Specification of Embedded Control Systems Behaviour Using Actor Interface Automata

Christo Angelov, Feng Zhou, Krzysztof Sierszecki

Mads Clausen Institute for Product Innovation
University of Southern Denmark
Alsion 2, 6400 Soenderborg, Denmark
{angelov, zhou, ksi}@mci.sdu.dk

Abstract. Distributed Timed Multitasking (DTM) is a model of computation describing the operation of hard real-time embedded control systems. With this model, an application is conceived as a network of distributed embedded actors that communicate with one another by exchanging labeled messages (signals), independent of their physical allocation. Input and output signals are exchanged with the controlled plant at precisely specified time instants, which provides for a constant delay from sampling to actuation and the elimination of I/O jitter. The paper presents an operational specification of DTM in terms of actor interface automata, whereby a distributed control system is modeled as a set of communicating interface automata executing distributed transactions. The above modeling technique has implications for system design, since interface automata can be used as design models that can be implemented as application or operating system components. It has also implications for system analysis, since actor interface automata are essentially timed automata that can be used as analysis models in model checking tools and simulation environments.

Keywords: distributed control systems, component-based design of embedded software, domain-specific frameworks, distributed timed multitasking, interface automata

1 Introduction

Control-theoretic models of computer control systems assume a synchronous pattern of computer-plant interaction, featuring sampling at a constant frequency and synchronism, i.e. zero delay between sampling and actuation. These assumptions have been adopted in the *perfect* synchronous model of computation used in a number of programming languages and environments, such as LUSTRE, SIGNAL and ESTEREL [1].

In practice, the perfect synchronous model can only be approximated, because the control program has non-zero execution time C and response time R, $R = C + I$, where I denotes the extra time due to interference (preemption) from higher priority tasks, interrupts, etc. However, such an approximation is valid only when the control

computer is infinitely fast, or at least much faster than the plant controlled, such that its response time is orders of magnitude smaller than the sampling period.

That is obviously not the case when task response time is of the same order as its period. In that case, the synchrony hypothesis is not valid – there is no longer an effective zero delay between sampling and actuation. Consequently, the real system may exhibit a closed-loop behaviour being substantially different from the modeled one. This requires a re-design of the control system taking into account the computational delay. Unfortunately, the task response time R varies with different invocations because the interference due to higher priority tasks I fluctuates with different task phasings. That is why it is impossible to precisely model the influence of the computational delay in the control-theoretic model of the real system.

The variation of response time is ultimately demonstrated as input and output jitter, which is detrimental to control system behaviour. Its effect is substantial for control loops having small sampling time that is comparable to the task response time R. Theoretical and experimental investigations have shown that I/O jitter may result in poor quality of control and even instability [2].

The above problems can be dealt with by adopting a modified model, i.e. the *clocked* synchronous model of computation [3], which is characterized by a *constant*, non-zero delay from sampling to actuation (e.g. one-period delay). That is why it can be easily taken into account in the discrete-time continuous model of the control system (e.g. by using zero-order hold and unit delay blocks). In this way, the behaviour of the real system becomes identical to its modeled behaviour and I/O jitter is eliminated.

This approach is popular among control engineers and has been recently adopted in a number of software frameworks, e.g. Giotto [5], which employs time-triggered tasks executing at harmonic frequencies, whose deadlines are equal to the corresponding periods.

A general solution to the above problems is provided by a particular version of the clocked synchronous model known as Timed Multitasking [4], which can be used with both periodic time-driven, as well as aperiodic event-driven tasks. This model assumes that task I/O drivers are executed atomically at task release/deadline instants, whereas the task itself is executed in a dynamic scheduling environment. In this way, task I/O jitter is effectively eliminated as long as the task comes to an end before its deadline, which is defined to be less than or equal to period. On the other hand, Timed Multitasking provides for a constant delay from sampling to actuation, which can be taken into account in the discrete-time model of the closed-loop control system.

Distributed Timed Multitasking (DTM) is a model of computation developed in the context of the COMDES framework [8]. It extends the original model to sets of application tasks (actors) executing transactions in single-computer or distributed real-time environments. A *denotational* specification of that model is given in [9], where system operation is described by means of composite functions specifying signal transformations from input signals to output signals, taking into account the constant delay from sampling to actuation. However, a precise operational specification is still missing.

The purpose of this paper is to develop an *operational* specification of distributed control system behaviour in terms of interface automata [7]. These can be used to formally specify actor and system behaviour under DTM, which is a prerequisite for

deriving accurate analysis models. Interface automata can also be used as design models needed to develop an operating system environment supporting DTM.

The rest of the paper is structured as follows: Section 2 presents an informal introduction of Distributed Timed Multitasking focusing on the behaviour of system actors during the execution of phase-aligned transactions. Section 3 presents interface automata modelling the behaviour and interaction of embedded actors executing distributed transactions with hard deadlines. Section 4 discusses the implementation of interface automata as operating system components incorporated in the event management subsystem of a real-time kernel. Section 5 presents related research. The last section summarizes the main features of the proposed model and its implications.

2 Distributed Timed Multitasking: an Informal Introduction

Distributed Timed Multitasking (DTM) is a model of computation, which has been developed in the context of COMDES – a component-based framework for hard real-time embedded control systems [8]. In this framework, an embedded system is conceived as a network of active objects (actors) that communicate with one another via labelled state messages (signals) encapsulating process variables, such as *speed*, *pressure*, *temperature*, etc. (see e.g. Fig. 1). DTM extends Timed Multitasking to distributed real-time systems in the context of communicating actors and transparent signal-based communication [9].

An actor consists of a signal processing unit (SPU) operating in conjunction with I/O latches, which are composed of input and output signal drivers, respectively [9]. The input latch is used to receive incoming signals and decompose them into local variables that are processed by the SPU. The output latch is used to compose outgoing signals from local variables produced by the SPU and broadcast them to potential receivers. This is accomplished by means of communication primitives that make it possible to transparently broadcast and receive signals, independent of the allocation of sender and receiver actors on network nodes. Physical I/O signals are treated in a similar manner but in this case, the latches are used to exchange physical signals with the environment at precisely specified time instants.

A control actor is mapped onto a real-time task having three parts: *task input*, *task body* and *task output*, implementing the input latch, SPU and output latch, respectively. The task body is executed in a dynamic priority-driven scheduling environment. It is released by the event that triggers the actor for execution, i.e. a periodic timing event, external interrupt or a message arrival event. During execution it may be preempted by other higher-priority tasks running in the same node, and consequently – suffer from I/O jitter.

Task input and output are relatively short pieces of code whose execution time is orders of magnitude smaller than the execution time of the actor task, which is typical for control applications. They are executed atomically in logically zero time, in separation from the task body (split-phase task execution). Specifically, task input is executed when the actor task is released, and task output – when the task deadline arrives (see Fig. 2). Consequently, task I/O jitter is effectively eliminated as long as the task is schedulable and comes to an end before its deadline.

When a deadline is not specified, the task output is executed immediately after the task is finished. That is e.g., the case with the intermediate tasks of phase-aligned transactions, which have to generate output signals as soon as they are computed, whereas an end-to-end deadline is imposed on the entire task sequence executing the distributed transaction (see below).

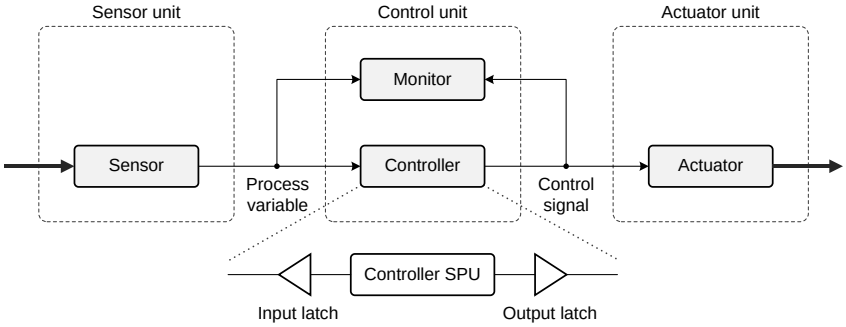


Fig. 1. COMDES model of a distributed embedded system

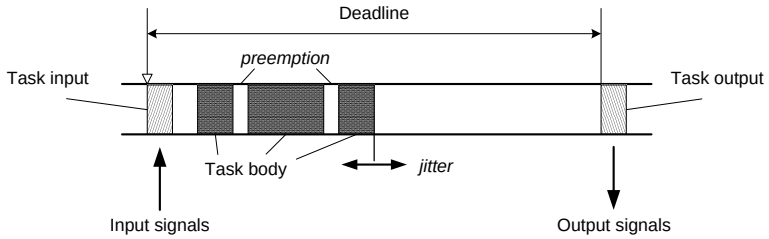


Fig. 2. Split-phase execution of actor under Distributed Timed Multitasking

The above techniques can be extended to task sets executing transactions in single-computer or distributed environments. These are treated in the same manner due to the transparent nature of signal-based communication, e.g. the phased-aligned transaction shown in Fig. 3, involving the actors *Sensor* (*S*), *Controller* (*C*) and *Actuator* (*A*) introduced in Fig. 1. This transaction is triggered by a periodic timing event, i.e. a synchronization (*sync*) message denoting the initial instant of the transaction period *T*, with deadline $D \leq T$.

In principle, such transactions suffer from considerable I/O jitter. That is due to task release/termination jitter as well as communication jitter, which is accumulated and ultimately – inherited by the terminal actor. However, in our case input and output signals are generated at transaction start and deadline instants, resulting in constant response time and the effective elimination of I/O jitter.

This resolves the main problem with phase-aligned transactions, which are otherwise simple to implement and commonly used in distributed applications. Transactions involving periodic tasks with the same or harmonic periods are also common for many applications and frameworks, e.g. Giotto [5]. In such transactions task deadlines are usually equal to task periods.

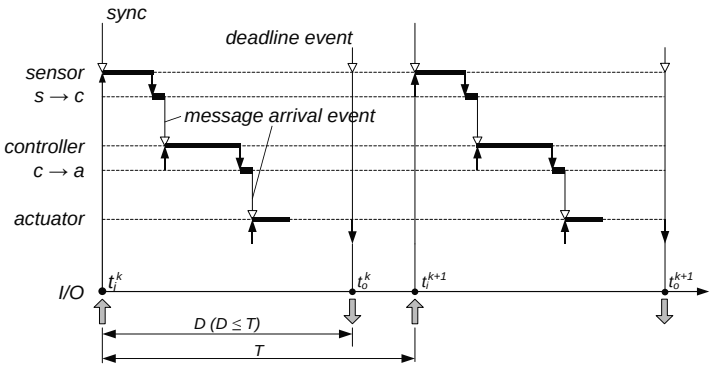


Fig. 3. Jitter-free execution of distributed transactions

The DTM model of computation is presently supported by the HARTEXμ kernel [10]. It has been experimentally validated in a number of computer control experiments, involving physical and computer models of plants such as DC motor, production cell, steam boiler, turntable, etc., as well as an industrial case study – a medical ventilator control system featuring fast, jitter-sensitive control loops with millisecond-range periods and deadlines.

3 Modelling Actor Behaviour with Interface Automata

Interface automata are state machine models used to describe the timed input/output behaviour of software components [7]. Specifically, they can be used to precisely describe the interaction between components in terms of communication protocols implemented by the interfaces of the interacting components, e.g. sequences of method invocations of the corresponding operational interfaces, or message exchanges involving interacting port-based interfaces, etc. In the context of DTM, interface automata can be used to formally specify actor behaviour and interaction, i.e. the behaviour of system actors executing a distributed transaction and communicating with one another via labelled state messages (signals).

The interface automaton can be viewed as an operational component which conducts the execution of a system actor. Accordingly, the actor can be modeled as a composition of input latch, signal processing unit (SPU) and output latch, which are controlled by the interface automaton (see Fig. 4). This model emphasizes separation of concerns: the SPU implements the functional behaviour of the actor in separation from its timed I/O behaviour, which is modeled by the interface automaton.

The interface automaton is enabled for execution by an external *start* event generated by another actor, and is subsequently triggered by periodically arriving tick events that are also used to update a timer measuring the corresponding period and/or deadline intervals. It generates control signals *get input* and *start* in order to activate the input latch and then start the SPU, which generates a *ready* signal when the computation is finished. Finally, the interface automaton generates the signal

produce output in order to activate the output latch and generate the corresponding output signals. In case of deadline violation, it generates the corresponding exception signal.

This kind of behaviour can be formally specified in terms of periodically executed, event-driven Mealy machines whose transitions are labeled with the corresponding < transition trigger/control signal > pairs. It is possible to define various types of actor interface automata, depending on the start event used to enable the actor and the type of transaction it is involved in. The following discussion assumes application scenarios featuring periodic tasks with harmonic periods or phase-aligned transactions that are common for distributed embedded systems. These can be implemented with several kinds of actor, as follows:

- Periodic time-driven actor with deadline less than or equal to period
- Event-driven actor triggered by either an external event or a message arrival event (e.g. a sync message, application message) – with or without deadline
- Event-driven actor triggered by a message arrival event – with deadline inherited from the transaction deadline (terminal transaction actor)

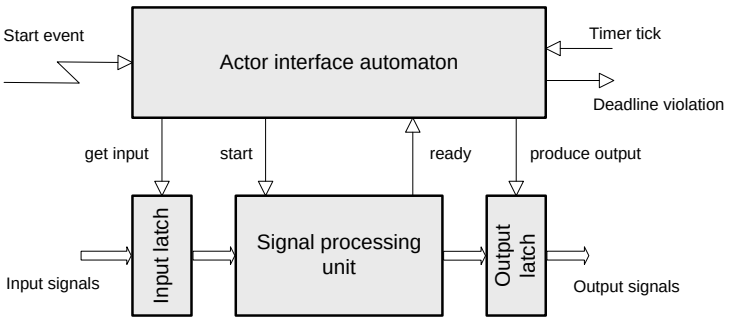
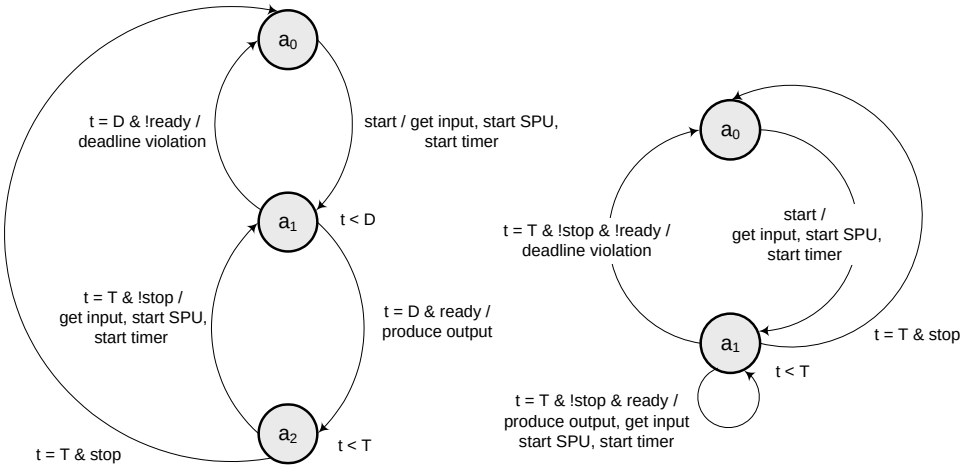


Fig. 4. Control actor modelled as a composition of signal-processing components controlled by an interface automaton

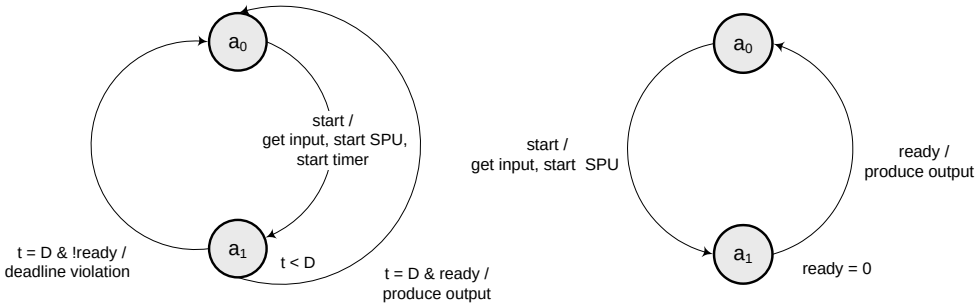
The interface automaton of a periodic time-driven actor with deadline less than period is shown in Fig 5-a. When started, the interface automaton generates the control signals needed to get input signals and start the SPU, starts a timer that will be used to measure both deadline and period, and makes a transition to state a_1 which will be maintained while current time is less than deadline. When $t = D$, a control signal is generated in order to produce output signals, provided that the SPU has finished computation (*ready = 1*), and a transition to state a_2 is made, which will be maintained while current time is less than period. When $t = T$, the interface automaton generates the signals needed to get input and start the SPU, restarts the timer and goes back to state a_1 (waiting until $t = D$). This mode of operation will be repeated over and over again up until a *stop* signal is issued by an external actor that will force a transition back to the initial state a_0. A transition to that state is also enforced in the case of deadline violation, when the interface automaton is in state a_1 and $t = D$ but the SPU has not finished computation (*ready = 0*).



a) with deadline less than period b) with deadline equal to period

Fig. 5. Interface automata of time-driven actors

Fig. 5-b shows the interface automaton of a periodic time-driven actor with deadline equal to period. Such actors are typically used in control applications featuring one-period delay from sampling to actuation as well as multi-rate control systems. The logic of the interface automaton is similar to the one discussed above, the only difference being the existence of a single wait state, which is maintained while $t < T$. When $t = T$, the interface automaton generates the control signals needed to produce output, get input, start SPU and restart timer, and goes back to a_1 waiting till the end of the period, and so on, until stopped.



a) with deadline greater than zero b) with deadline equal to zero (no deadline)

Fig. 6. Interface automata of event-driven actors

Fig. 6 shows interface automata for event-driven actors with and without deadline. The automaton of Fig. 6-a is enabled by the arrival of an external start event (e.g. external event, sync message or another message arrival event), whereupon it generates the control signals needed to get input, start SPU and start a timer measuring

deadline. When $t = D$ the automaton checks the *ready* feedback signal and generates the control signal needed to produce output (*ready = 1*) or a deadline violation exception (*ready = 0*).

An event-driven actor may not have a deadline, e.g. a non-terminal actor of a phase-aligned transaction that generates output signals immediately after its computation is finished (see Fig. 6-b). It can be shown that such an actor has a synchronous execution semantics ($D = 0$), whereas the terminal actor has a clocked synchronous semantics ($D = D_{trans}$), i.e. it inherits the end-to-end deadline of the transaction, such that its output signals are generated at the transaction deadline instant [9]. Fig. 7 shows an interface automaton describing the behaviour of a terminal transaction actor. The latter is triggered by a message arrival event but its deadline timer is started by the global start event when the transaction is released (see e.g. Fig. 3).

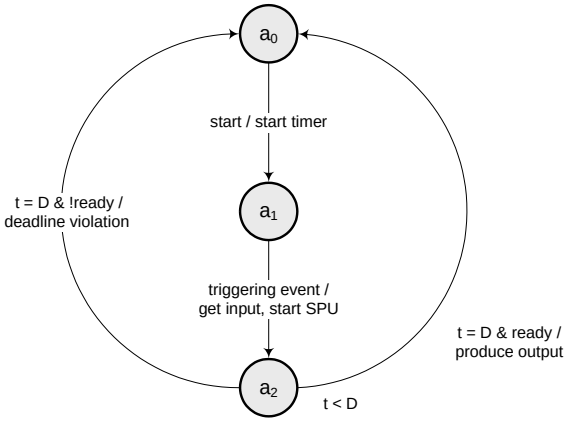


Fig. 7. Interface automaton for a terminal transaction actor

The presented modeling technique has implications for system design, since interface automata can be used as design models for operational components conducting the execution of system actors. It has also implications for system verification, since interface automata are essentially timed automata that can be used as analysis models for verification tools like e.g., Uppaal or simulation environments such as Simulink. In particular, interface automata can be used to develop analysis models that preserve the timed multitasking semantics of the design models. In that case, the actor is modelled by a pair consisting of an interface automaton and a functional automaton (or Simulink subsystem) modelling the behaviour of the SPU. Fig. 8 depicts such a construct using Uppaal notations.

The interface automaton is synchronized with another automaton (e.g. a sync message generator or a preceding actor) via the *start* broadcast channel, whereas the interface and functional automata synchronize with each other by means of the *start SPU* channel and the *ready* feedback signal. The functional automaton has two states - *initial* (a_0) and *executing* (a_1), whereby the transition from a_0 to a_1 is labelled with one or more functions from input to output signals, specifying the signal transformations performed by the SPU. State a_1 is a timed state with an invariant $t < R$, where t is the clock measuring task execution time and R is the task response time, which can be determined through response-time analysis. It is exited when $t = R$, whereby the feedback signal *finished* is set to *true.*

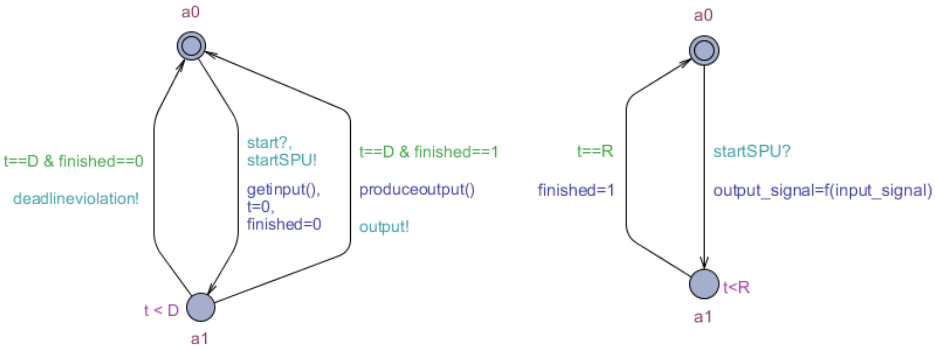


Fig. 8. Uppaal model of event-driven actor with deadline

The timed automata shown in Fig. 8 represent an event-driven actor with deadline, which can be triggered by external events such as global timing events, message arrival events, etc., modelled by the synchronization channel *start*. In a phase-aligned transaction, the corresponding signal will be generated by a predecessor actor via its *output* synchronization channel, which will be activated when the predecessor output is produced. In that case data can be exchanged via a shared data structure.

Similar models can be also developed for the other types of interface automata discussed in this section. Ultimately, the presented technique can be used to transform a system design model into a consistent analysis model accepted by verification tools, such as Uppaal. In particular, each actor is modelled as a pair of interface and functional automata, and actor interaction – as broadcast communication through shared data structures used by the corresponding interface automata, as well as broadcast synchronization channels modelling message arrival events.

4 Implementation Aspects of Interface Automata

The conceptual actor model of Fig. 4 can be used as a design model, whereby the interface automaton is implemented as an application component (state machine) interacting with the input latch, signal-processing unit and output latch. The interface automaton can also be viewed as an operating system component, which processes relevant events in accordance with the corresponding behavioural pattern, and conducts the execution of the actor by invoking the necessary kernel primitives. This interaction is explained below in more detail, in the context of the HARTEXμ kernel [10].

Interface automata can be implemented as service routines that are invoked by the kernel Event Manager while processing the corresponding timing and external events, whereas the signal-processing units are mapped onto actor tasks (task bodies). Likewise, I/O latches are implemented as task interface routines – task I/O, which are executed in separation from the main functions of the corresponding tasks.

In that case the interaction between the interface automaton and the actor task is accomplished by means of the corresponding kernel primitives (see Fig. 9). The primitive *release(task)* is used to implement the control actions *get input* and

start SPU of the conceptual model. Specifically, it is used to register a task - its input and main body for execution in the corresponding kernel data structures, i.e. Boolean vectors that are used to emulate system queues [10]. Likewise, the primitive *finish(task)* is used to implement the action *produce output* by registering the task output in the corresponding kernel vector.

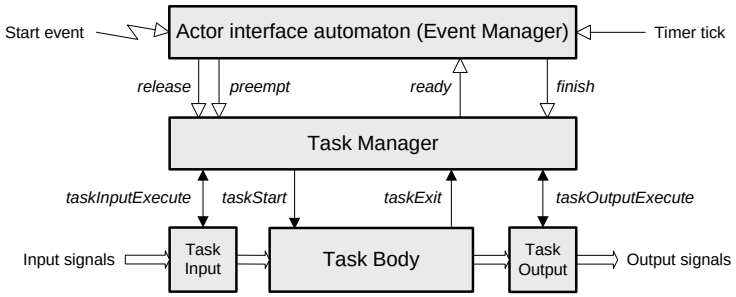


Fig. 9. Interaction between actor interface automaton and actor task

The Event Manager may be invoked by interrupt service routines processing external or tick interrupts. When activated, it processes all interface automata that are enabled for execution, which may result in one more newly released tasks having higher priority than the currently running task. Therefore, the execution of the Event Manager ends up with the invocation of the Task Manager primitive *preempt()*, which executes *atomically* all registered task outputs and then task inputs in order to observe the precedence relation between producer and consumer actors exchanging signals at the particular time instant; it proceeds further by selecting the highest-priority task to execute from among all registered tasks and the currently running task, which may be preempted or continued depending on task priorities. A high-priority task will be started immediately, whereas a lower priority task will wait for its turn to execute and will be eventually started when the previously running and/or registered higher priority tasks come to an end. The execution of a running task is finished by returning to the Task Manager, which resets its registration bit, thereby generating the feedback signal *ready*.

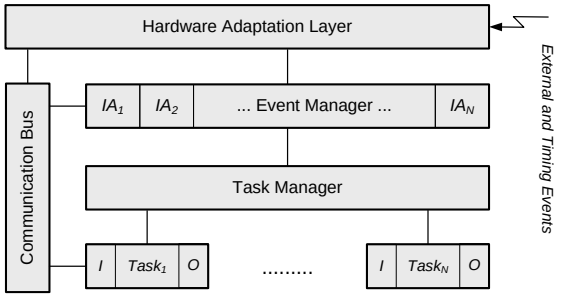


Fig. 10. DTM kernel architecture

The presented mechanisms constitute a new solution, which allows for very fast and uniform treatment of events and tasks in the context of a real-time kernel consisting of three main subsystems: *Event Manager* incorporating actor interface automata, *Task Manager* scheduling the execution of actor tasks, and a *Communication Bus* used to transparently exchange signals between communicating actor tasks via broadcast event notification and communication primitives (see Fig. 10). More details about the implementation of these and other kernel primitives can be found in [10].

5 Related Research

Distributed Timed Multitasking has been inspired by the original *Timed Multitasking* model [4] and is similar to the logical execution time (LET) model adopted in the *xGiotto* language [6]. However, both of these models employ port-based communication between actors, whereas DTM employs broadcast communication using labeled state messages (signals). This is a flexible solution, which rules out artifacts such as ports, mailboxes, operational interfaces with call-return semantics, etc., thus providing for reconfigurable and truly open distributed embedded systems.

The adopted communication mechanism supports transparent communication and is characterized by complete separation of computation and communication, since signal drivers are executed in separation from actor tasks and from each other. That is not the case with port-based objects, where ports are usually defined as communication objects whose methods are invoked within task I/O drivers in a conventional call-return fashion, much in the same way as operational interfaces, see e.g. [4]. Consequently, the communication pattern is 'hardwired' in the code of I/O drivers and cannot be reconfigured without reprogramming. Furthermore, in that implementation ports are conceived as shared data structures, which are not suitable for distributed applications.

DTM has certain similarities with the models of computation used in synchronous languages [1]. At the same time, there are some notable differences, and in particular:

- True actor-level concurrency vs. conceptual concurrency, which is 'compiled away' during the translation of synchronous programs
- Constant *non-zero* reaction time vs. instantaneous (zero-time) reaction assumed by perfectly synchronous systems.

The last feature facilitates the engineering of distributed systems and eliminates major problems related to fixpoints, instantaneous loops, etc. Furthermore, the synchronous model does not address the problem of I/O jitter because of the very nature of the synchrony hypothesis, whereas it is practically eliminated with DTM due to the constant delay from sampling to actuation inherent to that model.

6 Conclusion

The paper presents an operational specification of Distributed Timed Multitasking in terms of actor interface automata. This modeling technique has implications for

system design, since interface automata can be used as design models implemented as application or operating system components. It has also implications for system verification, since interface automata are essentially timed automata that can be easily transformed into analysis models used in model-checking tools like Uppaal or simulation environments such as Simulink.

Interface automata can be used to efficiently implement the event management subsystem of a DTM kernel, which allows for fast and uniform treatment of events and tasks in an operational environment consisting of three main subsystems: *Event Manager* incorporating actor interface automata, *Task Manager* scheduling the execution of actor tasks, and a *Communication Bus* used to transparently exchange labeled messages (signals) between communicating actor tasks.

This architecture has been used to implement the latest version of the HARTEXμ kernel. Research is now going on, aimed at a hardware implementation of the Event Manager and eventually – the entire kernel, with application tasks running in the soft cores of a multi-core FPGA chip.

7 References

1. A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic and R. de Simone, "The Synchronous Languages 12 Years Later", *Proc. of the IEEE*, vol. 91, No 1, Jan. 2003, pp. 64-83
2. P. Marti, R. Villa, J.M. Fuertes, and G. Fohler, "On Real-Time Control Task Schedulability", Proceedings of the European Control Conference, Porto, Portugal, Sept. 2001, pp. 2227-2232
3. A. Jantsch, *Modeling Embedded Systems and SoCs - Concurrency and Time in Models of Computation*, Morgan Kaufmann, 2003
4. J. Liu and E.A. Lee, "Timed Multitasking for Real-Time Embedded Software", *IEEE Control Systems Magazine: Advances in Software Enabled Control*, Feb. 2003, pp. 65-75
5. T.A. Henzinger, B Horowitz, and C.M Kirsch, "GIOTTO: a Time-Triggered Language for Embedded Programming", *Proc of the IEEE*, v. 91 (2003), pp. 84-99
6. A. Ghosal, T.A. Henzinger, C.M. Kirsch and M.A. Sanvido, "Event-Driven Programming with Logical Execution Times", Proc. of HSCC 2004, *LNCS 2993* (2004), pp. 357-371
7. L. de Alfaro and T.A. Henzinger, "Interface Automata", Proc of the 8th European Software Engineering Conference ESEC 2001, Austria, 2001
8. C. Angelov, X. Ke and K. Sierszecki, "A Component-Based Framework for Distributed Control Systems", Proc. of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications SEAA 2006, Cavtat, Dubrovnik, Croatia, Aug.-Sept. 2006, pp. 20-27
9. C. Angelov, K. Sierszecki and Y. Guo, "Formal Design Models for Distributed Embedded Control Systems", Proc. of the 2nd International Workshop on Model Based Architecting and Construction of Embedded Systems ACES-MB 2009, Denver, Colorado, USA, Oct. 2009, pp. 43-57
10. K. Sierszecki, C. Angelov and X. Ke, "A Run-Time Environment Supporting Real-Time Execution of Embedded Control Applications", Proc. of the 14th International IEEE Conference on Embedded and Real-Time Computing Systems and Applications RTCSA 2008, Kaohsiung, Taiwan, Aug. 2008