20th IMEKO TC4 International Symposium and
18th International Workshop on ADC Modelling and Testing
Research on Electric and Electronic Measurement for the Economic Upturn
Benevento, Italy, September 15-17, 2014

Logically Synchronous Models of Distributed Systems with Explicit Timing Specifications

Patricia Derler[1], Edward A. Lee[1], Michael Zimmer[1]

[1]*University of California, Berkeley,*
{pd, eal, mzimmer}@eecs.berkeley.edu

***Abstract* – Globally asynchronous, locally synchronous (GALS) has become the standard model of computation (MoC) for designing distributed software systems. Essentially, distributed components are each internally synchronous, but operate in different clock domains and communicate with other components asynchronously. Advances in clock synchronization mechanisms, supported by the increasing availability of clock synchronization implementations, allow for establishing a common notion of time across distributed platforms. We therefore advocate for a synchronous model of computation for the distributed system as a whole, assuming synchronized clocks. The paper discusses such a MoC called Ptides (Programming Temporally Integrated Distributed Embedded Systems). A Ptides model explicitly describes platform independent time delays within and across distributed components.** [1]

I. INTRODUCTION

The importance of correct timing in embedded and cyber-physical systems has been recognized and various efforts are dedicated to (re)introducing time into programming abstractions. When it comes to distributed systems, the model of computation (MoC) that has been successfully applied is globally asynchronous, locally synchronous (GALS). GALS [1] has been introduced in the 1980s to relax the synchrony assumption between different clock domains on distributed components. A system is split into synchronous islands that communicate asynchronously, for instance, via FIFO queues. GALS is used both for software and hardware systems. Here, we focus on modeling distributed software systems.

A synchronous system is one where at every time step, every variable has precisely one value, or the value is absent. The system is represented as a collection of communicating components whose state is updated in one zero-time step. Synchronous systems are typically deterministic, i.e. given the same input, the system always produces the same outputs. This property is important for testing and verification. Examples of synchronous programming languages are Lustre [2] and Esterel [3]. The discrete event (DE) MoC is also synchronous because at every time step, the exact value of each variable can be determined. Absence of a value at a certain time means that there is no event with that time stamp. DE is commonly used for simulation [4], where time is logical and merely used to order actions. In this article, we use DE as a programing model.

Synchronous subsystems in a GALS architecture communicate asynchronously with one another, and in many implementations, inputs for such subsystems are sampled. Nondeterminism can be introduced at this point. While nondeterminism is not erroneous behavior, it does impede the analysis of a system and makes it harder to give guarantees about the behavior, i.e. the timing and functionality.

A key assumption in GALS systems is that clocks are not synchronized. In recent years, clock synchronization technologies have been introduced and various implementations are available, ranging from synchronization to time servers or satellites all the way to protocols that take care of the time synchronization (see IEEE1588 [5] or NTP [6], for example). Clock synchronization technology provides a distributed system with a common notion of time.

In this article, we advocate for leveraging clock synchronization in order to develop an entire distributed system as a synchronous system with one, logical clock domain. We refer to this system as logically synchronous. Clock synchronization guarantees that the difference between clock values is bounded with a known clock error.

A synchronous MoC for developing distributed systems that relies on clock synchronization is Ptides, which stands for Programming Temporally Integrated Distributed Embedded Systems. Ptides extends the discrete-event MoC and links logical time to real time, also referred to as physical time, at I/O to achieve deterministic behavior. A Ptides model contains explicit time delays between input and output actions. In a Ptides system, all communication is time stamped, which allows for reasoning about the causality of events, the detection of timing errors and even the identification of error conditions such as missing inputs. We discuss the programming model on an example and discuss extension and error handling possibilities.

[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, which receives support from the National Science Foundation (NSF awards #0720882 (CSR-EHS: PRET), #1035672 (CPS: Medium: Timing Centric Software), and #0931843 (CPS: Large: ActionWebs)), the Naval Research Laboratory (#NOOI73-12-1-G015), and the following companies: Denso, National Instruments, and Toyota.

ISBN-14: 978-92-990073-2-7

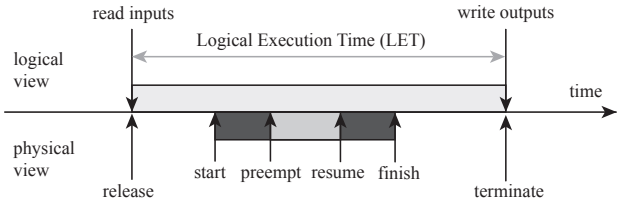

Fig. 1. The logical execution time abstraction.

II. BACKGROUND AND RELATED WORK

The design of a distributed system as if all distributed components were driven by a perfect global clock is also addressed in the work on physically asynchronous, logically synchronous (PALS) [7] architectures. While the PALS work deals with synchronization protocols and proposes special components to handle the interaction with the environment, our approach differs in that we assume synchronized clocks but do not require special synchronizers. Instead, all components communicate via time-stamped events according to the DE model of computation and components in the system consume events in time-stamped order.

Most synchronous languages have rigorous mathematical semantics and are used for programming real-time systems. The assumption is that processors are infinitely fast. Latencies arise from the implementation instead of being part of the programming abstraction. Here, we explicitly specify latencies as part of the programming model.

The logical synchrony idea is closely related to the logical execution time abstraction introduced in Giotto [8], a time-triggered programming model. The idea of the LET abstraction is illustrated in Figure 1, which represents one execution of a task, a part of a software system. All inputs are read at the release time of the task and all outputs are written at the termination time. A fixed logical execution time defines the distance between release and termination time. The actual execution of this task happens any time in between release and termination time. The only requirement for the task execution is that the time of I/O operations is fixed.

III. PTIDES

Ptides (Programming Temporally Integrated Distributed Embedded Systems) [9] is a programming model for the design of distributed, cyber-physical systems. Ptides models are deterministic with respect to value and timing. One of the key ideas is, just like in the LET abstraction, to fix the time between I/O operations, but only constrain execution and communication where necessary. Ptides abstracts away from execution and communication delays of actual implementations by using logical time delays, which are platform-independent.

Ptides is based on the discrete event MoC where components communicate via time-stamped events. The time stamps refer to logical time and are used to order events. Distributed Ptides components, also referred to as *platforms*, contain *sensors*, *actuators*, *network input ports*, *network output ports*, *computations* and *delay blocks*. Network ports send and receive events to and from other Ptides platforms, whereas sensors and actuators communicate with other non-Ptides components or the physical environment. We assume all clocks in the Ptides platforms to be synchronized with a known clock synchronization error of ϵ.

Within a Ptides platform, all communication is done via time-stamped events (t, v) with t as the time stamp and v the value. Sensors receive inputs from the environment either by detecting a change in the environment or by sampling. Sensors then create events with time stamps equal to the current physical time, which is obtained from the platform clock. Computations consume events and produce new output events. All inputs consumed by a computation are time-stamped events and for one execution, only events with the same time stamp are consumed. Multiple events with different time stamps result in multiple executions of the computation, one for every time stamp. Computations can have state, in which case they have to consume input events in time stamp order. If inputs are consumed in arbitrary order, the system might become nondeterministic. Computations do not match the time stamp of the event to real time.

Delay blocks increase the time stamp of input events by a specified amount. Delays are application specific and platform independent; they are derived from the application requirements. For instance, in the case of a control loop, the control engineer determines the delay between input and output that leads to a stable controller. Delay values between sensors and actuators act as the deadline for the actuator. Actuators receive events (t, v) and perform the communication with the environment at time $t_p \leq t$, where t_p is the platform time; i.e. the time of the platform clock. If the time stamp of the event received by an actuator is smaller than the current platform time, then the deadline was missed.

Network ports are used for communication between Ptides platforms via networks. Network output ports treat event time stamps similar to actuators. To simplify presentation, we are ignoring the clock synchronization bound ϵ. A network output port has to send an event (t, v) at a platform time $t_p \leq t$. A network input port has to receive an event (t, v) at a time $t_p \leq t+n$, where n is the upper bound on the network communication time. In order to give timing guarantees, such an upper bound must be specified. If a network output port receives an event with time stamp $t < t_p$, or if a network input port receives an event with time stamp $t < t_p - n$, then the timing specifications of the Ptides program were violated.

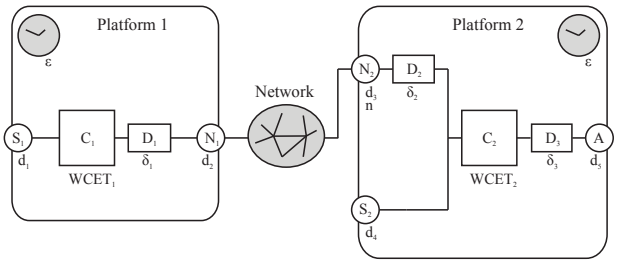

Fig. 2. Example of a distributed system model using Ptides.

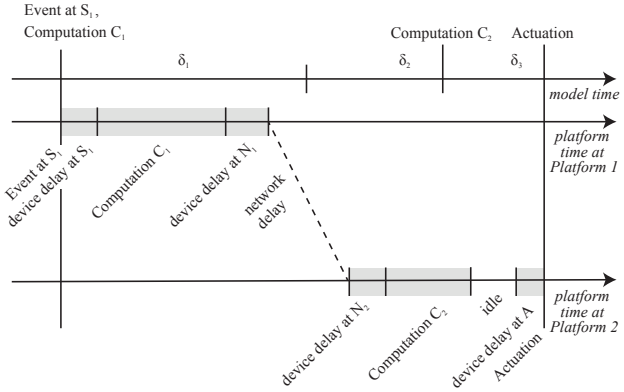

Fig. 3. Transmission of one event from sensor S_1 to actuator A in model and real time.

For every I/O device, a *device delay bound* parameter specifies a bound on the physical time it takes for the device to perform the I/O operation. For instance, it takes some time between measuring a change in the environment and creating a time-stamped event that can be posted on the event queue.

A. A Ptides Example

Figure 2 shows an example of a Ptides system with two platforms. The clock symbol in the platforms symbolizes the platform clock. ϵ is the clock synchronization error bound which is the same for all platforms in the system. The computation blocks C_1 and C_2 are associated with a worst case execution time WCET_1 and WCET_2, the delay blocks D_1, D_2 and D_3 are associated with logical time delays δ_1, δ_2 and δ_3 and all I/O devices have device delay bounds d_1 through d_5. Only δ_1, δ_2 and δ_3 are needed for modeling and analysis of the platform-independent behavior, so all executions and communications are assumed to have zero execution time. All other parameters are needed for the platform dependent schedulability analysis which determines whether this system can be executed correctly on a given hardware.

Figure 3 depicts the processing of one event from sensor to actuator and compares the logical view with the physical view. The process is described from the model perspective, using logical time, and from the physical perspective, by showing the platform time lines. The event is created by sensor S_1 on platform 1 and it is transmitted to actuator A on platform 2. The top time line shows the logical time behavior of the system. An event from Sensor S_1 is received by A after $\delta_1 + \delta_2 + \delta_3$ time units—a straightforward, platform-independent specification of functionality and timing behavior. Computations are assumed to have zero execution time in the modeling phase.

The bottom time lines illustrates the transmission of one event in a possible implementation of this system; this implementation must decide when to execute computations to achieve the specified behavior. Platform 1 and platform 2 have different time lines representing the different clocks. In this example, we require the actuator to send the output at the time of the event time stamp, not before. Thus, there is some idle time before the actuation. Note also that the actuator has to take into account the device delay meaning that the actuator has to start actuation before the deadline. Note that, in this example, we assume that sensor S_2 does not produce any events.

In order to ensure that an event created by S_1 can be received by the actuator A in time, all delays along the path have to be added up and compared to the logical time delays. In this example, an event can be processed in time, if both $d_1 + \text{WCET}_1 + d_2 \leq \delta_1$ and $n + d_3 + \text{WCET}_2 + d_5 + \epsilon \leq \delta_2 + \delta_3$. Now this case only deals with one event with no other demands on the processor. In a real system, there might be several events with short inter-arrival times. Also, with more elaborate program structures such as loops, merges and splits, the analysis becomes more difficult. The schedulability problem for Ptides becomes more complex, however, it is decidable as shown by Matsikoudis et al. [10].

On platform 2, events are merged between sensor S_2 and the network inputs. Here, a so called "safe-to-process" analysis is needed to determine when it is safe to process an event. Assume network input N_2 receives an event, which is then processed by the delay component D_2. D_2 increases the time stamp of the event by δ_2. Computation C_2 has to consume events in time stamp order, thus we need to make sure that there is no event on any other input, i.e. there is no event from sensor S_2, arriving with a smaller time stamp. Execution strategies that involve different ways of performing this safe-to-process analysis are discussed by Zou et al. [11]. Also, a more complex model can have multiple computations that are safe-to-process, requiring a scheduling decision [12].

The application behavior is value and timing deterministic. The implementation of the above described system can be distributed differently. For instance, computation C1 can be split into two computations that are performed on different platforms. Or, the system can be implemented

on a single platform. Because of the behavior implemented by the sensors, actuators and network ports, we can guarantee that, given that the system is schedulable, the I/O behavior will remain unchanged.

B. Interface of a Ptides platform

A Ptides platform can be used and connected to other Ptides platforms as a black box, if the Ptides platform is schedulable, and the network delay bounds n on all network delays are specified. The implementation must then guarantee that the network delays are not bigger than the given bounds.

An extension to the Ptides interface can loosen the timing requirements on platforms and expose additional information at the interface. In the example in Figure 2, if the computation C_1 takes longer than the delay δ_1, then a problem will be detected at the network port N_1. However, if computation C_2 only takes a very short time, the event from sensor S_1 could still be at the actuator in $\delta_1 + \delta_2 + \delta_3$ time units. An extension to the interface can be implemented by adding an additional parameter to both the network output and the network input port. The network output port additionally gets a parameter *platformDelayBound* p which is to be added to the explicit delays in the model. The network port then checks the time stamp of this event t and compares it to the physical time. In the event that physical time $t_p > t + p$, an error occurred. This error can stem from an erroneous design, wrong assumptions about the platform, or malfunction of the hardware.

On the receiving side of the event, the network input port also gets an additional parameter called *sourcePlatformDelayBound*. This parameter must have the same or larger value as the connected network output port. The network receiver can then check the time stamp of the incoming event with the same condition as the sending network port. If physical time $t_p > t + p + n$, then the event was received too late.

C. Error handling

A Ptides system can detect violations of timing specifications. Network ports and actuators know whether events have been received in time or too late. The latter is also referred to as a deadline miss. With Ptides, it is also possible to observe that an event is missing completely, for instance, due to hardware failure. Missing events can be detected with heart beat monitors. A model of such a detection and the monitor design in Ptides is illustrated by Eidson et al. [13].

Handling of such errors is application specific. Many hard real time applications are fault tolerant and can accept one or a few missed deadlines. In such cases, an event that was received late can still be transmitted. In applications where the late transmission of an event causes problems, the event can be discarded. Deadline misses should be recorded such that, in case of repeated deadline misses, users can be notified and appropriate countermeasures can be taken. The causes range from hardware problems to faulty system designs. In order to prevent further deadline misses, actions such as switching to operation modes with lower resource consumption can be taken. Another possibility to remedy the problem is to increase the logical delays in order to adapt to the increased load.

IV. CONCLUSIONS

This article advocates for modeling distributed systems logically synchronous, by assuming one common notion of time. With technology available today, this common notion of time is achievable: various clock synchronization mechanisms have been developed that keep clocks in distributed systems synchronized with a known upper bound on the error. We discuss Ptides as a synchronous programming model for distributed systems. Ptides extends the discrete event model of computation to include a careful mapping between logical time and real time. This mapping allows for the development of deterministic systems with the same I/O behavior in simulation and execution, even on different platforms. This work is demonstrated in a prototypical implementation in the Ptolemy II [14] framework, which includes the modeling capabilities that allow for designing and simulating Ptides platforms with all the parameters. Error handling mechanisms can be modeled as part of this framework. The workflow around the Ptides project also includes a code generator and schedulability analysis tools.

While the Ptides programming model is based on the discrete-event MoC, the application is not restricted to DE. We envision the use of Ptides as a coordination language for software tasks that can be implemented in other MoCs, as long as they can be encapsulated inside a DE block. This merely means that tasks are triggered by inputs. The time stamps of the inputs determine the order in which tasks are triggered. Tasks have to produce time-stamped outputs. If the MoC of the task is untimed, the output time stamps are the same as the input time stamps. For timed MoCs (where time is a logical concept used to order executions), appropriate delay blocks have to be inserted. Even legacy code that is non-deterministic can be made deterministic by wrapping legacy software tasks inside Ptides components that enforce deterministic timing of input and output operations.

REFERENCES

[1] D. M. Chapiro, *Globally-Asynchronous Locally-Synchronous Systems*. PhD thesis, Stanford University, Oct. 1984.

[2] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE," *Proceedings of the IEEE*, vol. 79, no. 9,

pp. 1305–1319, 1991.

[3] G. Berry and G. Gonthier, "The Esterel synchronous programming language: Design, semantics, implementation," *Science of Computer Programming*, vol. 19, no. 2, pp. 87–152, 1992.

[4] J. Misra, "Distributed discrete event simulation," *ACM Computing Surveys*, vol. 18, no. 1, pp. 39–65, 1986.

[5] "IEEE standard for a precision clock synchronization protocol for networked measurement and control systems," *IEEE Std. 1588-2008*, 2008.

[6] D. L. Mills, "A brief history of NTP time: confessions of an internet timekeeper," *ACM Computer Communications Review*, vol. 33, 2003.

[7] L. Sha, A. Al-Nayeem, M. Sun, J. Meseguer, and P. Ölveczky, "PALS: Physically asynchronous logically synchronous systems," tech. rep., Tech. rep., Department of Computer Science, University of Illinois at Urbana-Champaign (2009), 2009.

[8] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Embedded control systems development with giotto.," in *LCTES/OM* (S. Hong and S. Pande, eds.), pp. 64–72, ACM, 2001.

[9] 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), pp. 259 – 268, IEEE, 2007.

[10] E. Matsikoudis, C. Stergiou, and E. A. Lee, "On the schedulability of real-time discrete-event systems," in *13th International Conference on Embedded Software (EMSOFT)*, September 2013. Montreal, Canada.

[11] 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 *15th IEEE Real-Time and Embedded Technology and Applications Symposium, 2009*, pp. 77–86, IEEE Computer Society, April 2009.

[12] 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.

[13] J. Eidson, E. A. Lee, S. Matic, S. A. Seshia, and J. Zou, "A time-centric model for cyber-physical applications," in *Workshop on Model Based Architecting and Construction of Embedded Systems (ACES-MB)*, 2010.

[14] J. Cardoso, P. Derler, J. C. Eidson, E. A. Lee, S. Matic, Y. Zhao, and J. Zou, "Modeling timed systems," in *System Design, Modeling, and Simulation using Ptolemy II* (C. Ptolemaeus, ed.), Ptolemy.org, 2014.