# Deterministic Execution of Ptides Programs

*Patricia Derler*
*Eidson John*
*Goose Stuart*
*Edward A. Lee*
*Michael Zimmer*

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2013-65
http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-65.html

May 15, 2013

Acknowledgement

This work was supported in part by the iCyPhy Research Center (Industrial Cyber-Physical Systems, supported by IBM and United Technologies), and the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley (supported by the National Science Foundation, NSF awards #0720882 (CSR-EHS: PRET) and #0931843 (ActionWebs), the Naval Research Laboratory (NRL #N0013-12-1-G015), and the following companies: Bosch, National Instruments, and Toyota).

Special thanks go to Siemens for a grant and to National Semiconductor, XMOS, Renesas and National Instruments for equipment grants.

# Deterministic Execution of Ptides Programs

Patricia Derler, John C. Eidson, Stuart Goose, Edward A. Lee, Slobodan Matic, and Michael Zimmer

May 15, 2013

**Abstract**

This paper discusses the use of the Ptides model of computation as a coordination language for the design of deterministic, event-driven, real-time, distributed embedded systems. Specifically, the paper shows how the use of synchronized clocks in the context of Ptides enables explicit, platform independent specification of functionality and timing. From this specification, we generate code for two target platforms: Renesas and XMOS. The generated code includes a lightweight operating system which performs scheduling, I/O and network handling as well as application specific tasks.

Ptides models are developed in Ptolemy, a design and simulation environment for heterogeneous systems. This framework also contains a code generation framework which is leveraged to derive Ptides implementations from the models. We illustrate our approach by designing a simple Ptides application, a small component in a printing press responsible for on-the-fly changeover between paper rolls. We demonstrate the design process and show that the generated code exhibits identical timing at the cyber-physical boundary on multiple implementation platforms. [1]

## 1 Introduction

This paper discusses the use of the Ptides model of computation (MoC) [1] as a coordination language [2] for the design of deterministic, event-driven, real-time, distributed embedded systems. Such systems are common in industrial and commercial systems and typically include sensors, actuators and computing elements, often on different platforms, communicating via a network. The timing of sensing and actuation is often critical when the system is used to control or monitor external processes. Timing requirements for accuracy and precision are very application dependent and range from milliseconds for many devices, microseconds for high speed machinery and telecommunications, and nanoseconds or better for some military and scientific applications.

We illustrate how the Ptides methodology allows the system or device designer to explicitly specify both the functional and the external timing properties of a device or system in a platform independent manner. We also show how, with simple constraints, synchronized clocks with additional hardware support allow enforcement of highly accurate and precise timing constraints that

[1] This work was supported in part by the iCyPhy Research Center (Industrial Cyber-Physical Systems, supported by IBM and United Technologies), and the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley (supported by the National Science Foundation, NSF awards #0720882 (CSR-EHS: PRET) and #0931843 (ActionWebs), the Naval Research Laboratory (NRL #N0013-12-1-G015), and the following companies: Bosch, National Instruments, and Toyota).

Special thanks go to Siemens for a grant and to National Semiconductor, XMOS, Renesas and National Instruments for equipment grants.

are independent of the execution times of the embedded code. These characteristics should alleviate current problems in maintaining interfaces, especially timing interfaces, under code updates in devices or device upgrades in systems. We extend previous work [3] by reporting experimental data illustrating the properties of this approach and discussing the role of supporting hardware. Modeling, simulation and code generation is implemented in Ptolemy II, [4], an academic tool for designing and experimenting with heterogeneous system models.

The first section of this paper provides a basic understanding of the Ptolemy II system and the Ptides extension. The second section discusses the basics of Ptolemy II, the Ptides model and describes how to construct applications. The third section discusses the use of Ptides as well as the design and execution environment. The fourth section illustrates the process via a simple controller design, and an experimental verification of the timing. The paper concludes with a summary and a brief outlook on future work.

# 2 Ptolemy II and Ptides basics

Ptolemy II is a modeling and simulation framework for concurrent models of computation (MoC) and heterogeneous mixtures of MoCs. MoCs that are implemented in Ptolemy include discrete-event (DE), continuous time (CT), synchronous reactive, dataflow and process networks. We implement Ptides in Ptolemy as a DE-based MoC. CT is typically used to represent physical processes. This allows for co-simulation of plant models with Ptides designed control systems to study the functional and timing properties of the entire system. A DE model [5] was selected for Ptides because DE systems executing tokens in timestamp order have the property that given a set of input tokens, all correct DE implementations will produce identical sets of output tokens [6].

## 2.1 Actor-oriented design

Ptolemy follows the actor-oriented design principle [7] and graphically represents models where actors appear to a designer using the Ptolemy graphical editor as boxes with ports as illustrated in the center panel of Figure 1.

All communication between actors is via tokens passed between ports. In a Ptides design these tokens are a (value, timestamp, microstep) triple. The (timestamp, microstep) pair models a superdense formulation of time where the timestamp represents model time and the microstep allows ordering of events with identical values of the model time [8].

The model of computation used by a model is determined by the *Director*[2] component which governs the Ptolemy simulation engine as well as specifying the code generation MoC, in this case DE. In this model the *DiscreteClock* actor generates periodic events with tokens (1,0,1) and (0,100,1) repeating with a period of 200 so that the initial (time 0) set point is 1 which then changes to 0 at time 100.

Ptolemy models can be hierarchical as illustrated by the top and bottom panels. The top panel is a CT model of the *Plant Model* actor and implements the integral equation $(RC)\times T = \int (D-T)dt$ where $T$ is the *TachometerOutput* variable and $D$ is the *DriveInput* variable. For constant $D = D_0$ the solution is $T = D_0(1 - e^{-t/RC})$. The *ZeroOrderHold* and *PeriodicSampler* actors are required since the tokens representing the variables $D$ and $T$ cross the boundary between the CT and DE

[2] *Italic* fonts indicate names of artifacts in the figures.

domain in the hierarchy. The symbol labeled *RC 10.0* specifies the parameter RC with a value of 10. The other actors have their obvious interpretation.

The bottom panel illustrates the model for the controller. In this case the *PtidesDirector* actor specifies that Ptides semantics, a specialization of DE semantics, are enforced. Ptides semantics are embodied in five special actors:

- Sensor ports are illustrated by the *SetPoint* and *Tachometer* ports in the bottom panel of Figure 1. A Ptides sensor port takes the value of the sensor input, $S_1$, at time, $t_1$, and places an event token ($S_1$,$t_1$,1) on the event queue. When this model is executed using the Ptolemy simulator, the time $t_1$ is the sample time modeled by the top level DE director and represents physical time in the simulation. In code generated from this model, $t_1$ is in the frame of reference of the platform's real-time clock.

- Actuator ports are illustrated by the *SetPointMonitor* and *Drive* ports. A Ptides actuator port takes an input token ($A_2$,$t_2$,i) at its input and instantiates the value $A_2$ at the actuator output at a time $t_2$ where $t_2$ is the time in the frame of reference of the top level DE director in simulation and of the platform's real-time clock in a physical implementation.

- A network output port, (not illustrated in Figure 1), takes an input token ($V_3$,$t_3$,i) from its input and embeds it in a message on the external network at a time $t_{no}$ in the frame of reference of the top level DE director in simulation and of the platform's real-time clock in a physical implementation. $t_{no}$ is a parameter of the port and allows the designer to specify the value as a time offset from the time of origin of upstream events, typically a sensor input port.

- A network input port, (not illustrated in Figure 1), extracts the token ($V_3$,$t_3$,i) from the message received from the external network and places it on the local event queue at a time $t_{ni}$. The offset between $t_{no}$ on the sending platform and $t_{ni}$ is a function of the network protocol and typically the network traffic load.

- Time delay actors are illustrated by the *TimeDelay* and *TimeDelay2* actors. A Ptides time delay actor takes an input token ($V_4$,$t_4$,i) at its input and outputs a token ($V_4$,$t_4 + t_{delay}$,i). The value of $t_{delay}$ can be set as a parameter or taken from the bottom input port of the actor. This actor permits the designer to specify the time delay between sensor inputs and actuator outputs. For example, in Figure 1 a sensor reading at time $t_s$ at the *Tachometer* input results in an actuation by the *Drive* actuator at a time $t_s + 0.001$ where both the times $t_s$ and $t_s + 0.001$ are in the frame of reference of the top level DE director in simulation and of the platform's real-time clock in a physical implementation.

### 2.2 Timing considerations

It is easy to specify a system that cannot meet its timing constraints. In the example of Figure 1 if $t_{delay} = 0$ then, while the execution can be simulated, the system cannot be executed in a real system since the computations take a finite amount of time. However, provided the specified model delays, e.g. using the time delay actors, are greater than the execution times on a particular platform, then DE semantics, simple constraints on actor temporal semantics, and the properties of the sensor and actuator ports ensure that the external timing meets the specification to within

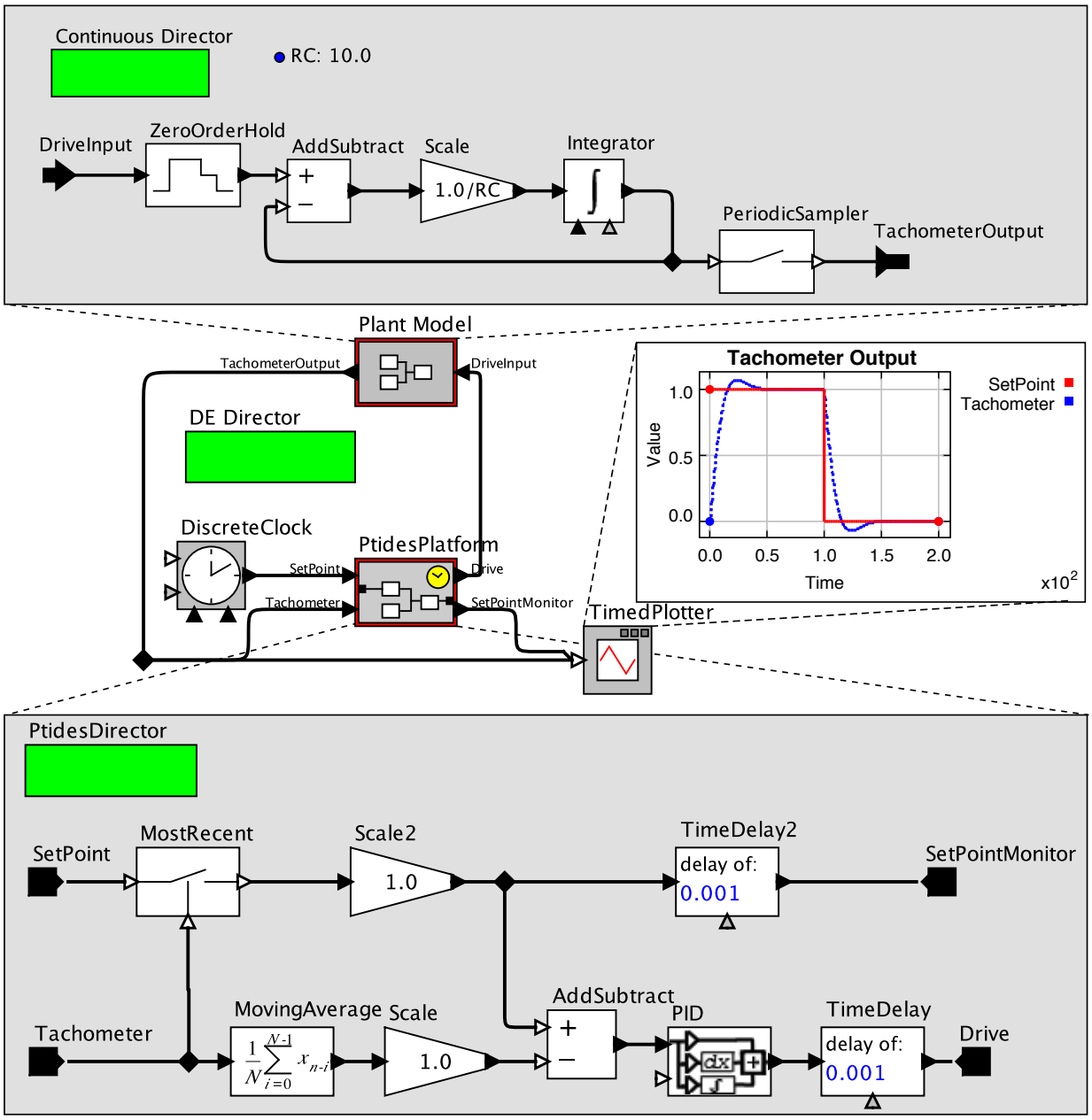


Figure 1: Model of a simple control system.

the accuracy permitted by the implementation of these ports irrespective of the actual execution time [9]. This is extremely important, since changes, such as improving algorithm performance or using a faster processor, do not change the external timing performance of the device.

If the sensor input and actuator output are on different platforms communicating via a network, the network delay must be bounded and included in the specification of end-to-end delay to ensure that timing specifications are met. In networked systems, the real-time clocks in the various platforms must be synchronized to the necessary degree of accuracy, for example via IEEE 1588

[10], since the execution clearly depends on all timestamps at platform inputs and outputs using a common time reference.

As noted, the timing accuracy depends on the implementation of the Ptides input and output ports. Logically, sensor and actuator ports appear as illustrated in Figure 2. For example, for an analog sensor, the sample and hold trigger is generally used as the *sensor indicator* that latches the time, the *sensor timestamp*, from the *real-time clock* into the *sensor timing register*. Likewise, an *actuation timestamp* placed in the *actuator timing register* by Ptides generated code is compared to the *real-time clock*, and when the times match, the *actuation trigger* is generated. If these functions are implemented in hardware, e.g. on an FPGA, the time accuracy will typically correspond to the LSB of the clock. If implemented by code in a microprocessor, the accuracy will be determined by interrupt latencies, and execution time considerations. Some of these considerations can be addressed by using architectures such as XMOS [11] or PRET [12] where execution times are predictable.

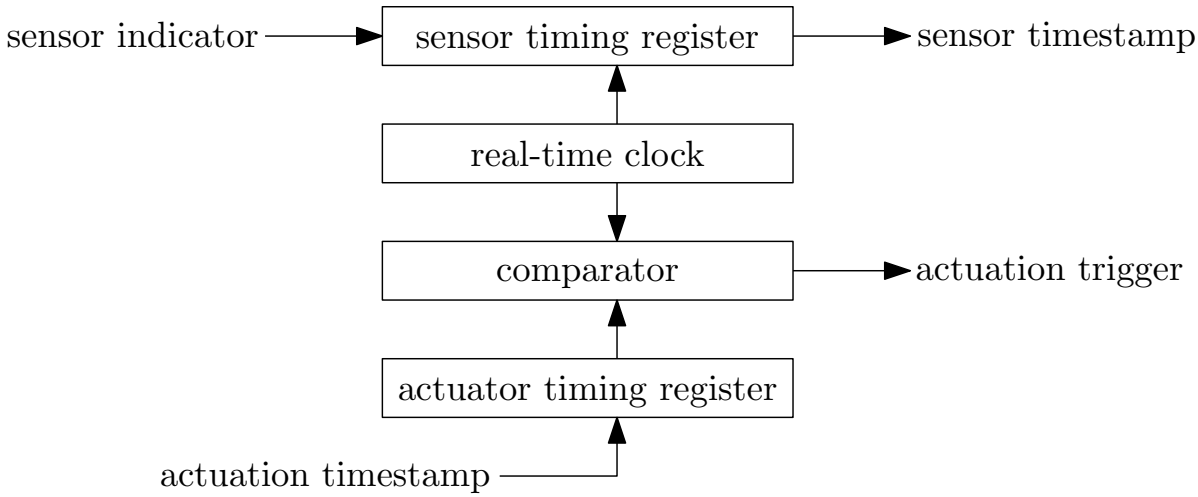


Figure 2: Logical model of sensor and actuator ports

### 2.3 Safe-to-process timing

A final important execution issue in correctly executing DE semantics in real time is illustrated in Figure 3. Suppose a token, $(S_B, 30, 1)$, appears on port $b$ of the *AddSubtract* adder. When can the adder execute given the DE requirement that tokens must be processed in timestamp order? Any event from *sensorC* on *platformA* with a timestamp $\leq 30$ will already be in the event queue for port $c$. Consider an event with timestamp 30 generated at *sensorA* on *platformD*. The *sensorAnetwork* output port on *platformD* has a *platform delay bound* value of 1 ensuring that this event will be placed on the network no later than time $30 + 1$ relative to the platformD's clock. Assume the network transit time is bounded by a value of 5. Hence the event $(S_A, 30, 1)$ must arrive at the *sensorAnetwork* input port and be placed on the event queue of *platformA* at a time no later than $30+1+5 = 36$. Assume the maximum clock synchronization error is $\epsilon$. Therefore, the *AddSubtract* adder must delay processing the event $(S_B, 30, 1)$ on port $b$ until at least time $36 + \epsilon$ relative to the clock on *platformA*. In analyzing a design for these safe to process values, accurate values

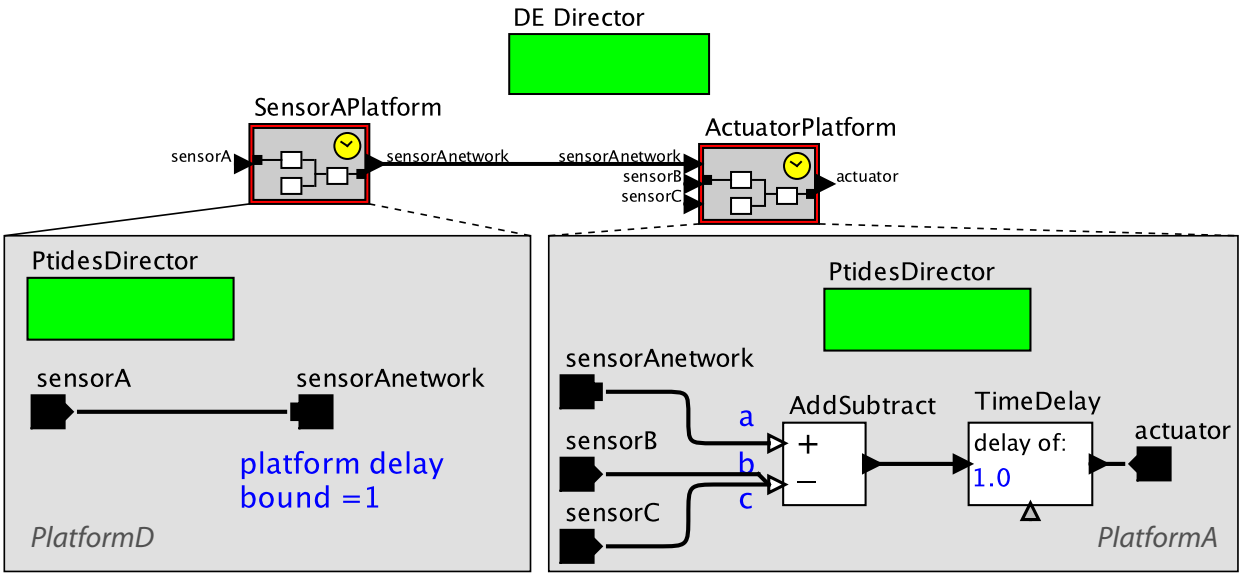


Figure 3: Safe to process example

for network delays and $\epsilon$ are required. Note that an event with a timestamp $< 30$ arriving after the computation can be detected and appropriate application specific action taken. This may be valuable in some applications.

## 3 Design and execution environment

The design and execution environment is illustrated in Figure 4. The designer typically uses a graphical editor to produce a Ptides-based system model as shown in Figure 1.

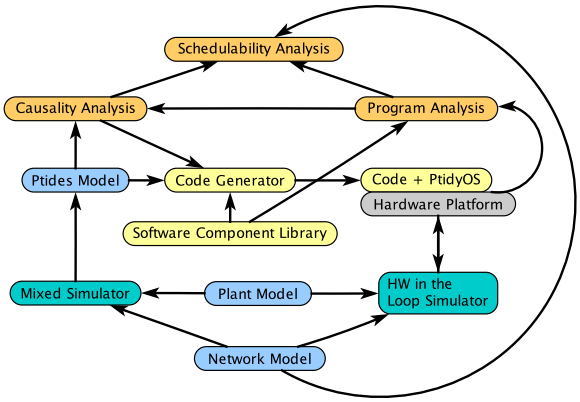


Figure 4: Ptides design environment

The model properties are observed using a mixed MOC simulator (one that supports multiple MOCs), in our case the Ptolemy simulator. Once the designer is satisfied, the target platform is specified which in turn specifies platform specific versions of Ptides actors, e.g. I/O and network

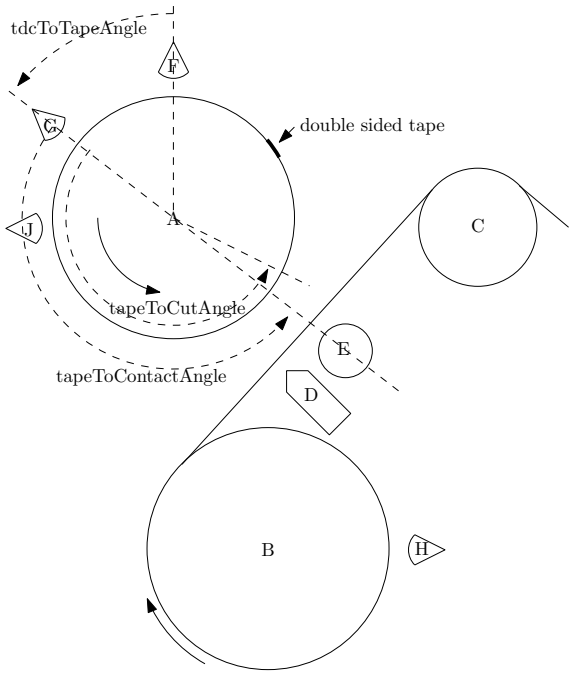


Figure 5: Flying paster

port actors. Causality analysis determines causal dependencies in the design to enable optimal DE execution [13]. Schedulability analysis determines whether the selected platform is able to meet the execution time constraints discussed in section 2.2. The code generator produces code that executes on a lightweight OS, PtidyOS [14], that supports a DE style of execution, the platform specific I/O and network ports and the synchronization of the platform real-time clock to its peers in other system platforms. The portions of PtidyOS code that implement access to the real-time clock, and I/O and network ports is platform specific. For other actors the PtidyOS code is the output of standard compilers for the microprocessor. This permits automatic generation of the final executable code from the design models.

# 4 Experimental results

To illustrate the temporal properties of a Ptides-based system, we implement a controller for a flying paster, which is the device in a printing press that allows on-the-fly changeover from an active to a spare roll of paper. The schematic of a flying paster is shown in Figure 5 and a video of a flying paster in operation can be found at [15].

In Figure 5, the current roll of paper $B$ feeds paper, i.e. the *web*, to the rest of the machine via an idler wheel $C$. A sensor $H$ detects the radius of the paper on $B$, generates a signal, *radius*, and at a designated value directs a controller to bring the reserve roll $A$ up to speed such that the velocity of the paper at the edge of $A$ matches that of the web. At a specified minimum radius, sensor $H$ generates an arming signal, *arm*. A sensor $J$ detects the radius of the paper on $A$. The outer layer of paper on $A$ has a strip of double sided tape that can be detected by sensor $G$, which then generates a signal, *tape*, the first time the tape is detected after the *arm* signal. This event

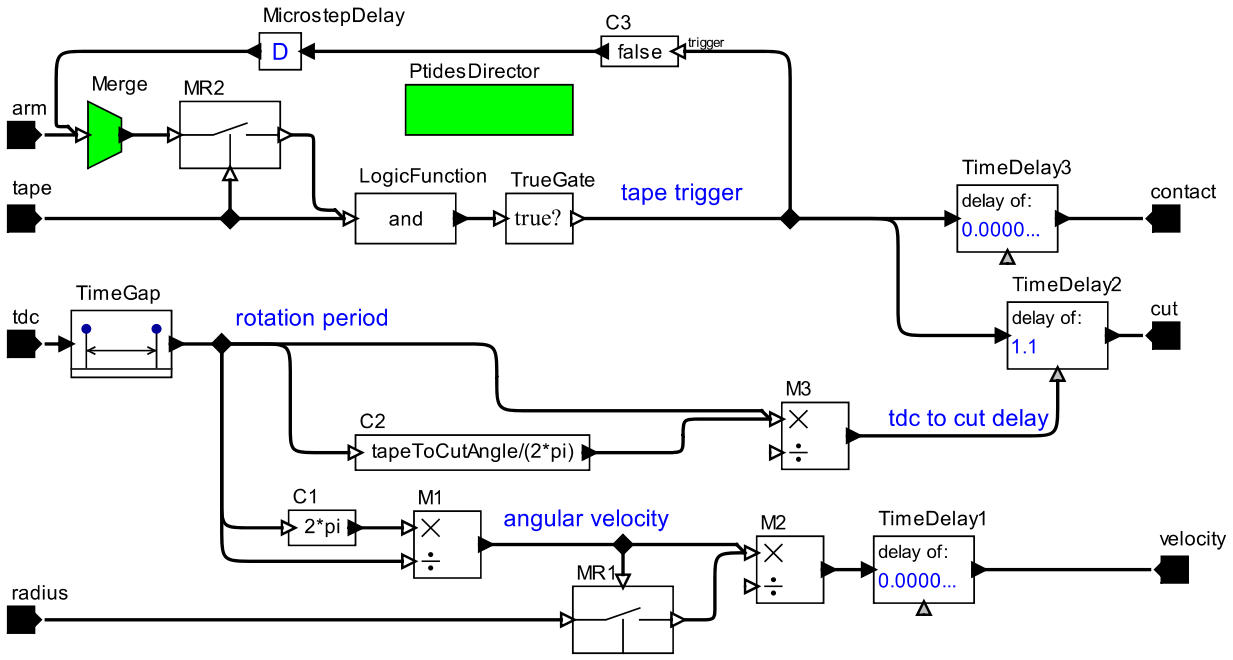


Figure 6: Flying paster controller

also causes the idler wheel $E$ to press the web against the reserve roll $A$, such that when the tape reaches a point relative to $G$ by an angle *tapeToContactAngle*, the two pieces of paper attach and the paper on $A$ follows the path of the web. At the angle *tapeToCutAngle* relative to $G$, the cutter $D$ cuts the paper feed from $B$ allowing $A$ to supply paper to the web. Clearly *tapeToCutAngle* must be greater than *tapeToContactAngle* to ensure that the reserve paper is attached to the web prior to cutting the feed from $B$. A sensor $F$ detects the top dead center point, *tdc*, on roll $A$.

To demonstrate the timing properties of Ptides generated code, we implement the controller shown in Figure 6, that accepts the sensor signals *arm, tdc, tape*, and *radius*, and generates the signals *contact* and *cut* to drive the actuators. The controller also outputs the linear velocity of the paper on roll $A$.

Recall from section 2.1 that events, e.g. *arm*, consist of a ($\text{value}_{arm}$, $\text{timestamp}_{arm}$, $\text{microstep}_{arm}$) triple. The *Merge* actor accepts the *arm* signal from the sensor and transmits it to the *MR2* actor. The *MR2* actor has the property that the value, e.g. $\text{value}_{arm}$, of the most recent token on the input is output as the ($\text{value}_{arm}$, $\text{timestamp}_i$, $\text{microstep}_i$) triple where $\text{timestamp}_i$ and $\text{microstep}_i$ are the timestamp and microstep of a token received on the trigger port. This type of actor has a default parameter specifying a value, in this case false, applied if the trigger occurs before a token is received on the input. Therefore, when a *tape* signal is received after the *arm* signal, the logic generates the *tape trigger* event with the timestamp of the *tape* event. The *TimeDelay3* actor increments this timestamp and sends the event to the *contact* actuator. A slight delay is required since these computations take a finite amount of time to execute and Ptides semantics requires that actuation occur when the timestamp and the platform clock values match. This is an example of the schedulability requirement discussed in section 3. The *tape trigger* event also causes a false event to be fed back to the *Merge* actor, which in turn sets the most recent value in *MR2* to false

thus suppressing the generation of additional *tape trigger* events. The *MicrostepDelay* actor increments the microstep value of the token and is necessary since inputs to actors, in this case *MR2*, by DE semantics must be processed in strict superdense time order. Since the timestamp of the token from the feedback loop is identical to that of the trigger input, the increment in the microstep provides the needed sequential ordering. The process of checking for these causality loops is part of the *Causality Analysis* block of Figure 4.

The tape trigger event is also sent to the *TimeDelay2* actor. This actor increments the timestamp by the last value provided by an event on the port on the bottom side of the actor. This value is produced by the actors on the path starting with the *tdc* sensor input. The *TimeGap* actor outputs the time interval between successive *tdc* signals, which from Figure 5 represents the rotation period of roll $A$. This value is multiplied by the fraction of a period represented by the *tapeToCut* angle thus yielding the correct delay for the *tape* sensor to the *cut* actuator path. Similarly, the bottom path computes the paper velocity output based on the *radius* input from sensor $J$ and the *rotation period*.

### 4.1 Experiment design

To test the proposed design environment, code was generated from the Ptides model of Figure 6 for two very different execution platforms. Simulation and execution results are shown in Figure 7. The microprocessor on the Renesas platform [16] is a typical embedded controller found in industrial applications. The clock and timing functions discussed in section 2.2 were implemented using timing hardware in the DP83640 Ethernet PHY of Texas Instruments [17] used as the Ethernet interface on the Renesas platform. The second test platform used an XMOS multicore microcontroller [18] that provides up to 8 hardware supported threads of execution. XMOS devices execute with single cycle execution of instructions and therefore provide very deterministic execution timing for the code generated in this experiment.

In this experiment, the external input signals *tdc, arm* and *tape* were generated using National Instruments PXI cards in lieu of using an actual flying paster. These inputs and the resulting outputs *contact* and *cut* were captured using an oscilloscope. In Figure 7 the first set of inputs show the signals *tdc* followed by *arm* and *tape* each separated by 50$\mu$s. The resulting outputs *contact* and *cut* follow the *tape* signal by 200$\mu$s and 500$\mu$s respectively. The second set of inputs consist of the *tdc* and *tape* signals but since there is no *arm* signal there are no outputs. In Figure 7 the timescales of the top, middle and lower panels are in ms, $\mu$s, and ms respectively resulting from the different measurement instrumentation used for each. Note that the plots for the simulation and for execution on the two platforms are identical in spite of the very different architectures and capabilities of the two execution platforms. Although not visible at the scale of these plots, the results were precise to 8ns on the Renesas platform using the DP83640 and to a similar precision on the XMOS platform.

In XMOS, if more than four hardware execution threads are active, the throughput of each thread is reduced. The XMOS plot shown is for the case where four threads are used for the execution (but not always active). Although not illustrated, we also compiled the code for two other cases: using all eight hardware threads and utilizing a second core. The resulting IO timing from each case is identical, even though the execution timing varied.

The invariance of the external timing with respect to changes in platform capability results from the properties of the Ptides actors discussed in section 2. Consider the token $(true, 100\mu s, 1)$ placed on the event queue at the *tape* sensor. The token $(true, 100\mu s + tdctocutdelay, 1) = (true, 600\mu s, 1)$

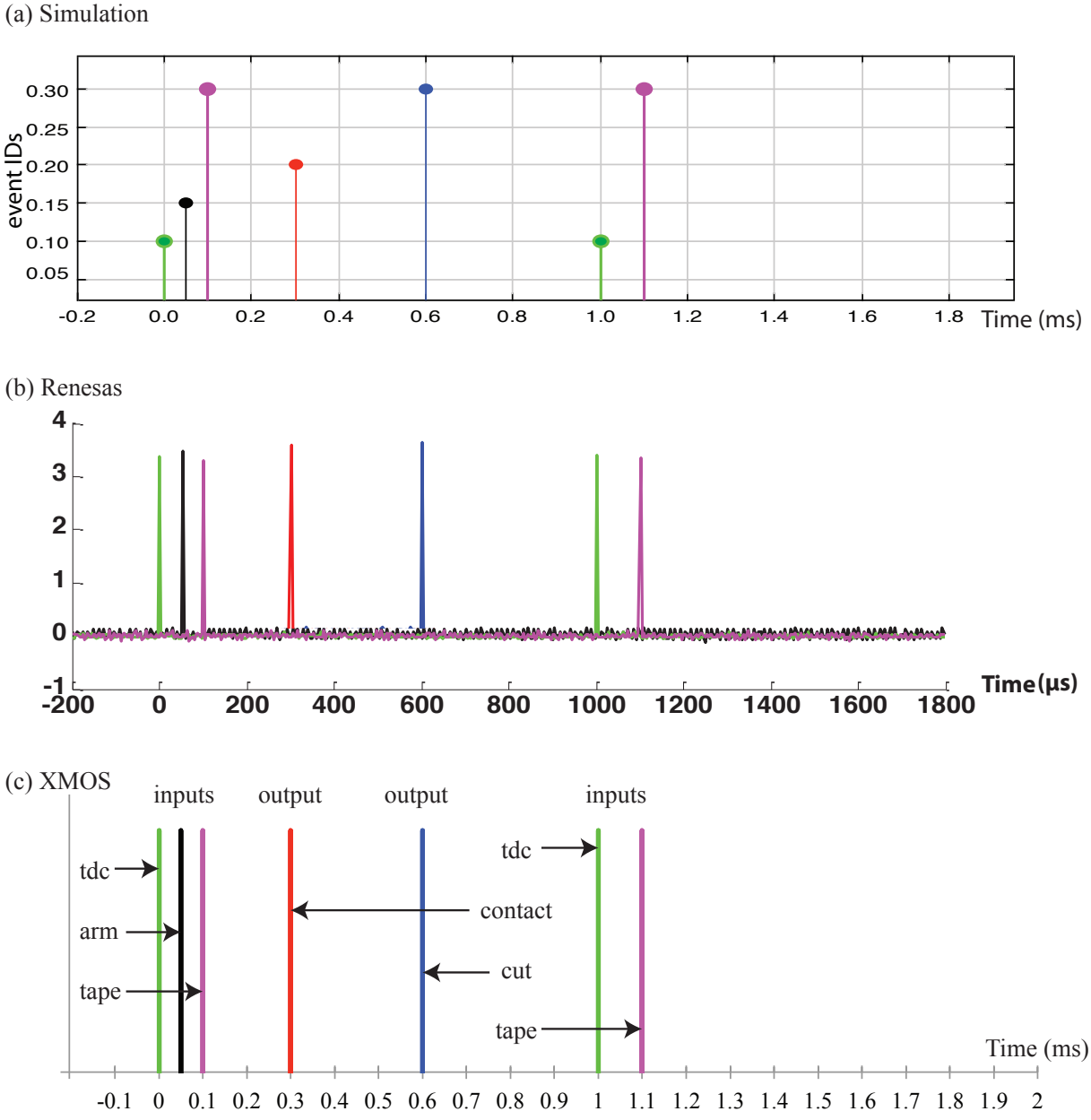


Figure 7: Flying paster results

produced at the output of actor *TimeDelay2* is applied to the *cut* actuator at time $600\mu s$ on the clock of the controller platform. Note that the time $\tau$ that this token was produced on the output of actor *TimeDelay2* is $100\mu s < \tau \leq 600\mu s$. For a slow processor $\tau$ will be closer to $600\mu$s and for a fast processor closer to $100\mu$s but as long as $\tau \leq 600\mu s$ the external timing specifications are met.

### 4.2 Insights gained

External timing accuracy depends on the implementation of the architecture features discussed in connection with Figure 2. These features can be implemented in software but the latency and jitter will reduce the accuracy compared to the hardware assisted approaches demonstrated in these experiments. Less obvious are the other time constraints imposed by the implementation architecture. For example, if the interfaces used by software to acquire sensor data or cause actuation have excessive latency this reduces the available software execution time and therefore may preclude a feasible schedule during schedule analysis. Likewise, the safe-to-process analysis at a multiport actor requires that the execution be postponed for the computed real-time delay. The value of this delay is relative to the time of the platform clock and for efficient implementation requires a low latency read path to this clock. This suggests that external hardware should have a memory mapped interface to the platform clock and sensor and actuator registers rather than the serial or network path interfaces often found.

## 5 Conclusions and future work

We outlined the principal ideas of the design, simulation, and code generation environment for Ptides and illustrated how it is used on a simple example. This example describes functional and timing behavior in a platform-independent way. A comparison of simulation results with results from executions from two very different implementations of this example shows that correct functional and timing behavior can be achieved and the generated code exhibits identical timing at the cyber-physical boundary on the different implementation platforms. Specifically, network latency and clock synchronization error must be bounded and the platforms must be capable of executing the code within the design times. It is also possible to detect timing errors at run time, e.g. errors introduced by transient out of specification network delay. These characteristics are valuable in a variety of application domains such as test or control systems where deterministic timing is critical and must be invariant over changes in internal component design, upgrades in microprocessor speed and the like.

The Ptides principles should also allow machine readable timing interface specifications to be created for devices. This raises the possibility of designing components and systems that can do run time checks for timing feasibility. This possibility and the investigation of optimum designs for the hardware support of Ptides semantics provide interesting opportunities for future research.

## References


[1] Y. Zhao, E. A. Lee, and J. Liu, "A programming model for time-synchronized distributed real-time systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. Bellevue, WA, USA: IEEE, 2007, pp. 259 – 268. [Online]. Available: http://ptolemy.eecs.berkeley.edu/publications/papers/07/RTAS/

[2] G. Papadopoulos and F. Arbab, "Coordination models and languages," in *Advances in Computers - The Engineering of Large Systems*, M. Zelkowitz, Ed. Academic Press, 1998, vol. 46, pp. 329–400.

[3] J. Eidson, E. Lee, S. Matic, S. Seshia, and J. Zou, "Distributed real-time software for cyber-physical systems," *Proceedings of the IEEE (special issue on CPS)*, vol. 100, no. 1, pp. 45–59, 2012.

[4] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity—the Ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 2, pp. 127–144, 2003. [Online]. Available: http://www.ptolemy.eecs.berkeley.edu/publications/papers/03/TamingHeterogeneity/

[5] C. G. Cassandras, *Discrete Event Systems, Modeling and Performance Analysis.* Irwin, 1993.

[6] E. A. Lee, "Modeling concurrent real-time processes using discrete events," *Annals of Software Engineering*, vol. 7, pp. 25–45, 1999. [Online]. Available: http://dx.doi.org/10.1023/A:1018998524196

[7] E. A. Lee, S. Neuendorffer, and M. J. Wirthlin, "Actor-oriented design of embedded hardware and software systems," *Journal of Circuits, Systems, and Computers*, vol. 12, no. 3, pp. 231–260, 2003.

[8] Z. Manna and A. Pnueli, "Verifying hybrid systems," in *Hybrid Systems*, vol. LNCS 736, 1993, pp. 4–35.

[9] J. C. Eidson, E. A. Lee, S. Matic, S. A. Seshia, and J. Zou, "Time-centric models for designing embedded cyber-physical systems," EECS Department, University of California, Berkeley, Technical Report UCB/EECS-2009-135, October 9 2009. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-135.html

[10] I. Instrumentation and M. Society, "1588: IEEE standard for a precision clock synchronization protocol for networked measurement and control systems," IEEE, Standard Specification, July 24 2008.

[11] D. May, "The XMOS XS1 architecture," XMOS Limited, White Paper, 2009. [Online]. Available: https://www.xmos.com/zh/download/public/The-XMOS-XS1-Architecture(1.0).pdf

[12] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee, "Predictable programming on a precision timed architecture," in *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, Atlanta, 2008.

[13] Y. Zhou, "Interface theories for causality analysis in actor networks," PhD, University of California, Berkeley, 2007.

[14] J. Zou, S. Matic, and E. Lee, "PtidyOS: A lightweight microkernel for Ptides real-time systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS).* IEEE, April 2012.

[15] Drupaloge. (2010, December) Printip - lithoman iv flying splice. [Online]. Available: http://www.youtube.com/watch?NR=1\&v=wYRGiXMUzA4

[16] Renesas, "Renesas demonstration kit for SH7216," Retrieved February 1 2013. [Online]. Available: http://am.renesas.com/products/tools/introductory_evaluation_tools/renesas_demo_kits/yrdksh7216/index.jsp

[17] T. Instruments, "Precision PHYTER - IEEE 1588 precision time protocol transceiver," Retrieved February 1 2013. [Online]. Available: http://www.ti.com/product/dp83640

[18] XMOS Limited, "XK-1A development kit," Retrieved February 1 2013. [Online]. Available: http://www.xmos.com/discover/products/xkits/starter#tabs