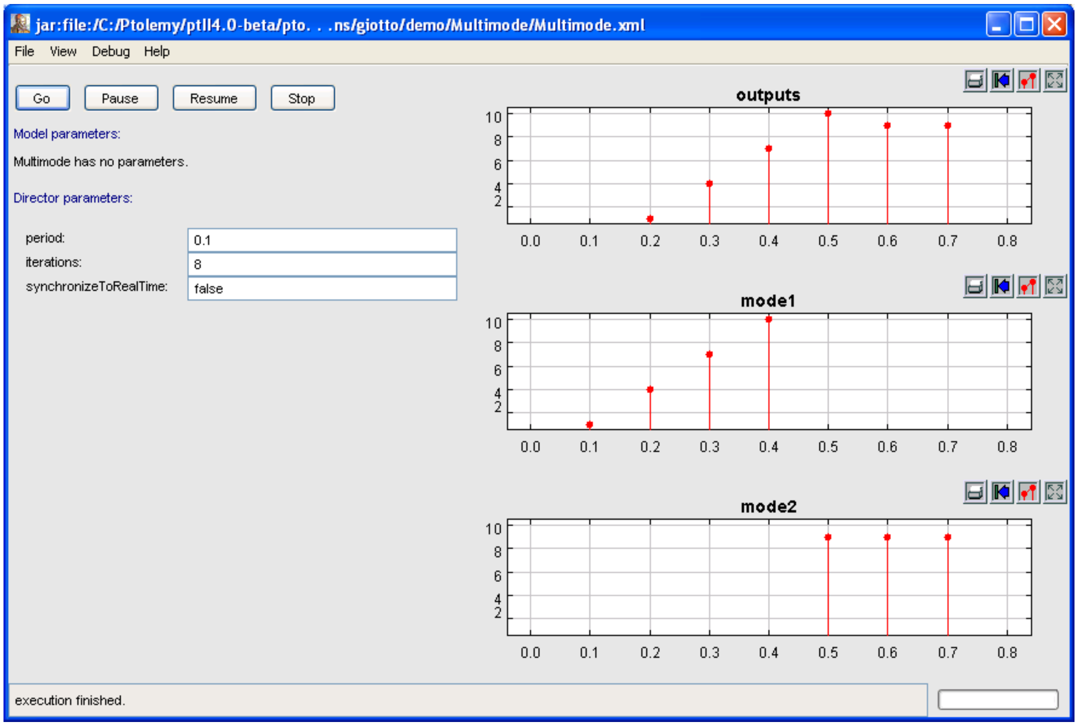

FIGURE 5.7. Simulation results for model in Figure 5.6.

5.4.1 GiottoDirector

GiottoDirector extends StaticSchedulingDirector class. It implements a model of computation according to the Giotto semantics with the help of the GiottoScheduler and the GiottoReceiver. GiottoScheduler provides a list of schedules and GiottoReceiver provides the buffered states.

There are three parameters associated with the GiottoDirector: *period*, *iterations* and *synchronizeToRealTime*. The execution phases of GiottoDirector include *initialize, prefire, fire* and *postfire*.

1. In the *initialize* phase, the director resets all the receivers and properly initializes the output ports of actors. The director also gets the list of schedules. A schedule is a list of actors to be fired at the same time. It synchronizes to the cpu time if the parameter synchronizeToRealTime is true.

2. In the *prefire* phase, the director updates the current time from upper level director if necessary. It also decides to firing or not by checking whether the current time is less than the expected execution time.

3. In the *fire* phase, the director iterates the list of schedules via index indicator *unitIndex*. Each time, the unitIndex is incremented by 1 referring to the next schedule. When it exceeds the schedule list size, it rounds back to 0. The director does two things in sequence: invoking all the actors listed in the schedule and transferring outputs of the actors after their executions. The director needs to be synchronized to real time if the parameter synchronizeToRealTime is true.

4. In the *postfire* phase, if the Giotto model is embedded, the director does not advance time by itself. Its next firing is scheduled by the executive director (in the example in Figure 5.4, the DE direc-

tor). Note that the last transfer of outputs happens after the execution of all the actors and no actors are fired. A boolean variable *transferOutputsOnly* is introduced to indicate the transfer. When the iterations requirement is first met, the director sets *transferOutputsOnly* to true and prepares for the next iteration. The postfire() method returns true. In the immediately following postfire phase, *transferOutputsOnly* is set back to false. The postfire() method returns false to terminate the model execution.

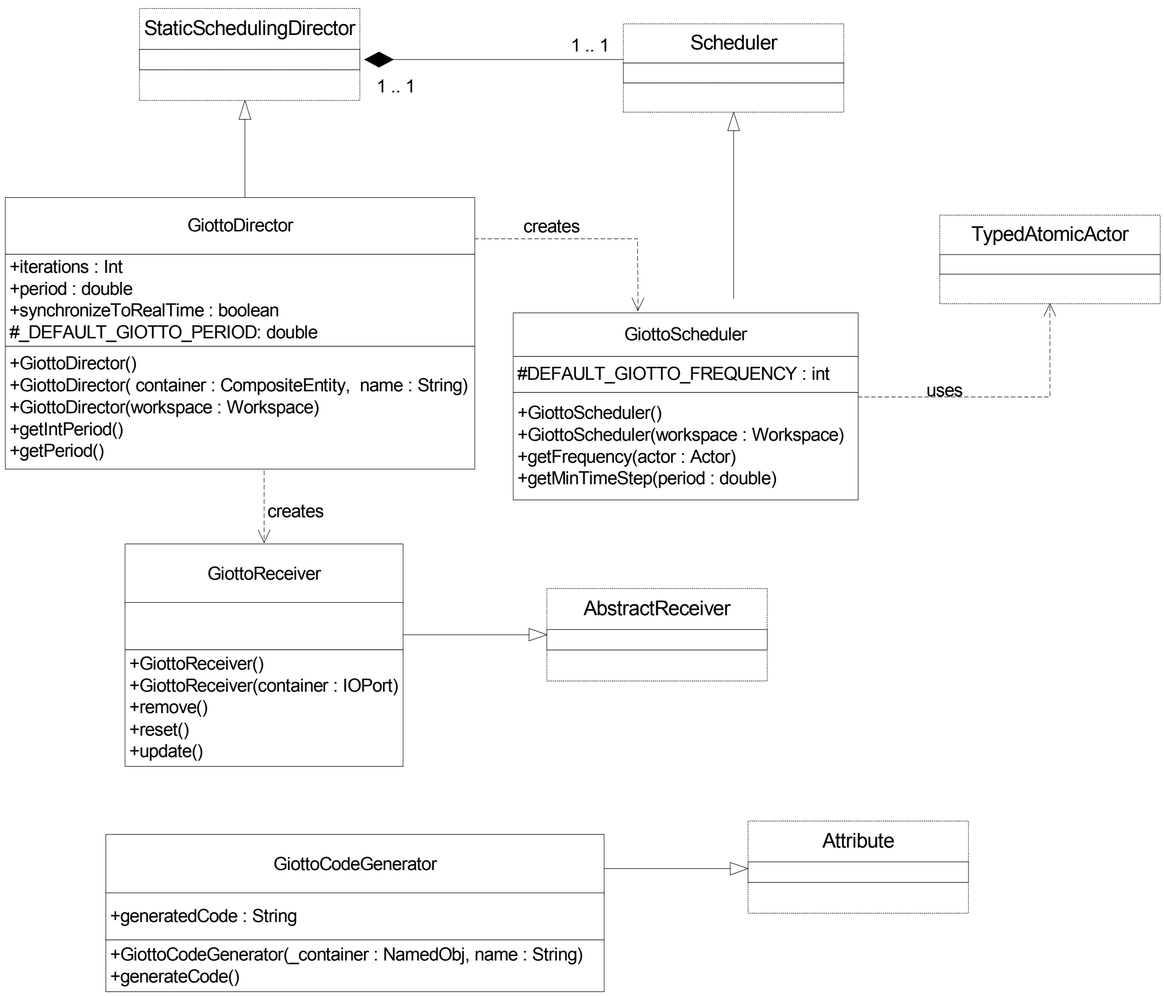

FIGURE 5.8. The static structure of the Giotto package kernel classes.

When the Giotto model is embedded inside other models, for example, the model in Figure 5.4. The Giotto director asks GiottoReceiver to call remove() instead of get(), otherwise, the *states* plotter will always be fired because the _token is not cleared.

5.4.2 GiottoScheduler

GiottoScheduler extends the Scheduler class. It is used to construct a list of schedules for the GiottoDirector. A schedule is a list of actors that will be fired by the GiottoDirector at the same time. GiottoScheduler provides two things for GiottoDirector: the minimum unit time increment for GiottoDirector to advance time and the list of schedules. To get schedule, use getSchedule() method from GiottoDirector.

GiottoScheduler first makes topology analysis to construct a list of the actors including the opaque composite actors and atomic actors. It also constructs an array *frequencyArray*, the elements are the frequency values associated with the actor list. With the frequencyArray, the greatest common divider (*gcd*) and the least common multiple (*lcm*) of all the frequency values are calculated. The minimum unit time increment is defined as *period / lcm*. With frequencyArray and lcm, another array: *intervalArray* is constructed to indicate when the actor to be added into schedule.

In order to compute the schedule, a simple timer: *giottoSchedulerTime* is introduced, which iterates from 0 to *lcm* with tick increment of *gcd*.

When constructing the list of schedules, there are two loops. The outer loop iterates the giottoSchedulerTime. The inner loop iterates the intervalArray. The inner loop constructs the *fireAtSameTimeSchedule*. The outer loop constructs a *schedule*, the list of the fireAtSameTimeSchedules. The Java code of schedule computation is shown in Figure 5.9.

```
Schedule schedule = new Schedule();

for (  _giottoSchedulerTime = 0; _giottoSchedulerTime < _lcm; ) {

    Schedule fireAtSameTimeSchedule = new Schedule();
    actorListIterator = actorList.listIterator();

    for (i = 0; i < actorCount; i++ ) {
        Actor actor = (Actor) actorListIterator.next();
        if ((_giottoSchedulerTime % intervalArray[i]) == 0)
                    {
                        Firing firing = new Firing();
                        firing.setActor(actor);
                        fireAtSameTimeSchedule.add(firing);
                    }
    }

    _giottoSchedulerTime += _gcd;
    schedule.add(fireAtSameTimeSchedule);
}
```

FIGURE 5.9. Schedule computation of GiottoScheduler.

5.4.3 GiottoReceiver

GiottoReceiver extends the AbstractReceiver class. The key point is that the GiottoReceiver has double buffers: *_nextToken* and *_token*. When the get() method is called, a **copy** of _token is consumed. When the put() method is called, only the _nextToken is updated. When the update() method is called, the _token is updated by _nextToken. When the remove() method is called, a copy of the _token is returned and the _token is cleared. It is the GiottoDirector that delays *update* calls to realize the Giotto semantics.

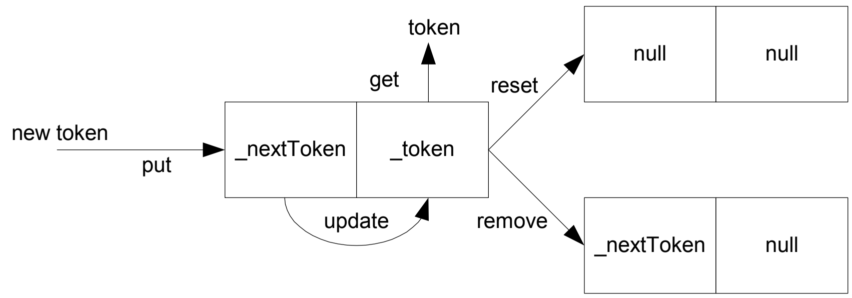

FIGURE 5.10. Working mechanism of GiottoReceiver.

The GiottoReceiver also has a reset() method. Reset is used to clear all the tokens including _nextToken and _token but returns nothing. Remove is used to return the _token and clear it but keeps _nextToken. Reset is used for initialization and remove is used for transfer of outputs to outside environment when the Giotto model is embedded inside other models.

5.4.4 GiottoCodeGenerator

GiottoCodeGenerator extends Attribute class. It is used to generate Giotto code for E-Compiler for schedulability analysis.

The current GiottoCodeGenerator works for one mode only. It iterates all the entities and treats them as tasks. From the input ports of the entities, source ports and their containers are traced. The model inputs are treated as sensors and the model outputs are treated as actuators.

The generated Giotto code usually has six parts: sensorCode, actuatorCode, outputCode, taskCode, driverCode and modeCode. The sensorCode and actuatorCode are the interfaces to the outside environment. The outputCode and driverCode describe the data dependencies. Note that for outputCode, it is illegal for an input port to have more than one source. TaskCode is the description of the computation of tasks (actors). ModeCode defines which tasks are in each mode, along with their parameters.

The example code is in Figure 5.3.