Proper Handling of Interrupts in Cyber-Physical Systems

Mateus Krepsky Ludwich and Antônio Augusto Fröhlich
Software/Hardware Integration Lab (LISHA)
Federal University of Santa Catarina (UFSC)
P.O.Box 476, 880400900 - Florianópolis - SC - Brasil
Email: {mateus,guto}@lisha.ufsc.br

***Abstract*—Interrupt handling plays a fundamental role in Cyber-Physical Systems (CPS), particularly for those whose complexity cannot go with simplistic sensing-control-actuation loops designed around ordinary polling operations. Such systems usually rely on some sort of Real-Time Operating System (RTOS) to support the concurrent execution (or parallel execution, for multicore platforms) of periodic threads that interact with hardware devices mainly through interrupts. Since hardware interrupts are asynchronous with respect to the execution flow of threads, inappropriately handling them has the potential to disrupt the system's real-time specification, causing undesirable jitter and potential deadline losses.**

In this paper, we investigate interrupt handling strategies for RTOS considering CPSs designed around specific Models of Computation (MoC). We compare traditional Interrupt Service Routines (ISR), which perform both interrupt reception and the servicing all together, with a time-predictable mechanism that decouples interrupt reception from servicing, making the first deterministic in terms of time and scheduling the second along with other real-time threads. We compare them in terms of latency and jitter considering two different scenarios: that of systems designed around the Discrete Events (DE) MoC and that of those designed around the Time-Triggered (TT) MoC. Our results show that the time-predictable mechanism is essential for TT, while the traditional interrupt handling mechanism is more suitable for DE, thus demonstrating that RTOS need to be configurable in this respect in order to support forthcoming Cyber-Physical Systems.

I. Introduction

A modern Cyber-Physical System (CPS) is usually designed and implemented following some sort of model-based methodology [7], [10]. Actors, ports, interfaces, and messages are concepts typical of this domain, in which components concurrently interact with each other following the semantics given by a set of rules commonly designated as a Model of Computation (MoC) [14]. Each MoC defines entities that enable such components to interact with the real world either by sensing it through transducers or by interfering with it through actuators. Though this interaction can be done by directly polling devices whenever the related information is needed, models such as Time-Triggered (TT) and Discrete Events (DE) visible fit better in a platform that uses interrupts to notify meaningful state changes. Interrupts are available in virtually any control platform and besides nicely fitting some MoCs, they can also save considerable amounts of energy through the elimination of busy-waiting cycles. Nevertheless, since hardware interrupts are asynchronous in respect to the execution flow of processors, inappropriately handling them can disrupt the system's timing specification, causing undesirable jitter and potentially causing deadline losses that can lead to general system failures.

In a recent work, we investigated the use of components commonly found in modern Real-Time Operating Systems (RTOS) to support CPSs designed around a specific MoC [9]. In this paper, we further extend that investigation to encompass the relationship between the interrupt handling mechanisms commonly available in modern RTOS and the different MoCs around which CPSs are usually designed. We considered the two most common interrupt handling strategies: Interrupt Service Routines (ISR), which perform both interrupt reception and the servicing at once, and Interrupt Service Threads (IST), which decouple interrupt reception from servicing, scheduling the latter along with other real-time threads. We analyze both strategies in terms of latency and jitter while considering two meaningful scenarios: that of systems designed around the Discrete Events (DE) MoC and that of those designed around the Time-Triggered (TT) MoC. Our results show that a time-predictable mechanism is essential for TT, while traditional ISRs are more suitable for DE.

We also introduce the *Concurrent Observer* design pattern, which enables an efficient, and time-predictable decoupling of interrupt reception and servicing. Inspired by the original Observer design pattern [2], the Concurrent Observer achieves predictability by relying on two common features of RTOSs: the real-time scheduler and real-time semaphores. Interrupts risen by the hardware are converted into signals that release threads to be scheduled following the current system policy. We demonstrate that the implementation of the Concurrent Observer for the Embedded Parallel Operating System (EPOS) [1] RTOS yields a construct whose methods have constant execution times.

The remaining of this paper is organized as follows: Section II presents related work that focus on supporting MoC for CPS development and how they interact with the environment. Section III presents how our MoC-Oriented RTOS handles interrupts for DE and TT MoCs. Section IV presents the *Concurrent Observer* design pattern and how interrupts in our MoC-Oriented RTOS can be modeled using it. Section V presents results while evaluating ISR and IST-based interrupt

978-1-4673-8276-2/15/$31.00

handling encompassing average ISR time and interrupt latency. The final discussion and conclusions are presented in Section VI.

II. Related Work

This section presents related work regarding interrupt and design patterns applied to MoCs.

Traditionally, interrupt handling is performed by having an Interrupt Service Routine (ISR) that is invoked every time an interrupt occurs to service. Servicing an interrupt usually combines a data transfer phase, followed by some processing and either actuation or event signaling phases. As Figure 1 shows, for every interrupt occurrence, the interrupt reception, handling, and acknowledgment are executed in the context of the running thread. In this way, ordinary ISRs "invade" the running thread, causing jitter and consuming time that, in some cases, even compromise the job's deadline.

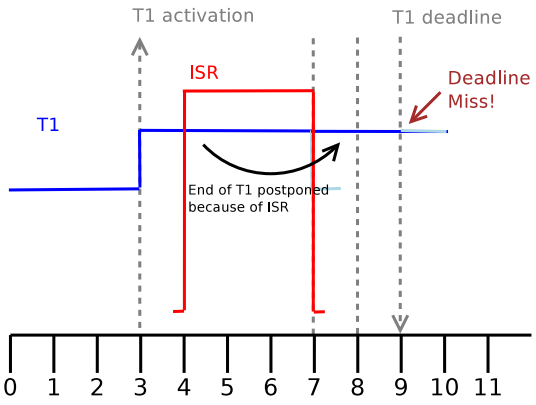

Fig. 1. Interrupt handling by an ISR.

Nevertheless, ordinary ISRs have the advantage of presenting very low latency and a virtually constant servicing activation time. This characteristic is exploited by some RTOS, including Sloth [3], to ensure a predictable execution environment for embedded software. In Sloth, threads and interrupts are handled by an interrupt request arbitration unit. A thread can be activated by a hardware interrupt or by another thread that sets its IRQ bit. The hardware interrupt dispatching mechanism takes over the role of the scheduler. If needed, a task prologue can be executed during thread dispatching to handle context saving and restoring. Schemes like this fit well in some classes of embedded systems for which events can be fully modeled at design-time (such as OSEK [13]). Each interrupt must be assigned a unique priority in the same scheme used by tasks and the latency of higher level interrupts or tasks must be accounted for the execution time of lower priority ones. If two interrupts happen at the same time, then the lowest priority one will experience a latency that is much higher than that of the hardware platform's. The possible accumulated delays quickly build up to unpredictable execution time for some tasks that are forced from a real-time category into a best-effort one [8].

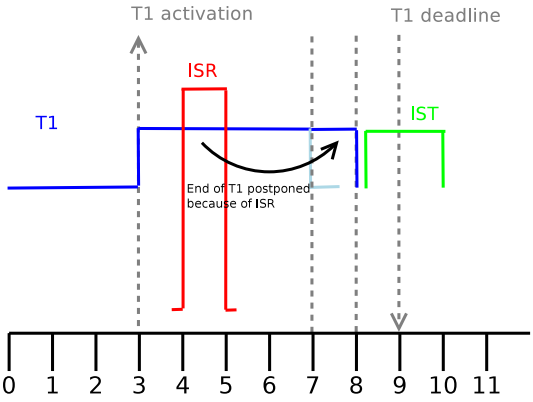

Fig. 2. Interrupt handling by an IST.

A long-standing alternative to deal with the interference caused by ISRs on more complex systems, for which it is usually impossible to specify a non-interfering model of interrupt triggering and task execution, are *Interrupt Service Threads* (IST), shown in Figure 2. RTOS supporting this mechanism, decouple interrupt reception from handling. Interrupt handling is performed by ordinary threads, scheduled following the same global policy applied to other threads [4], [5].

According to Pont and Banner, the use of design patterns can simplify the development of embedded systems, specially time-triggered, cooperatively scheduled (TTCS) [12]. They have assembled a collection of embedded systems design patterns named PTTES, including operating system's scheduling policies. The collection is intended to support the development of applications with a TTCS architecture. A washing machine control system was designed based on PTTES to illustrate their design. The Run-time Support System (RTSS) employed to support their application example is based on cooperative scheduling and uses polling to sample data from sensors.

Tripakis et al. proposed an approach for implementing synchronous model based systems on top of an abstract loosely time-triggered architecture (LTTA) [15]. The communication mechanism defined for the proposed architecture, called communication by sampling, is implemented using ordinary finite FIFOs. In this way, one could say that the proposed approach conveys means to map the synchronous reactive and the time-triggered MoC to relatively straightforward OS services.

As Pont and Banner [12], Tripakis et al. focus mostly on TT-based implementations of systems and use polling of sensors to interact with the environment. While polling is a feasible alternative for small systems it does not scale well for larger systems because it usually prevents modularity (the time and requirements need to be analyzed taking into account the whole system) and can consume more energy (in the case it is implemented using busy-waiting strategies).

III. Interrupt Handling in a MoC-Oriented RTOS

In a previous work, we have investigated the use of components commonly found in modern RTOS to support the

execution of a CPS designed around specific MoCs. The result could be called a MoC-Oriented OS [9]. In this section, we extend that design to address issues pertaining interrupt handling. While for the DE MoC interrupts are modeled as events that drive the application data flow, for the TT MoC interrupts represent a state variation that must not be propagated before the correct time dictated by the application requirements. This section explains how the OS handles such MoC variability across the system.

Whenever possible, our RTOS design is exposed to programmers through entities that are well-known to model-based practitioners. Processing is primarily abstracted through the *Actor* construct, which is accordingly mapped into a periodic real-time thread, an aperiodic real-time thread, or a best-effort thread. Actors communicate with each other through *Ports* and *Interfaces*. An Actor can have input and output Ports, which are respectively employed for reading and writing values. Interfaces connect output ports to input ports.

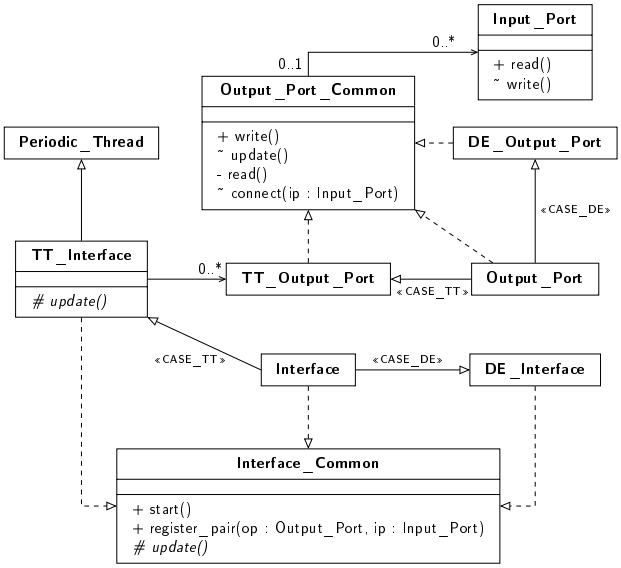

Fig. 3. Interfaces and Ports.

Figure 3 shows the relationship between Interfaces and Ports. Input and output Ports are connected to each other through an Interface via the `register_pair()` method. In a pair of output and input Ports, values flow from the output Port to the input port during the invocation of the `updated()` method of `Output_Port_Common`. Such a method traverses the list of input ports copying the value of the output port to the input ports. The value that an input Port holds has a validity that defines for how long such a value can be read without being updated. The validity time is specified in microseconds at the time the Port is created. It is usually set to the period of the system operation, or to a value higher than the period. A just created input Port is invalid for reading.

Actors can read from an input Port if the input port validity time has not yet expired. Otherwise, the reading Actor blocks on a Mutex until the port value becomes valid again. A Port is valid again whenever the input Port is updated as the values from output Ports gets propagated to it. During normal operation, input Ports are expected to be always valid, with invalid states being associated with delays or component failures (e.g. a sensor). In such cases, the system can be programmed either to raise an exception or (if it can tolerate delays) simply to wait[1].

In case of a CPS modeled around the DE MoC, which assumes *events* to be triggered along a discrete timeline, writing in an output Port immediately triggers the Port update and propagates the new value of that Port. In DE, Interfaces are used only to connect output to input Ports. A fundamental question that raises while mapping the DE MoC onto an RTOS is how to process events that have the same timestamp. In a DE simulator, the discrete time only goes forward after the simulator finishes processing all events tagged with the current timestamp. On a real system, however, this is not an option. We have therefore modeled DE execution by mapping events to real-time threads that carry the actions associated with the corresponding events. Conflicts are thus handled by the RTOS scheduler. Figure 4 shows how we have mapped the DE MoC onto RTOS components.

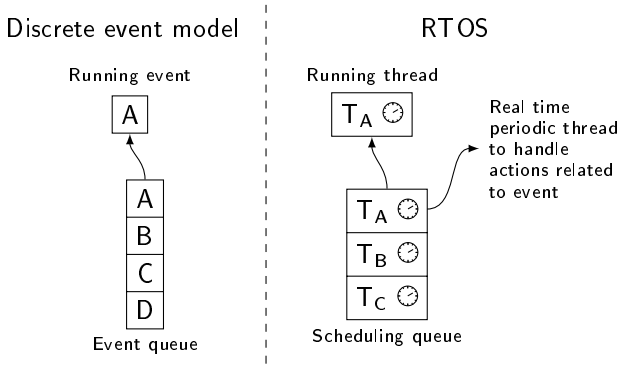

Fig. 4. Mapping of the DE MoC in our RTOS.

Interrupts in the DE MoC are seen as events that drive the application data flow. Therefore, traditional ISRs, which perform the reception, handling, and acknowledgment of interrupts at once, is the most convenient way to realize the communication of the Actor that generates the interrupts with others Actors. While using ISRs, the writing on an output Port is immediately followed by the propagation of the new value to the connected input Ports. For instance, a sensor would be modeled as an Actor that raises an interrupt whenever it produces new samples.

For the TT MoC, which uses the passing time instead of events to trigger actions [6], the Interface coordinates the propagation of values from output Ports to input Ports. A TT Interface maintains a list of registered output Ports that are to be updated on each activation through the invocation of the respective `update()` methods. The interaction of an Interface that handles data propagation and the Actor that writes on an output Port is coordinated by a Semaphore named

[1]Note that this mechanism does not aim to replace watchdog timers.

`updated`. An Actor writing to an output Port updates the port value and signalizes the `updated` Semaphore. On the other side, the Interface waits on the `updated` Semaphore and invokes its `update()` method.

Figure 5 shows how we have mapped the TT MoC onto traditional RTOS components: Subsystems are mapped to *Actors* and Interfaces (along with aspect of the communication system) are mapped to *Interfaces* (denoted as IF in the figure). Following the model proposed by Kopetz [6], the *Interface* coordinates the updating of Ports to Actors. That is achieved implementing an *Interface* as a periodic thread that, on each activation, propagates the values of the registered output Ports to all connected input Ports. Communication is managed by the interface, so Actors are implemented as aperiodic threads (denoted as Th in the figure) that are activated by the periodic threads representing the Interfaces.

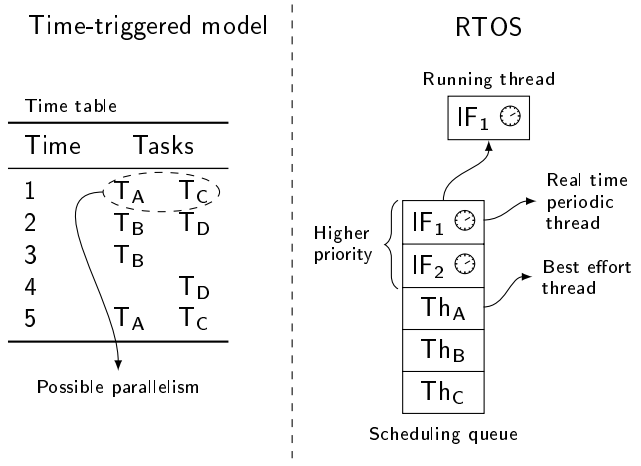

Fig. 5. Mapping TT onto RTOS.

In the case of the TT MoC, the occurrence of an interrupt must not disrupt the system timing mechanism used to coordinate TT Interfaces. Therefore, in this scenario the interaction between Actors that generate interrupts and TT Interfaces is better implemented using ISTs than ISRs, since they will cause less interference on the time perceived by Interfaces. A short ISR associated to an output Port is used to update the Port's value and to announce the interrupt occurrence by signaling the `updated` Semaphore. The periodic threads representing the connected TT Interfaces will subsequently propagate the updated Port value to other Actors. It is assumed that the period of the TT interface is smaller than (or equal to) the interrupt generation period so the thread can attend all interrupts (i.e. no interrupts are dropped). Alternatively, the size of the output Port internal buffer can be increased, and the TT Interface can have a period greater than the interrupt generation period.

IV. Abstracting Interrupt Handling with Concurrent Observer

During the implementation of the configurable MoC-oriented RTOS described here, we looked for means to confine the variability associated to ISRs and ISTs so it would not propagate widely across the system, eventually demanding many components to exist in multiple flavors to match particular MoC demands. Our answer to the question is the *Concurrent Observer* design pattern, which enables an efficient, and time-predictable decoupling of interrupt reception and servicing.

The *Concurrent Observer* is a concurrent variant of the *Publisher/Subscriber* design pattern (a.k.a. *Observed/Observer*) [2]. It is depicted in Figure 6. The main difference between Concurrent Observer and the original Publisher/Subscriber is that the `notify()` method of the Publisher does not invoke the `update()` method of its subscribers. Instead, it only notifies subscribers that a state change has occurred through a synchronization mechanism. Subscribers are modeled as threads that wait for such state change notification on the shared synchronization mechanism and then perform the updates.

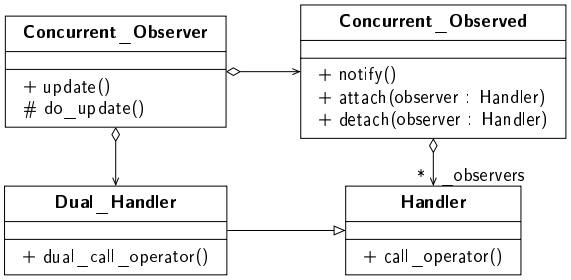

Fig. 6. Concurrent Observer.

The synchronization mechanism in the pattern is abstracted as a generic *Handler*, so programmers can select from a choice of Semaphore, Mutex, Condition Variable, or direct Thread operations like suspend and resume. The Handler is shared between the Concurrent Observer and the publisher (i.e. the Concurrent Observed), which holds a list of handlers as its subscribers. The `notify()` method of `Concurrent_Observed`, described in the sequence diagram of Figure 7, invokes the `call operator` for each handler. It should be noted that a *Semaphore Handler* is the most suitable for interrupt handling. Since a semaphore has memory, interrupts notifications are never lost even if the frequency with which interrupts are generated is higher than the frequency with which interrupts are serviced. In such a case, however, in order to prevent data loss (in the case there is data generation associated to the interrupt occurrence) a larger buffer and/or a ring buffer strategy should be employed.

The `update()` method of `Concurrent_Observer` depicted in Figure 8, invokes the "dual call operator" of its handler. For example, for a `Semaphore_Handler`, the operator is `v()` and its dual call operator is `p()`. From the point of view of the Concurrent Observer pattern, the call operator is the one used by the publisher to notify a change and the dual call operator is the one used by the subscriber to wait for a change.

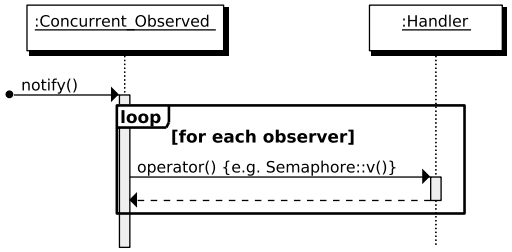

Fig. 7. *Concurrent_Observed::notify*.

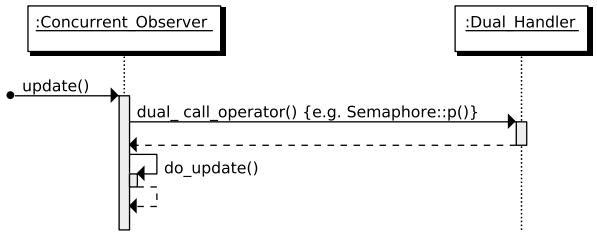

Fig. 8. *Concurrent_Observer::update*.

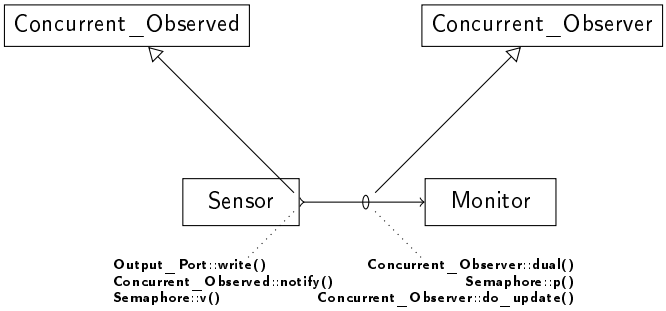

Fig. 9. Concurrent Observer used to implement TT output port communication.

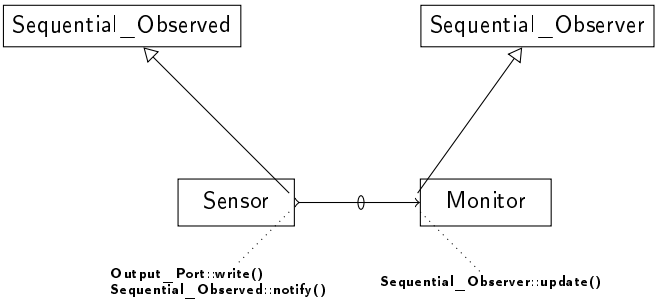

Fig. 10. Classic (Sequential) Observer used to implement DE output port communication.

It should be noted that the use of semaphores (or any other synchronization mechanisms) in Concurrent Observer does not introduce priority inversion problems since they are not being used to guard a critical section. They are are used only to synchronize the ISR with the corresponding IST, much in a Producer/Consumer way.

Taking into account the RTSS architecture presented in Section III, we have identified which design pattern better models the distinct interrupt handling strategies employed by each MoC.

As mentioned in Section III, in the case of the TT MoC, the communication between the actor that generates interrupts with others is driven by the TT interface that executes periodically following the application requirements. As the interaction between the sensor actor that produces the interrupts and the TT interface is concurrent, the *Concurrent Observer* design pattern is well suited for realizing such communication. In such a mapping, the sensor actor and its output Port (realized as a `Concurrent_Observed`) are assigned to the short ISR, producing and announcing interrupts. As depicted in Figure 9, connecting the sensor actor output Port and input port of a monitor (or controller actor) is the TT interface that is realized as a *Concurrent_Observer*.

As mentioned in Section III, in the case of the DE MoC, the communication of the actor that generates the interrupts with other actors is better performed by an ISR since the application data flow is driven by the event (interrupt) itself. As the interaction between the producer of the interrupt and its consumers is sequential, the classic *Observer* [2] design pattern is the most appropriate to realize such an interaction. In that case, the output Port of the sensor actor can be realized as a `Sequential_Observed` and each input port connected to it as a `Sequential_Observer` as shown in Figure 10.

As Figure 11 shows, for both cases (TT and DE), the communication between input port and actors can be realized by a variation of the *Concurrent Observer* pattern, named *Conditional Concurrent Observer*. The difference between `Conditional_Concurrent_Observer` and `Concurrent_Observer` is that the `dual` operator of the observer is invoked only if a certain condition is true. In the figure, the input port of a monitor or controller actor is realized as `Conditionally_Concurrent_Observed`, the actor itself as a `Conditional_Concurrent_Observer` and the condition is related to the port temporal validity. In such a case, the configuration is not necessarily related to interrupts but as a way to decouple in time the actor that "consumes" values from an input port that "produces" them. The handler shared by the input port and the actor, in that case, will be the "validity barrier" (modeled as a mutex) as mentioned in Section III that will block the actor in the case the value the port holds is not valid anymore (according to the input port validity parameter) and will unblock the actor when there is a new value on the port.

V. Evaluation

This section presents the results on evaluations for traditional ISR and IST according to average ISR time and interrupt latency. One goal of the experiment is to check how much the time of servicing an interrupt, named in the experiments as "consumer time" (CT), affects the time taken to service an interrupt (ISR time) and the interrupt latency (the time between the arrival of the interrupt and the time its servicing begins). Another goal of the experiment is to evaluate the proposed *Concurrent Observer* design pattern, which is used to

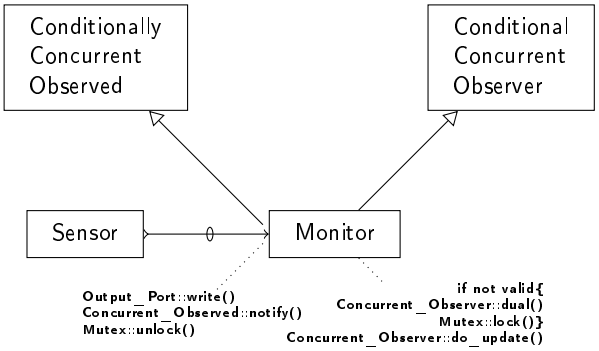

Fig. 11. Conditional Observer used to implement input port communication in DE and TT MoCs.

implement IST and the *Sequential Observer* used to implement ISR. In the case of traditional ISR, ISR time comprises interrupt reception, servicing and acknowledgement, while in the case of ISTs, ISR time comprises only interrupt reception and acknowledgment as interrupt servicing is performed by the interrupt servicing thread. Both experiments were conducted using a Core 2 Quad Q9550 PC platform running at 2.83 GHz. The experiments focus on interrupt generation and servicing. Other measurements concerning our MoC-based RTOS in distinct topologies and number of actors, ports, and interfaces are available in our previous paper [9].

In order to measure time, we have used a chronometer abstraction based on the Time Stamp Counter (TSC) provided by the processor. Values presented for ISR time represent the average of 1000 interrupts while values for interrupt latency represent the value of a single interrupt and respective servicing thread activation. All experiment treatments were replicated 10 times for statistical significance.

The experiments for both evaluations (for ISR and IST) were designed as a single-factor experiment using CT as the factor at six levels. The interrupt generation period (IGP) was fixed at 1ms. The levels used for CT were:

1) 1ms (100 % of IGP)
2) 600us (60 % of IGP)
3) 300us (30 % of IGP)
4) 0
5) 10ms (10 × IGP)
6) 100ms (100 × IGP)

CT generation was implemented by using a delay based on the TSC. In practice level 4 is not zero since the ISR takes time on the measurement itself. However, such time is constant and has the same value for all levels thus it does not appear in the experiment modeling.

For both experiments, we have applied single factor Analysis of Variance (ANOVA) to verify whether the distinct levels of the experiment have influence on time overhead. The factor effect is evaluated based on the statistical significance level (p-value), where p-value less than 0.05 (5 %) indicates that the investigated effect is considered statistically significant. Once the factor was considered significant, we have performed

TABLE I
AVERAGE ISR TIME.

	Level	Average ISR (μs)	Std. Dev. (μs)
ISR			
	100 % of IGP	1.01×10^3	0.138
	60 % of IGP	9.63×10^2	0.118
	30 % of IGP	9.84×10^2	0.956
	0	1.01×10^3	0.005
	10 × IGP	1.00×10^4	0.242
	100 × IGP	1.00×10^5	0.225
IST			
	100 % of IGP	1.00×10^1	0.000
	60 % of IGP	1.00×10^1	0.000
	30 % of IGP	1.00×10^1	0.000
	0	1.00×10^1	0.000
	10 × IGP	1.00×10^1	0.000
	100 × IGP	1.00×10^1	0.000

Tukey's multiple comparison test, which compares all pairs of means [11].

Table I shows average ISR time and the standard deviation for ISR and IST in all levels, taking into account all experiment replications.

In the case of ISR, the obtained p-value in ANOVA is less than 2×10^{-16} indicating that the difference between CTs has statistical significance. Therefore, in the case of ISR, CT does have influence on ISR time. The multiple comparison test indicates that there is no significant variation on ISR time for levels where CT is zero or a fraction of IGP (levels 1 to 4). For those cases, ISR time is dictated by the IGP itself being near to it (1ms). That makes sense since, for all experiments, an interrupt acknowledgment is issued after the execution of the corresponding ISR. However, for levels where CT is higher than IGP (levels 5 and 6) the ISR time grows as CT grows.

In the case of IST, the obtained p-value by ANOVA indicates that the difference between CT levels does not have statistical significance. Therefore, in the case of IST, CT does not influence ISR time (i.e. ISR time is constant). That is expected since the short ISR employed in the IST strategy does not perform the interrupt servicing, only interrupt reception and acknowledgment that in turn comprises the `notify` operation of `Concurrent_Observed` that signalizes its respective handlers.

Table II shows interrupt latency and the standard deviation for IST at all levels, taking into account all experiment replications. The obtained p-value by ANOVA is less than 2.85×10^{-6} indicating that the difference CTs has statistical significance. Despite being statistically significant, which means that the difference is not caused by random errors in the experiment, the difference between all levels in CT can be considered small in many practical situations (since it is less than $0.05\mu s$). In all cases the average interrupt latency is around 4.34 μs. In the case of ISR, by definition, there is no interrupt latency since the interrupt is received and serviced in the same context.

As demonstrated by the obtained results, ISR time and

TABLE II
INTERRUPT LATENCY FOR IST.

Level	Interrupt Latency (μs)	Std. Dev. (μs)
IST		
100 % of IGP	4.35	0.000
100 % of IGP	4.33	0.022
100 % of IGP	4.34	0.021
0	4.33	0.021
10 × IGP	4.35	0.002
100 × IGP	4.32	0.001

interrupt latency in IST is practically constant, regardless of CT. On the other hand, ISR changes as CT changes but does not suffer from interrupt latency. As shown in Figures 1 and 2, ISR time has a direct impact on the time a task activation will finish. A high ISR time means a high jitter and, consequently, a highly delayed task activation finish that can compromise the task deadline. In the case of interrupt latency, as discussed in Section III, a high interrupt latency will demand faster ISTs or larger buffers, so no interrupts are dropped.

VI. CONCLUSION

Interrupt handling is employed by many real-time embedded systems to implement data acquisition and communication, thus playing a fundamental role in such systems. Also, many real-time embedded systems are implemented following a Model of Computation (MoC) that defines the rules for computation and communication of its components. Consequently, interrupt handling must fulfill real-time requirements imposed by the application and respect the MoC that it is part of.

This paper evaluates the intersection of interrupt handling and MoC for DE and TT MoC. Such evaluations are conducted taking into account an RTSS architecture that is built relying on components commonly found in RTOSes.

We have evaluated ISR and IST, according to average ISR time and interrupt latency. Experiments demonstrate that ISR time is almost constant while using IST and changes completely while using ISR. Therefore IST is time-predictable while ISR demands a Worst-Case Execution Time (WCET) analysis for each handler. However, ISR present the advantage of having no interrupt latency since interrupt reception and servicing is performed in the same context. We find out that IST fits well on TT on the sensor-controller communication while ISR is more suitable for DE.

REFERENCES

[1] A. A. Fröhlich. *Application-Oriented Operating Systems*. GMD - Forschungszentrum Informationstechnik, Sankt Augustin, 2001.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[3] W. Hofer, D. Lohmann, and W. Schroder-Preikschat. Sleepy sloth: Threads as interrupts as threads. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 67–77, Nov 2011.

[4] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 3A. Intel Corporation, 2014.

[5] S. Kleiman and J. Eykholt. Interrupts as threads. *SIGOPS Oper. Syst. Rev.*, 29(2):21–26, Apr. 1995.

[6] H. Kopetz. The time-triggered model of computation. In *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, pages 168–177, Dec 1998.

[7] E. A. Lee. Heterogeneous actor modeling. In *Proceedings of the Ninth ACM International Conference on Embedded Software*, EMSOFT '11, pages 3–12, New York, NY, USA, 2011. ACM.

[8] L. Leyva-del Foyo, P. Mejia-Alvarez, and D. de Niz. Predictable interrupt scheduling with low overhead for real-time kernels. In *Embedded and Real-Time Computing Systems and Applications, 2006. Proceedings. 12th IEEE International Conference on*, pages 385–394, 2006.

[9] M. K. Ludwich, J. G. Reis, S. A. F. Soares, , and A. A. Fröhlich. Run-time support system for models of computation in cyber-physical systems. In *First Workshop on Cyber-Physical System Architectures and Design Methodologies*, page 6, New Delhi, India, Oct 2014.

[10] P. Marwedel. *Embedded System Design*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[11] D. C. Montgomery. *Design and Analysis of Experiments*. John Wiley & Sons, 6th edition, 2005.

[12] M. J. Pont and M. P. Banner. Designing embedded systems using patterns: A case study. *Journal of Systems and Software*, 71(3):201–213, May 2004.

[13] O. V. Portal. Osek vdx portal, 2008.

[14] J. E. Savage. *Models of Computation: Exploring the Power of Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1997.

[15] S. Tripakis, C. Pinello, A. Benveniste, A. Sangiovanni-Vincent, P. Caspi, and M. Di Natale. Implementing synchronous models on loosely time triggered architectures. *Computers, IEEE Transactions on*, 57(10):1300–1314, Oct 2008.