C Code Generation from the Giotto Model of Computation to the PRET Architecture

Shanna-Shaye Forbes, Ben Lickly, and Man-Kit Leung
{sssf,blickly,jleung}@eecs.berkeley.edu
May 15, 2009

***Abstract*—We present code generation from the Giotto model of computation in Ptolemy II to the Precision Timed (PRET) Architecture. Giotto is a time-triggered programming model that provides the user with methods to specify timing at a high level, and PRET is a processor architecture that emphasizes predictable timing. The goal of code generation is to automatically generate code that correctly implements the semantics of the model as the designer has specified. We use the ISA-level timing controls of PRET in the C code we generate to fulfill the timing constraints of the Giotto model. We run the generated code on the cycle-accurate PRET simulator to verify that our designs meet their deadlines.**

I. Introduction

In real-time embedded systems it is important to guarantee correct functionality as well as timing constraints. Timing in real-time embedded systems has always been of high importance. However, with recent trends toward incorporating X-by-wire systems in automotive and avionics systems, the need for ensuring and predicting timing has gained renewed attention. C, the de facto programming language for most embedded platforms, lacks constructs to specify precise timing. In addition, most embedded processors also lack constructs to provide deterministic and precise timing at the hardware level. In hard real-time systems where timing precision is critical, hardware support for precise timing is necessary. One model of computation with timing semantics used in embedded controllers is Giotto. A processor that provides deterministic and precise timing is PRET. This project maps the Giotto timing semantics to a processor with direct hardware support instead of best effort attempts at timing.

II. Giotto

Giotto is a time-triggered language and model of computation that allows a control engineer to specify the semantics of time-triggered sensor readings, task invocations, actuator updates, and mode switches independent of the platform used to implement it [3], [4].

Giotto is best utilized by hard real-time specifications that are periodic and feature multi-modal behavior. It is useful for control systems such as fly-by-wire or brake-by-wire where the responses of the system must be fairly periodic and have multiple modes of operation. The modes of operation can include a startup, cruise control/autopilot, normal operation as well as a mode in case of partial equipment failure.

In Giotto semantics tasks are executed at a specific frequency w_t within a specific period π. Tasks communicate through ports and they get input values from ports at the beginning of the tasks logical execution time π/ω_t and produce their outputs at the end of their logical execution time. Tasks execute concurrently and there is a one unit delay in communication between tasks.

The Giotto model of computation is implemented as a domain in the Ptolemy II simulation and modeling environment. To select a particular model of computation the user selects and uses a director associated with the model of computation. A Giotto model is created with a Giotto Director in Ptolemy II. The period π of the mode is specified as the period parameter to the director and the frequency of each task ω_t is specified as a frequency parameter to each Ptolemy II actor. If no values for the period and actor frequencies are provided as parameters default values of 0.1s and 1 are assumed respectively [1].

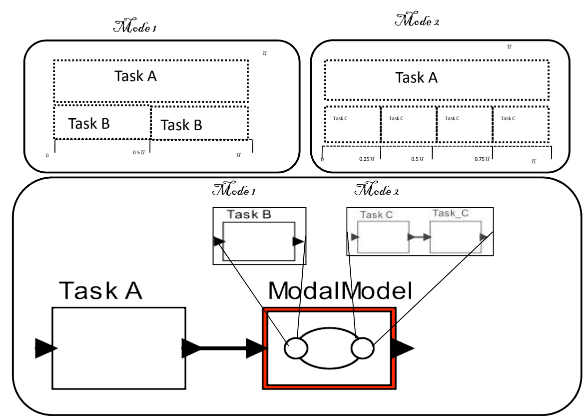

Fig. 1. A multi-modal Giotto specification modeled by Ptolemy II using the Modal Model Actor. Each mode is further refined to a sub-model containing, in this case, other Giotto tasks.

In [3] a mode in Giotto consists of all tasks to be run concurrently with a particular period. In Ptolemy II, a mode is slightly different but allows all models expressible in [3]. Ptolemy II allows the use of hierarchy that proves to be very convenient in the specification of control behavior. In addition it also reduces the number of distinct mode combination specifications that are necessary in [3]. A Ptolemy II mode is specified inside a finite state machine modal model and

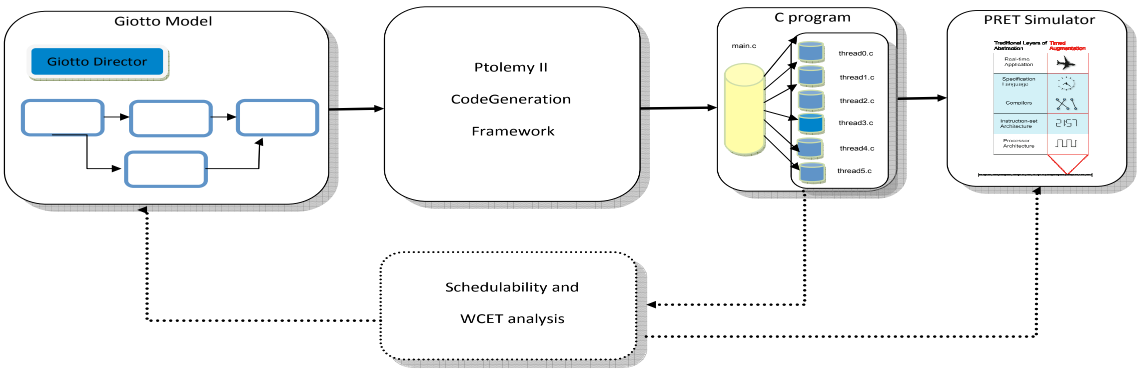

Fig. 2. The work flow of the design framework that iteratively refines code generation using schedulability and WCET analysis

improves the flattened specification present in [3] with the use of hierarchy. In Ptolemy II tasks, which are referred to as actors, at the same level of hierarchy execute concurrently and a modal model contains tasks that should be switched when a guard is enabled. If it is desirable to have three tasks: A, B, and C, where task A is always running and task C should replace task B when a certain condition is met, a designer could specify that in Ptolemy II as is shown in Figure 1. In Figure 1 task C is shown twice in the lower figure to indicate a frequency of 2. The lower portion of Figure 1 shows how the model is specified with Ptolemy II and the upper portion of the figure shows the logical execution times of each task based on their frequencies, and on the period parameter π of the Giotto Director.

Ptolemy II allows hierarchy through the use of composite actors. A composite actor contains actors and in some cases a director. If no director is present inside the composite actor the actor is transparent. If however there is a director present inside a composite actor the frequencies of the tasks inside the composite actor are all interpreted to be relative to the frequency of the composite actor itself. If a composite actor with frequency 2 contains a Giotto Director, and a task with frequency 3, the interpreted frequency of the task inside a composite actor is 6.

Each Giotto model is expected to specify a period as an attribute to the Giotto Director, the frequency of each task as an attribute to each actor, as well as initial values for outputs. If Giotto directors are used inside a composite actor, the period of the top most Giotto director is used, but the frequencies of the tasks inside the composite actor are relative to the frequency of the composite actor.

III. PRET Architecture

PRET [6], the Precision Timed Architecture, is a processor architecture aimed at guaranteeing timing predictability and analyzability. To do this, it replaces traditional architectural enhancements that improve average-case performance at the expense of worst-case performance with optimizations that have more predictable timing behavior. These include a hardware thread interleaved pipeline, on-chip scratchpad memories instead of caches, and instructions in its instruction set to control timing behavior. These timing instructions work as Ip and Edwards' deadline instructions [5], using a special set of registers that are decremented every cycle to specify the timing behavior of the code between timing instructions.

We feel that these features provide an ideal platform on which to implement a Giotto program. In particular, the PRET architecture's timing instructions and hardware threads provide much of the functionality required by the real-time specifications of Giotto. Using macros that wrap these timing instructions, C code can be written that includes timing instructions. In this way, we can map Giotto specification into a program that very literally implements the real-time specifications of the program, without the need for preemption or software threads.

IV. Code Generation

To generate PRET C code from a Giotto model we created a PRET specific adapter in the Ptolemy II code generation framework [2]. This includes a Giotto code generation domain and a PRET C code generation target. The C code generation framework is split into preinitialize, initialize, fire, and postfire methods, mimicking the actor abstract semantics of Ptolemy II. The code generation produces a single monolithic C file that the user can compile and run on their target. Since the PRET simulator expects a separate executable for each hardware thread, we use C preprocessor definitions to define separate implementations within the single C file generated by the adapter framework.

Our work is a central piece of the design framework shown in Figure 2. A Giotto model specified in Ptolemy II is processed by the Ptolemy II code generation framework; it creates drivers for each actor/task specified in the model and also generates code for each actor. The code generation framework produces one C file, that we compile down into the executables to run on the PRET cycle accurate simulator. We currently use hand calculated WCET times, however when automated we plan to extend this work to use the PRET WCET analysis tool currently being developed at Columbia University to fully automate the process.

PRET has no real-time operating system, so each task maps directly to a hardware thread. This allows for much less timing jitter since the overhead of task switching is much smaller. Each hardware thread has its own registers and can be treated as a parallel processor. Like other shared memory architectures, PRET features an area of memory shared among all the processors. Since we implement tight timing controls to ensure each thread accesses shared memory at the correct time, we do not need to use semaphores for synchronization. Also since each task executes at a specified rate, we do not need to use a scheduler to manage communication. We map each task seen by a Giotto Director to its own hardware thread and ensure that its inputs are read at the beginning a task's iteration and outputs are written at the end of a task's iteration.

Hierarchy in Ptolemy II enables rich heterogeneous models, but also introduces complexities to code generation. As a result we currently support code generation for a subset of the actors in Ptolemy II and we allow the user to generate code with composite actors containing Synchronous Data Flow directors as well as Giotto directors. The code generator targeting PRET currently supports a SDF director inside a modal model refinement, however we have determined a feasible mechanism to support Giotto directors inside modal models which we will implement in the near future.

The user should also note that since we map each Giotto task to its own hardware thread, they are limited to at most 6 distinct Giotto tasks being executed concurrently if they target the current PRET simulator.

A. Example with Sample Generated Code

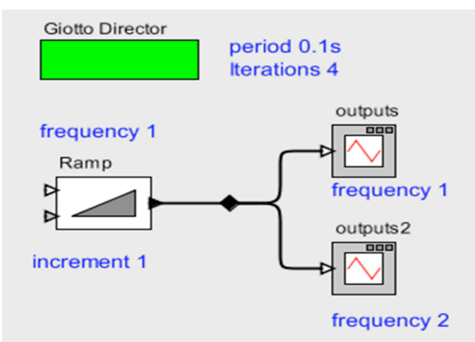

Fig. 3. A simple Giotto model in the Ptolemy II environment. The frequency annotations specify how often each actor is executed per iteration.

In Figure 3, we can see a simple Giotto model in Ptolemy II. In Listing 1 we present a snapshot of the current status of C code generation of this example to the PRET architecture. Before starting the main loop, we use a synchronization instruction to ensure that all the threads start at the same time. We convert the period of the director and the frequency of a task to processor cycles, and this is the total execution time of one iteration of that task. Conceptually, we want the input driver to run at the start of an iteration, and the output driver to run at the end, to ensure that output values are written at the end of the logical execution time of the task. In order to achieve this, we use timing instructions to delay the call to the output driver until as late as possible. This can be seen in lines 16, 27, and 38, where we specify that the following code takes an amount of time equal to the task frequency minus the WCET bound of the output driver. On lines 19, 30, 41 are the corresponding specifications that the output drivers do not take longer than their bounds.

Listing 1. Main method of the Simple Giotto Model

```
1  int main(int argc, char *argv[]) {
2      initialize();
3      jmp_buf __deadline_trying_jmpbuf__;
4      register_jmpbuf(0,&__deadline_trying_jmpbuf__);
5      if (setjmp(__deadline_trying_jmpbuf__)!=0) {
6          puts("Timing failure!\n");
7          END_SIMULATION;
8      }
9      SYNC("3F");
10     while(true){
11         #ifdef THREAD_0
12         #ifndef Simple_Ramp_OUTPUT_DRIVER_WCET
13         #warning "Simple_Ramp_OUTPUT_DRIVER_WCET was not defined."
14         #define Simple_Ramp_OUTPUT_DRIVER_WCET 1000
15         #endif
16         DEADBRANCH0(25000000-Simple_Ramp_OUTPUT_DRIVER_WCET); //period-driver_wcet
17         Simple_Ramp_driver_in(); //read inputs from ports deterministically
18         Simple_Ramp();
19         DEADBRANCH0(Simple_Ramp_OUTPUT_DRIVER_WCET); // driver_wcet
20         Simple_Ramp_driver_out(); // output values to ports deterministically
21         #endif /* THREAD_0*/
22         #ifdef THREAD_1
23         #ifndef Simple_outputs_OUTPUT_DRIVER_WCET
24         #warning "Simple_outputs_OUTPUT_DRIVER_WCET was not defined."
25         #define Simple_outputs_OUTPUT_DRIVER_WCET 1000
26         #endif
27         DEADBRANCH0(25000000-Simple_outputs_OUTPUT_DRIVER_WCET); //period-driver_wcet
28         Simple_outputs_driver_in(); //read inputs from ports deterministically
29         Simple_outputs();
30         DEADBRANCH0(Simple_outputs_OUTPUT_DRIVER_WCET); // driver_wcet
31         Simple_outputs_driver_out(); // output values to ports deterministically
32         #endif /* THREAD_1*/
33         #ifdef THREAD_2
34         #ifndef Simple_outputs2_OUTPUT_DRIVER_WCET
35         #warning "Simple_outputs2_OUTPUT_DRIVER_WCET was not defined."
36         #define Simple_outputs2_OUTPUT_DRIVER_WCET 1000
37         #endif
38         DEADBRANCH0(12500000-Simple_outputs2_OUTPUT_DRIVER_WCET);
39         Simple_outputs2_driver_in(); //read inputs from ports deterministically
40         Simple_outputs2();
41         DEADBRANCH0(Simple_outputs2_OUTPUT_DRIVER_WCET); // driver_wcet
42         Simple_outputs2_driver_out(); //output values to ports deterministically
43         #endif /* THREAD_2*/
44     }
45     exit(0);
46 }
```

Theorem 1: Let A and B be actors with an output of A connected to an input of B. Using our code generation algorithm, if no exception is raised at runtime, then the following conditions are true.

(a) There is no write/write hazard.
(b) There is no write/read hazard.
(c) Let g be the greatest common divisor of the periods of A and B, $WCET_{A_{out}}$ to be the provided bound for A's output driver, and $EXEC_{B_{in}}$ to be the execution time of B's input driver. If $EXEC_{B_{in}} + WCET_{A_{out}} < g$, then there is no read/write hazard.

Proof:

(a) Since every global memory location has only a single writer, this is trivially true.
(b) The pathological ordering for a write/read hazard is when the iterations of A and B end in the same cycle. Since the ordering of the threads is arbitrary, B may be earlier in the pipeline than A and start its next iteration first. But this only means that the timing instruction of A will be simultaneous with the first instruction of B. Since the memory load instruction is no longer in the pipeline and PRET memory accesses are blocking, no write/read hazard occurs.
(c) The minimal possible positive interval between the start of an iteration of B and the end of an iteration of A is g cycles. A's output driver starts $WCET_{A_{out}}$ cycles before the end of A's iteration, and B's input driver finishes $EXEC_{B_{in}}$ cycles after the start of the iteration of B. Thus if $EXEC_{B_{in}} + WCET_{A_{out}} < g$, by similar

reasoning to case (b), we can show that no read/write hazard occurs.

■

Note that sufficient conditions to prevent a read/write hazard in case (c) only depend on knowing an execution time bound on the input driver, the output driver, and the greatest common divisor of the periods. If we required the user to specify a bound on the input driver in addition to the output driver, this condition could be checked at compile time.

B. *Implementation*

In order to make sure that communication between tasks takes place at the proper time with respect to Giotto semantics, we have added separate methods called drivers responsible for communication. These drivers that are responsible for reading the inputs and writing outputs to and from global memory locations are called *input drivers* and *output drivers* respectively. We make sure that these drivers execute at the correct times by including timing instructions that bind the time at which the drivers run. In particular, we start the input drivers at the beginning of each iteration of an actor and delay the writing of outputs to the end of the iteration.

In order to make sure that the output writing takes place as late as possible, we delay by a time equal to the period of the actor minus the worst-case execution time of the output driver. This worst-case execution time bound is not known at the time the Giotto model generates its code, so it is parametrized as a C preprocessor define. This allows a user to use a separate tool to calculate a worst-case execution time bound of the output driver after the C code has been generated without having to then return to the Giotto model and regenerate the C code. In the case that no value is defined, we have provided a default value and a compiler warning. This is only to allow the generated code to be immediately compilable, and does not mean that a user should depend on this value in the deployment to the final PRET target.

To ensure that all deadlines are met, we include exception code that detects missed deadlines and displays an error. Since Giotto does not specify behavior in case timing constraints are missed, we consider all missed deadlines fatal and end the simulation. In this respect, our deadline detection mechanism provides support for testing that deadlines will be met, but not deployment-time support for recovering from missing deadlines.

V. Application

To demonstrate the use of the C code generation for Giotto models we generated the controller for a simplified toy elevator controller shown in Figure 4. Since the elevator only serves two floors the control algorithm is fairly straightforward, but it is simple nonetheless. Riders can call the elevator from either floor or select a destination floor, and the controller opens and closes the doors and moves between floors. Since we target the PRET simulator we use a sequence actor in Ptolemy II to generate the inputs to the elevator controller and use the EmbeddedCActor to generate code we display on the screen during a run of the simulator.

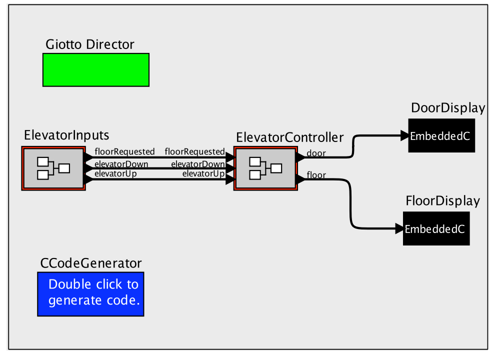

Fig. 4. Top level structure of two story elevator controller.

In order to find appropriate execution time bounds for the output drivers, we use existing knowledge of the timing of the PRET simulator to choose reasonable values. The simple nature of the generated C code along with the exception mechanism allow us to verify that runs of the control program meet their execution time bounds. In more complicated or resource constrained situations, one may prefer to perform more formal worst-case execution time analyses to produce higher confidence bounds.

VI. Conclusion

Giotto is a useful and intuitive programming model for the PRET architecture. We have built an extension to the existing Ptolemy II code generator to target the PRET processor. It compiles Giotto models into C programs with explicit deadlines that establishes precise timing coordination between execution threads. This is made possible because of the precise-time control provided by the underlying hardware. Along with the ability to synchronize execution, we employ PRET's mechanisms to throw a fatal runtime exception in cases when the deadlines of the Giotto model cannot be met. We provide the possibilities of doing both static and run-time checking for execution safety.

References

[1] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, Jan 2003.

[2] M.-K. L. Gang Zhou and E. A. Lee. A code generation framework for actor-oriented models with partial evaluation. In *International Conference on Embedded Software and Systems*, LNCS 4523, pages pp. 786–799, May 2007.

[3] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. volume 91(1) of *Proceedings of the IEEE*, pages 84–99, 2003.

[4] T. A. Henzinger, C. M. Kirsch, and S. Matic. Schedule-carrying code. In *In Proc. EMSOFT, LNCS 2855*, pages 241–256. Springer, 2003.

[5] N. J. H. Ip and S. A. Edwards. A processor extension for cycle-accurate real-time software. In *Proceedings of the IFIP International Conference on Embedded and Ubiquitous Computing (EUC)*, volume 4096, pages 449–458, Seoul, Korea, Aug. 2006.

[6] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee. Predictable Programming on a Precision Timed Architecture. Proceedings of International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES), 2008.