Code Generation for PTIDES Models

Jia Zou, Isaac Liu, Jeff C. Jensen
EE290N Class Project Report

***Abstract*—PTIDES is a programming model for distributed real-time embedded systems. PTIDES builds on Discrete-Event (DE) semantics and leverages a global notion of time throughout distributed platforms. Applications expressed in PTIDES models can be designed without any knowledge about the hardware platform upon which it is implemented, thus completely decoupling the design from the implementation. To bridge this gap, a code generator is implemented to automatically generate platform-specific C code from PTIDES models designed in Ptolemy II [1] to a particular hardware platform. This paper present our approach in generating C code from PTIDES models, and statically link them against the PtidyOS scheduling libraries to produce a real-time executable we call PtidyOS.**

I. Introduction

PTIDES (Programming Temporally Integrated Distributed Embedded Systems) is a programming model for distributed real-time embedded systems that provides an abstract framework for distributed real-time applications. The purpose of PTIDES is to enable programmers to efficiently develop correct, deterministic programs. To meet the constraints of real-time systems, we additionally require that the time at which the output signals are generated is also deterministic, *i.e.* for any two executions of the same PTIDES model and input signals, the applicaiton will generate the same output signals.

Designing and modeling applications in high-level models of computation such as PTIDES allows for robust design and easier analysis of concurrent software, and prevents many of the common pitfalls introduced though lower-level programming models such as threading. One of the key benefits of the PTIDES programming model is its inclusion of timing semantics, so that the user is able to design an application without any knowledge of the hardware platform upon which it is implemented, with the guarantee of deterministic behavior. A PTIDES program may be designed within a high-level programming environment such as Ptolemy II. The design can be verified and tested through simulation until it satisfies all functionality and timing constraints. This design can then be mapped to a specific embedded target platform. As Fig. 2 shows, to bridge this gap between design and implementation, a code generator is ideal, so that the design can be automatically generated into an executable for a specific platform. We call this executable PtidyOS.

Our code generator relies on and extends the current code generation framework in Ptolemy II. Ptolemy II models are composed of directors (which govern the semantics of the model), and actors (which govern its behavior). In our case, the PTIDES Embedded director embodies the semantics of a PTIDES model, both for simulation and for C code generation. In simulation, actors define input and output behaviors, and are composed according to the rules of the director. In C code generation, Ptolemy II uses *helpers* (adapters) for each hardware platform and actor. In this paper, we present Ptolemy II actors written for the PTIDES Embedded domain, as well as the C code generator helpers that target our embedded platform, the Luminary Micro LM3s8962.

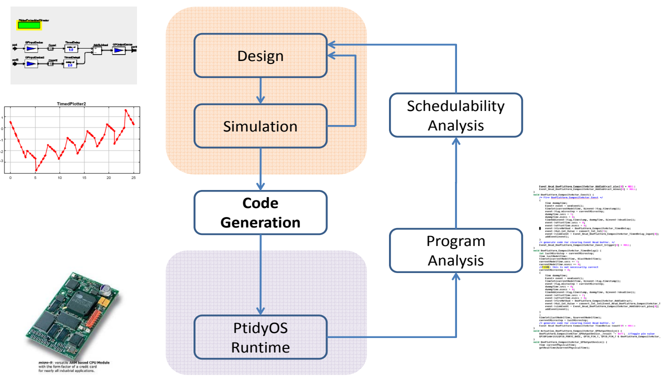


Fig. 1. PTIDES Programming Workflow

A. PTIDES Programming Model

PTIDES builds on the Discrete Event (DE) model of computation, an event-triggered model where software components communicate via timestamped events. We refer to the values of these timestamps as *model time*. Physical time, by contrast, is time in the physical world when a sensor event is received or an actuation event is set to occur. Physical time is totally ordered, while model time need only be partially ordered. The scheduler may schedule an event with later timestamp before processing one of an earlier timestamp, as long as no actor observes any violation of causality with respect to physical time. Model time and physical time are bound at the interfaces of sensors and actuators, as defined in [2]. Using this relationship, as well as causality interface as defined in [7], PTIDES specifies a safe-to-process algorithm [8] which determines if an event may be safely used in computations or sent for actuation. In the most simple case, an event is safe-to-process if its timestamp, when added to a constant model time delay (or *offset*), is greater than or equal to the current physical time. This *offset* is specified for each input port of the system at design time. Input ports are inputs which are connected to external systems on the same hardware environment, external sensors, or network devices. These offsets may be calculated using causality information of events within the system, physical response time of hardware IO devices, or network delay and synchronization error, respectively. In our current implementation these offsets are annotated by the user in the Ptolemy II model, though automated analysis of the model may be achieved in the future.

B. PtidyOS Real-Time Operating System

The final product of our code generation is PtidyOS. One way to think about PtidyOS is that it is a collection of software

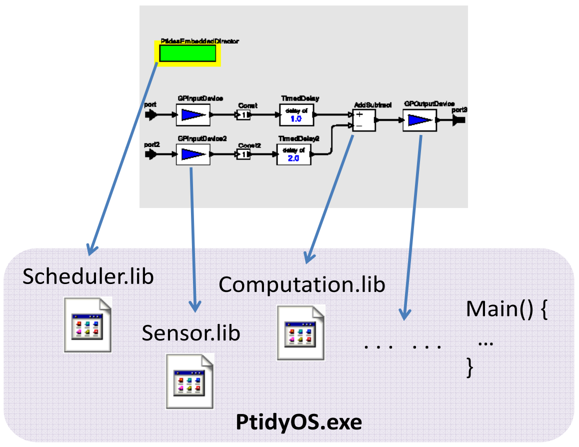


Fig. 2. PtidyOS

libraries. As Fig. 2 shows, during code generation, we produce a scheduling library that implements PTIDES semantics and dictates the scheduling of the system. Each of the actors in the system can also be viewed as a software component library. Thus the code generation framework glues together all the libraries into one executable, which we call PtidyOS.

In this regard, PtidyOS is very different from general embedded operating systems such as OpenRTOS [6], vx-Works [4], and RTLinux [5]. Unlike these conventional RTOS's that leverage a threaded programming model, PtidyOS instead performs all event processing and context switching in interrupt service routines. The use of interrupt service routines minimizes latency between software components and dramatically reduces the number of context switches, as context switching occurs only when an incoming sensor event is received, when an event becomes safe-to-process, or when an actuation event has been scheduled in accordance with its predetermined deadline. To ensure the most time-critical event is always executed first, PtidyOS relies on reentrant interrupts that may interrupt both themselves or other interrupt service routines so that higher-priority events take precedence. PtidyOS leverages the event-order execution semantics of PTIDES with traditional scheduling methods such as earliest deadline first (EDF). EDF guarantees optimal scheduling (when a feasible schedule exists), and when combined with PTIDES semantics, allows for correct execution without the need to totally order event processing. Finally, PtidyOS tries to achieve a small memory footprint in part through this code generation, which performs optimization through partial evaluation of the application model.

C. *Related Work*

PTIDES is not the first to introduce timed semantics into a model of computation. Giotto [3] does so by introducing "logical execution time"; both Giotto and PTIDES limit the interaction between non-physical and physical time. PTIDES is more explicit in that only sensors and actuators bind model time to physical time, while in Giotto logical execution time can relate to physical time at all components. Furthermore, PTIDES takes an event triggered approach instead of exclusively focusing on periodic tasks.

PtidyOS solves traditional real-time scheduling problems, but differs from conventional real-time operating systems in its explicitly timed semantics and reentrant interrupt context switching. PtidyOS, as its name indicates, share somes resemblance to TinyOS, a light-weight operating system used mainly in small-scale sensors. For example, Distributed controls systems benefit from the light footprint of PtidyOS. Both TinyOS and PtidyOS employ a single stacked memory scheme, which simplifies memory management and provides more predictable timing behaviors by simplifying memory use.

II. Implementation

A. *PtidyOS Application Anatomy*

Since a pilot implementation of PtidyOS was done in C, it was used as a basis for our generated code. Moreover, much of the code incorporates the scheduler that implements PTIDES semantics, which could be simply copied as the core of the scheduler library. To efficiently identify and reuse the available function in the pilot code, the functions in PtidyOS are partitioned into "static" code and "generated" code.

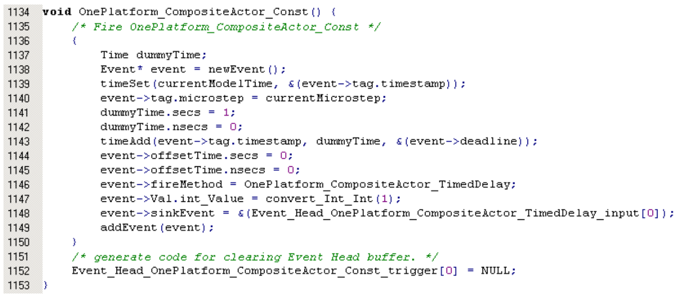
"Static" code refers to core PtidyOS components such as scheduling methods, stack manipulation, and utility functions. Examples of these functions are processEvents(), disableInterrupts(), and convertCyclesToNsecs(). These functions are static in the sense they do not need to be partially evaluated, and only one instance of each function is necessary for any PtidyOS application. In the Ptolemy II code generation infrastructure, these functions are generated in the "sharedBlock" region, which is generated only once for any given target. notice this code can be platform independent (e.g. scheduling methods) or platform dependent (e.g. interrupt enable/disable), but they are generated together through platform independent and dependent adapters that were created for Ptides Embedded director.

In order to correctly implement a PTIDES semantics, the scheduler needs to access causality information about the PTIDES model. Since one of our goals for PtidyOS is to minimize its footprint, a full duplicate representation of the model in PtidyOS is undesirable. Thus the causality information about this model is preprocessed, and as we will show later, they are stored as a part of *Events*, which is accessed when a particular event is processed.

Finally, "generated" code refers to methods specific to actors in a model, which mainly include the fire() functions for each actor. At the end of firing of each actor, code is also generated for the production of an event, which we will discuss in detail in the next subsection.

B. *Code generation Infrastructure*

When code generating an implementation from a PTIDES model, the structure of the model must be preserved. In the pilot PtidyOS implementation, the structure of the model is created by *Actor* and *Port* data structures, which are used to construct the model at program initialization time. However, because we are leveraging the Ptolemy II code generation framework, we can use partial evaluation before we generate the code to save the code space and memory overhead of storing the model. The structure of the model is static, so the destination actor of an event will never change. Since our code generator has access to this static model, when events are generated by each actor, the destination of the event is known.



```
void OnePlatform_CompositeActor_Const() {
    /* Fire OnePlatform_CompositeActor_Const */
    {
        Time dummyTime;
        Event* event = newEvent();
        timeSet(currentModelTime, &(event->tag.timestamp));
        event->tag.microstep = currentMicrostep;
        dummyTime.secs = 1;
        dummyTime.nsecs = 0;
        timeAdd(event->tag.timestamp, dummyTime, &(event->deadline));
        event->offsetTime.secs = 0;
        event->offsetTime.nsecs = 0;
        event->fireMethod = OnePlatform_CompositeActor_TimedDelay;
        event->Val.int_Value = convert_Int_Int(1);
        event->sinkEvent = &(Event_Head_OnePlatform_CompositeActor_TimedDelay_input[0]);
        addEvent(event);
    }
    /* generate code for clearing Event Head buffer. */
    Event_Head_OnePlatform_CompositeActor_Const_trigger[0] = NULL;
}
```

Fig. 3. Event Generation Code

Thus, we do not need to generate and store this model at run time, but instead, we directly code generate the firing code of an actor to place its output event at the known destination actor input. In our code generation framework, actors are simply generated to be a single function in the c code, which we call the *firing function*. This *firing function* will contain the semantics of the actor, which reads from its inputs ports and produces outputs to its output port. The structure that carries the data passed around by actors are called *Events*.

An *Event* contains a place holder for the data value that's being passed around, as well as a timestamp to signify the model time it was created. So far this is the same as in a discrete event system. However, leveraging the partial evaluation of Ptolemy II, we can store the destination actor (firing function) of this event in this *Event* structure, as well as its destination port. By doing so, we avoid the need to store any information about the model at run time in the generated implementation. When an event is processed by the scheduler, it simply looks in its own structure, and calls the function stored in its destination firing function field. This is illustrated by the code snippet in Fig. 3, which is the *fire function* of a Const actor. This actor simply produces a constant value each time it is triggered.

Since the computation it performs is trivial, the entire fire function is basically the code that produces an output event. As it shows, an event is first created by calling newEvent(). Notice since no dynamic allocation of event is allowed, this method simply takes an event from a repository of event structures that are allocated at initialization. The timestamp and microstep of the event is then set to the currentModelTime and currentMicrostep, which keep track of the current model time of the system. The deadline and the event as well as its offset are then set. Notice these values are preprocessed by the code generator to reflect the causality of this event with respect to the rest of the system. This is followed by a function pointer that points to the fire method of the destination actor, as well as the data value of the event. sinkEvent, which is a pointer to a pointer of an event, represents the port this event is destined to. When this event is processed, the scheduler will use this pointer to ensure the input channel slot points to this event. Next, addEvent() is called to add this event into the event queue, which completes the event generation process. Notice at the end of the firing function, the input channel slot of this actor is cleared, to indicate that the input event has been consumed.

C. Communication between actors

For dataflow like models of computation, for example, Synchronous Dataflow or Process Network, the communication channels from the model is simply a First In First Out (FIFO) buffer. The output actor generates tokens to be stored in the communication buffers while the destination actor simply reads from the buffer when it is fired. Since each firing of a dataflow actor is based upon a firing rule, whenever its fire function is executed, it knows that there will always be tokens in its input. However, for PTIDES, which is based on the Discrete Event model of computation, each firing of an actor is not dependent on a firing rule. Instead, PTIDES actors are fired whenever an event arrives at any of its input ports, and the actor only consumes the token(s) with the smallest timestamp at each firing. Thus, even if we have communication buffers for each connection, the firing function would still need to determine which events to consume based on the timestamp of the events, instead of simply taking an event off each of the buffers of its incoming connections. Along with that, PTIDES contains an event queue which orders the events to determine which one to process next. With the event queue storing the processing order of events, and the events themselves containing the destination firing function and port, we no longer need to generate a buffer for each connection between the actors.

We create an empty *Event* slot for each channel on each input port for an actor. When events are taken off the event queue for execution at a particular timestamp by the PTIDES scheduler, the scheduler points the destination input channel slot to the event that is currently being processed, and then it calls the firing function for the destination of the event. The firing function now simply checks each input slot to see if an event is present, and generates the correct output based upon the inputs its received. The event that is produced by each firing is then simply added onto the event queue, which is self sorted by the deadline of the event. By doing so, we avoid the need to generate lengthy buffers for each connection between actors.

D. Bridging physical time and model time

The fundamental goal of PTIDES is that it bridges the model and the physical world. It does so with sensor and actuator actors. A sensor is an actor which fires only when triggered by the physical world. An actuator is an actor that when fired, produces an actuation on the physical environment. The deterministic time at which the actuation occurs is the fundamental property of PTIDES. Thus, special care needs to be taken when generating code for sensors and actuators.

For sensors, its generated fire function is always registered as an interrupt service routine for the specific device its trying to model. For example, if the sensor actor represents a general purpose input pin, then its fire function will be registered as the interrupt service routine of the specific general purpose input pin. The fire function of the sensor does the following:

1) Get the current physical time
2) Reads in the value of the sensor
3) Generates an event that encapsulates the sensor value and sets the timestamp of the event to be the physical time obtained in step 1
4) Add the event to the event queue

Thus, whenever a sensor actor is triggered, it generates an event to be put into the system with its timestamp equal to physical time. As this event is propagated through the system, its timestamp is manipulated and increased by other actors in the system, such as *TimeDelay* actors. When it finally triggers an actuator, the physical time of the actuation should occur at the timestamp specified by the event.

Thus, for actuators, two separate firing functions are generated. The *firing function* of an actuator is the function which is called when the scheduler schedules an event to be run on the actuator actor. The *actuation function* is the function that actually does the physical actuation through the output device. The *firing function* of the actuator does the following:

1) Get the current physical time
2) Reads the timestamp of the event which triggered this firing
3) Sets up a timer with the difference between physical time and the timestamp of the event

The *actuation function* is simply the timer interrupt service routine that is run when the timer expires. This ensures that the physical actuation from the system occurs when physical time equals the timestamp of the event.

E. Platform Dependency

The goal of the code generation infrastructure is to generate the same model semantics for different target platforms. The pilot version of PtidyOS targets the Luminary Micro LM3s8962, an ARM Cortex-M3 based embedded platform. Thus, the partitioning of generating platform dependent and independent code had to be designed into the code generation framework. Ptolemy II's code generation framework already provides a solid framework which to extend: Platform independent code is generated from Ptolemy II by a PTIDES domain adapter, while platform-dependent code is generated from a subadapter which is architecture specific.

Platform independent code includes most of the scheduling libraries of PtidyOS, and the structures required for the basic PTIDES framework. This includes the event queue, the *Event* structure, and the functions which manipulate the event queue to schedule events. Typically the structure of the model would also be platform independent code, but in this case the structure of the model is optimized away by partial evaluation, thus we don't need generate those.

For sensors, actuators, and any device configuration code, they are generated in the subadapter for the specific target. This is done through the code template theme of the code generation framework in Ptolemy II. Platform specific actors, such as general purpose input/output pins or speakers, can be created by writing its platform specific code in the code blocks of code templates. The subadapter for the PITDES director will then place specific code blocks in the correct location in the code, including initializing the device, and the code to use the device.

By doing so, we can extend the code generation framework for another target by simply writing different code templates for the different targets. This allows us to have a platform independent code generator.

III. Example PtidyOS Application

We present a simple example of a code-generated PtidyOS application, as shown in Fig. 4. The model, built in Ptolemy II with the PtidesEmbedded director, receives an external input signal, converts the signal into a note of a song, and sends the note to an actuator (in this case a speaker) with an annotated physical delay.

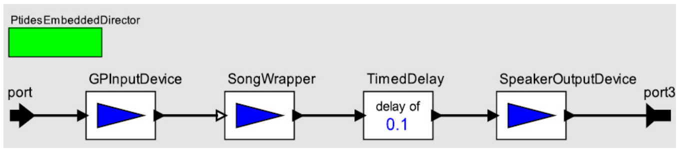


Fig. 4. PTIDES Example Application

Given a periodic input signal, a correct execution of this model will play sequential notes in a song on a regular beat. An observer can easily detect beat frequencies if notes are played irregularly, missed notes if deadlines are not met, and incorrect playback if tasks are executed out of timestamp order.

IV. Results and Conclusion

In this project, we augmented the code-generation framework of Ptolemy II with the capability of generating PTIDES models into embedded C applications. PTIDES semantics are embodied by the PtidyOS scheduling and context-switching libraries, and PTIDES models are statically linked against these libraries to define the behavior of a model. Unlike a typical threaded operating system scheduler, PtidyOS benefits from the partial-evaluation [1] steps of code generation which statically analyze causality and dependency relationships and converts them into predetermined firing paths. Constructing PTIDES models in Ptolemy II allows for simplified design, evaluation, and verification through simulation without concern for low-level design considerations such as memory management, mutual exclusion locks, or thread priorities. When satisfied with the design, the code generation framework allows for fast prototyping and implementation of the PTIDES model on varied target platforms.

We plan to continue development of PtidyOS by adding additional target platforms, implementing more sophisticated PTIDES scheduling libraries, improving system performance, minimizing code footprint, and supporting a more complete subset of Ptolemy II actors.

References

[1] C. Brooks, C. Cheng, T. H. Feng, E. A. Lee, and R. von Hanxleden. Model engineering using multimodeling. In *1st International Workshop on Model Co-Evolution and Consistency Management (MCCM '08)*, September 2008.

[2] T. H. Feng and E. A. Lee. Real-time distributed discrete-event execution with fault tolerance. In *Proceedings of RTAS*, St. Louis, MO, USA, April 2008.

[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] W. River. Vxworks: Embedded rtos with suport for posix and smp. http://www.windriver.com/products/vxworks/.

[5] W. R. RTLInuxFree. Rtlinuxfree. http://www.rtlinuxfree.com/.

[6] The FreeRTOS.org Project. Openrtos. http://www.freertos.org.

[7] Y. Zhou. *Interface Theories for Causality Analysis in Actor Networks*. PhD thesis, EECS Department, University of California, Berkeley, May 2007.

[8] J. Zou, S. Matic, E. A. Lee, T. H. Feng, and P. Derler. Execution strategies for ptides, a programming model for distributed embedded systems. In *to appear in RTAS*, 2009.