Run-Time Support System for Models of Computation in Cyber-Physical Systems

Mateus Krepsky Ludwich, João Gabriel Reis, Sérgio Aurélio Ferreira Soares, and Antônio Augusto Fröhlich
Laboratory for Software and Hardware Integration (LISHA)
Federal University of Santa Catarina (UFSC)
P.O.Box 476, 880400900 - Florianópolis - SC - Brasil
{mateus,jgreis,sergio,guto}@lisha.ufsc.br

ABSTRACT

Models Of Computation (MoCs) define the rules for computation and communication for components in a computational system. One largely employed approach for mapping MoCs onto Cyber-Physical Systems (CPSs) is to use an automatic code generation tool, which has limits both in size and system complexity. This paper introduces an open source runtime support system (RTSS) for the Discrete Event (DE) and the Time-Triggered (TT) MoCs. The RTSS maps abstractions found in MoCs onto components commonly found in modern Real-Time Operating Systems (RTOS). We have evaluated the time overhead introduced by the RTSS through experiments with different number of components and interconnection between them. Based on the statistical analysis of the results, regression models were developed to estimate the time overhead given the number of elements used in the system design. The analysis shows a small time overhead caused by the RTSS (in the order of hundreds of microseconds). Additionally, the proposed RTSS was used to implement a CPS that controls the room temperature, humidity, and CO_2 levels inside a Smart Room.

Categories and Subject Descriptors

F.1.1 [**Computation by Abstract Devices**]: Models of Computation; D.4.7 [**Operating Systems**]: Organization and Design—*real-time systems and embedded systems*

General Terms

Design, Measurement, Performance.

Keywords

Models of Computation, Run-Time Support System, Cyber-Physical System, RTOS, Time-Triggered Architecture, Discrete Event Architecture, Embedded Systems.

1. INTRODUCTION

A modern Cyber-Physical System (CPS) is usually designed and implemented following some sort of model-based methodology [2, 4]. 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) [7]. Well established tools such as SIMULINK and PTOLEMY II support the modeling and simulation of CPS under a given MoC [2]. However, the tool-assisted implementation of such CPS is usually carried ignoring the interfaces established by contemporary Embedded Operating Systems. For instance, the *synchronous reactive* MoC is often implemented by a cyclic executive interwoven with the embedded application itself. In some other cases, proprietary infrastructures are used along the synthesis of components from models. For instance, the TTTECH solution for the *time-triggered* MoC, which has been successfully employed by the avionics industry [1]. In our view, binding the software elements of a CPS this way is mostly a bad thing to do, either due to the lack of portability or because these approaches do not scale well with the growing complexity of real systems. Portable, architecture-independent, reusable software assets can bring CPS projects closer to current Software Engineering practices.

This paper introduces an open source Run-Time Support System (RTSS) for the implementation of CPS software components, designed following traditional MoCs, on top of operating system abstractions available in most contemporary RTOSs [1]. The proposed RTSS was designed following a domain engineering process of both domains, CPS and RTOS. MoC elements such as actors, ports, and interfaces are abstracted and subsequently mapped onto RTOS abstractions according with the selected MoC. The RTSS was designed considering the constant interaction between software and the environment at real-time, and therefore does not add any indeterminacy to the underlying RTOS. Initially, two MoCs have been addressed: Discrete Event (DE) and Time-Triggered (TT). The choice of these two MoCs was mostly motivated by the fact that TT is widely accepted as a successful case in the CPS community and DE is often taken as a challenge whenever it is deployed out of the realm of simulations. By demonstrating how we consistently mapped these two MoCs onto ordinary RTOS abstractions we establish the guidelines for the mapping of virtually any other

[1]The source code for the proposed RTSS can be obtained at http://epos.lisha.ufsc.br

MoC.

The remaining of this paper is organized as follows: Section 2 presents an overview of existing approaches that aim at supporting the implementation of MoCs; Section 3 introduces the proposed run-time support system and explains how it was designed to support DE and TT MoCs; Section 4 evaluates the proposed run-time support system time overhead; Section 5 presents the implementation of a smart room as a case-of-study of the proposed run-time support system; Section 6 closes the paper with our final considerations.

2. RELATED WORK

Maraninchi and Bouhadiba propose the "42" component model for heterogeneous embedded systems (i.e. embedded systems following different MoCs) [3]. The goal is to provide a way to express timing aspects and several types of concurrency as models of computation. Their approach is inspired by PTOLEMY and thus captures some of its main aspects, including as micro-steps, macro-steps, and the Director (called controller in their work). It is an abstract framework that supports the specification of components independently from a programming language. Component elements such as inputs, outputs, and controls are abstracted similarly to a Finite State Machine, with inputs defining activation conditions for transitions that trigger specific actions (finite steps, but non necessarily deterministic) which consumes the inputs and generates control and data outputs. Components can be modeled hierarchically, but in this case a controller must be provided. Controllers describe (using a programming language) how components are connected and also define which actions are taken in which order. The concepts and artifacts of "42" are not mapped onto any specific RTOS, so we believe it could be easily combined with the one proposed in this work.

Lee presents a strategy to mix different models of computation (MoCs) for the design of complex embedded systems [2]. Complex models are hierarchically subdivided in a tree of nested sub-models that are homogeneous at each level. Sub-models are connected to form a network of interacting actors. A component called Director specifies which MoC (also called domain) is being used by an actor, defining its control, communication, and timing semantics. The key concept in this hierarchy is the isolation of coordination and communication from the actual functionality of individual actors. Therefore, besides helping to design and organize complex systems by organizing them hierarchically, it also increases the reusability of components and models. Nevertheless, the synthesis of components is not itself hierarchical and does not tackle implementation aspects for an RTOS. Similarly to Maraninchi and Bouhadiba, Lee's proposal could be enhanced by properly mapping the elements defined to abstract the MoC to a contemporary RTOS interface.

To Pont and Banner, the use of design patterns can simplify the development of embedded systems, specially time-triggered, cooperatively scheduled (TTCS) [6]. They claim that the use of design patterns does not exclude previously used methodologies but can be coupled with them. Moreover, 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. While this work focuses on applications composed of higher level abstractions such as actors and interfaces, Pont and Banner map, on-the-fly, tasks used by the applications onto TTCS.

Tripakis et al. proposed an approach for implementing synchronous model based systems on top of an abstract loosely time-triggered architecture (LTTA) [8]. 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-trigger MoC to relatively straightforward OS services. Nevertheless, the work focuses on semantic preservation between model and implementation and does not explicitly deals with the issues pertaining mapping different MoCs onto RTOS services or components.

3. RUN-TIME SUPPORT SYSTEM

This section introduces the proposed run-time support system (RTSS) for MoCs. The implementation of the RTSS requires some abstractions commonly found in RTOSs. Such abstractions are: (I) A real-time scheduling service encompassing real-time schedules policies such as Rate-Monotonic (RM), Earliest Deadline First (EDF), etc.; (II) Real-time processes or threads following the real-time scheduling policies, such as periodic threads or processes; (III) Synchronization mechanisms such as semaphores, mutexes, and condition variables; (IV) Timers and Alarms that can, respectively, count time and call functions when a timeout is reached. The proposed RTSS currently supports Discrete Event (DE) and Time-triggered (TT) models of computation.

3.1 Discrete Event

The DE MoC is characterized by *events* processed by the system following a discrete passing of time. The usual structure of DE MoC comprises an event queue that contains events to be processed. Each event contains a timestamp that indicates when it must be handled and an event identifier/type to distinguish the event from others. At a given time T the DE system process all events in the event queue that have a timestamp equals to T. Finally, the processing of events can generate new events with timestamps set to the future.

A fundamental question that rises while mapping 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 with the current timestamp. That is not the reality, however, of a CPS where the time is the time of the environment and, therefore, a real-time support is required. Another question is how to deal with the creation of events scheduled to the future, events that are resulting of the processing of the events with the current timestamp.

In order to map the DE onto an RTOS this work follows two main assumptions: (A1) the processing of an event in a CPS require the execution of one or more actions such as reading a sensor, a computation, and writing to an actuator; (A2) the majority of the events that are scheduled to the future in a CPS are other instances of the events being processed in the present. In other words, such events are *periodic*.

Figure 1 shows how we have mapped the DE MoC onto

RTOS components. With the Assumption A1 in mind, instead of creating an event queue as in DE simulators and processing each event of that queue, we have mapped events to threads that execute event actions. Therefore, the event queue becomes the scheduling queue that is handled by the RTOS. In order to handle the periodicity of events, as described by Assumption A2, we have employed periodic threads. Thus, an *Actor* in DE is a periodic thread that executes its actions from time to time. In the case of non-periodic events, a periodic thread can still be used as long it is set to run only once, which is done by configuring *Actor* accordingly.

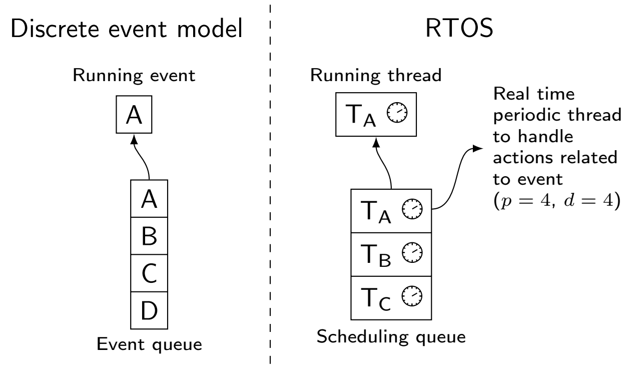


Figure 1: Mapping DE onto RTOS.

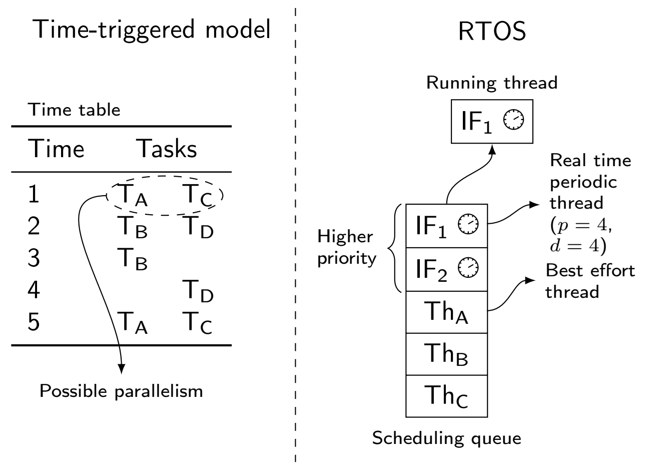
3.2 Time-Triggered

The TT MoC is characterized by *subsystems* connected to each other using *interfaces* and coordinated by a *communication system* [1]. In TT time passes according to real-time, which already corresponds to the reality of CPSs. Differently from event-triggered systems, in TT the passage of the time is responsible for triggering actions, not the occurrence of an event. In the original TT model proposed by Kopetz and others, all communication, performed using *interfaces*, is managed by the *communication system* and is performed according to a priori known time table. Therefore, all subsystems must follow such time table in order to operate properly.

Figure 2 shows how we have mapped the TT MoC onto RTOS components. In the proposed RTSS, subsystems are mapped to *Actors* and interfaces and the communication system are mapped to *Interfaces* (denoted as *IF* in the figure). Following the model proposed by Kopetz and others, the *Interface* coordinates the updating of ports of actors. In order to do so, an *Interface* is implemented as a periodic thread that, on each activation, propagates the values of the registered output ports to all input ports connected to such output ports. As the communication is managed by the interface, actors in TT are implemented as best-effort (non-periodic) threads.

3.3 RTSS Architecture

The main processing element in the proposed RTSS is the *Actor* that can be a periodic real-time thread or a best effort thread. Actors communicate to each other by using ports and interfaces. An actor can have input and output ports, which are respectively employed for reading and writing values. Interfaces connect N number of output ports to N input ports.



Time	Tasks	
1	T_A	T_C
2	T_B	T_D
3	T_B	
4		T_D
5	T_A	T_C

Figure 2: Mapping TT onto RTOS.

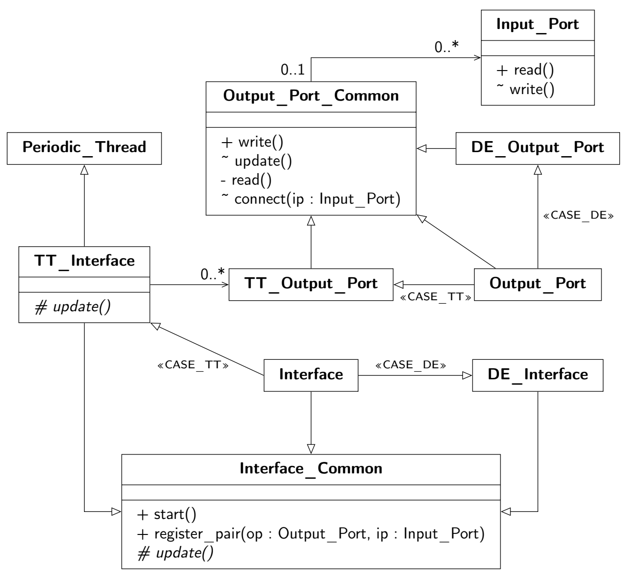


Figure 3: Relation between interfaces and ports.

Figure 3 shows the relationship between interfaces and ports. Input and output ports are connected to each other through the interface (*register_pair* method). In a pair of output and input port, values flow from the output port to an input port up to the invocation of the *updated* method of *Output_Port_Common*. Such a method traverses the list input ports copying the value from the output port to the input ports.

3.3.1 Ports Reading

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 on the creation of an input port. The validity time of an input port is usually set as the period of the system operation, or a value higher than the period. A just created input port is invalid for reading.

An actor can read from an input port if the input port validity time has not yet expired. Otherwise, the actor waits on a "validity barrier" (modeled as a *Mutex*) until the port value is valid again. A port is valid again whenever the input port is updated as the values from output ports are propagated to the input ports. The validity time is controlled by a chronometer that relies on an RTOS timer.

During the normal operation of the system, input ports are expected to be always valid. A port that is judged as invalid has not been updated in a time greater than the port validity time. In practice, that could indicate a failure in a system component (e.g. a sensor). In such cases, a special handler (e.g. which triggers an alarm or emergency procedure) might be called before waiting on the "validity barrier" *mutex*.

3.3.2 Ports Writing and Update

In the case of DE, writing in an output port triggers the port update and propagates the new value of that port. First the *Output_Port_Common::write* method is called, performing the value updating of the port. Then the method *Output_Port_Common::update* is called, copying the just updated output port value to every input port connected to the output port. In DE, interfaces are used only to connect output ports with input ports. Since the propagation of a value update is triggered by an write on an output port, the interfaces in DE are passive elements and perform no action.

In the case of TT, the interface coordinates the propagation of values from output ports to input ports. A TT interface maintains a list of output ports registered on such interface. Then, on each activation the TT interface invokes its *update* method that, in turn, invokes the *update* method for each output port registered on the interface and executing the actual value propagation.

The interaction of an interface that is responsible for data propagation and the actor that writes on an output port is coordinated by a pair of semaphores named "clear to update" and "updated". The actor that writes on the output port waits on the "clear to update semaphore" whenever is needed, performs the port update, and finally signalizes the "updated" semaphore. On the other side, the interface waits on the "updated" semaphore, invoke its *update* method, and signalize on the "clear to update" semaphore that actors may write new values on the output ports.

4. EVALUATION

We have performed two experiments to evaluate the time overhead caused by the use of the proposed RTSS. The first experiment aims to detect which factor or combination of factors has the biggest influence on the overhead. For that experiment, factors are the number of actors, interfaces, and pairs of output and input ports. The second experiment aims to check whether the topology of the system (which is defined by the form that actors, interfaces, and ports connect to each other) has influence on the overhead or not. Both experiments were conducted using the EPOS Mote II platform, which comprises an ARM-based processor running at 24 MHz with 96KB of RAM and 128KB of flash memory.

The time overhead was measured by connecting actors to each other (using interfaces and ports) being these actors with no actual behavior: the only actions performed by these actors are to read from their input ports and write on their output ports. As there is no useful computation inside actors, all measured time is considered overhead and therefore there is no need to compare the MoC implementations of the proposed RTSS to any "baseline" implementation. In all experiments, the application was configured to run forever since this is the reality of most CPS applications. We have measured the elapsed time of one "cycle" of the system execution. A cycle is the execution of one activation of all actors in the topology. We have measured the elapsed time from the moment that the first actor in a topology starts to execute until the time the last actor finishes. In order to measure the time, we have used an oscilloscope and the General Purpose Input Output (GPIO) pins of EPOS Mote II. When the first actor starts to execute, a GPIO pin is set high and when the last actor finishes it is set low. For all experiments, we have used the Deadline Monotonic (DM) scheduling policy. The deadline was set to be equals to the period.

4.1 Experiment 1: Number of Elements

The first experiment aims at answering the question: which system element (actor, interface, or pair) or a combination of elements influences the execution time the most? This experiment was designed as a full factorial 2^3 experiment (i.e. three factors at two levels each) [5]. The factors and respective levels are: (I) number of actors (levels 2 and 10); (II) number of interfaces (levels 1 and 9); (III) pair of output and input ports (levels 1 and 9).

Levels represent the number of instances on each factor (e.g. level one for factor "number of actors" uses two actors). Factors are the controllable variables in the experiment. This experiment uses the same topology for all its treatments (combination of levels for each factor). Figure 4 illustrates such topology for each evaluated treatment. It is a topology where actors are connected sequentially. As the topology is the same, the variation in the execution time is expected to be caused by the number of system elements.

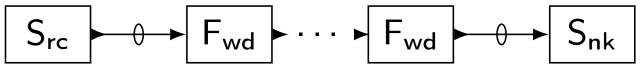


Figure 4: Topologies used for Experiment 1.

We have applied Analysis of Variance (ANOVA) multi-factor for each MoC in separate to verify which factor (or factor interaction) have influence at the 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. We also have generated for DE and TT their regression model, which defines the expected time overhead for a given experiment level.

In the case of DE, the obtained p-value by applying ANOVA was lesser than 2.2×10^{-16} for all factors and factors interactions, indicating that the difference between levels has statistical significance. Therefore, in the case of DE, all factors: pair of ports, interfaces, and actors and their interactions have influence on time overhead.

Equation 1 presents the regression model for predicting the time overhead while using DE. As all factors and factors interactions are statistically significant, all of them contribute to the time overhead (denoted by the response variable "Y"). Variables "XP", "XI", and "XA" represented the coded variables for each factor: ports, interfaces, and actors, respectively [5].

In the case of TT, the obtained p-value was lesser than 2.2×10^{-16} for all factors and factors interactions, indicating

Table 1: Time overhead for distinct topologies.

Topology	Average (μs)	Std. Dev. (μs)
A_{DE}	150.10	0.00
B_{DE}	137.90	0.00
C_{DE}	160.20	0.00
D_{DE}	149.23	0.05
A_{TT}	100.38	0.04
B_{TT}	101.41	0.03
C_{TT}	101.54	0.05
D_{TT}	101.45	0.05

that the difference between levels has statistical significance. Therefore, in the case of TT, all factors: pair of ports, interfaces, and actors and their interactions have influence on time overhead.

$$\begin{aligned} Y = 1.273 \times 10^2 &+ 1.301\ XP + 1.327\ XI + 1.325\ XA \\ &+ 1.296\ XP\ XI + 1.324\ XP\ XA \\ &+ 1.298\ XI\ XA + 1.329\ XP\ XI\ XA \end{aligned} \quad (1)$$

Equation 2 presents the regression model for predicting the time overhead while using TT. TT is more sensitive to the variation of the number of elements than DE. As indicated by equation 2, the number of interfaces is the major reason for that. This result is expected since interfaces are the ones responsible for propagating port values. In DE, a value written to an output port is propagated immediately. As all actors are periodic, there is no risk of overwriting a value that was not yet consumed. However, in TT values must be propagated at the specific time determined by the interface periods.

$$\begin{aligned} T = 2.218 \times 10^2 &+ 4.094 \times 10^1\ XP + 8.979 \times 10^1\ XI \\ &+ 1.138 \times 10^2 XA + 4.094 \times 10^1\ XPXI \\ &+ 4.367 \times 10^1\ XI\ XA \\ &+ 4.094 \times 10^1\ XP\ XI\ XA \end{aligned} \quad (2)$$

4.2 Experiment 2: Topologies

The second experiment aims at answering the question: has the form that actors, interfaces, and ports are connected (i.e. the system topology) influence on the time and memory overhead of the RTSS? This experiment is complementary to the Experiment 1 in the sense it fixes the number of elements and variates only how these elements are connected to each other. Therefore, we have designed this experiment as a single-factor experiment [5]. The factor, in this case, is the topology. The different levels are the different topologies the system may assume. Figure 5 shows the distinct topologies used in this experiment. For this experiment the number of actors, interfaces, and pairs of output and input ports where fixed in two instances of each.

Table 1 shows the obtained time overhead while using the DE and TT MoCs. Column *topology* corresponds to the topologies of Figure 5. The "DE" subfix indicates the DE MoC and the "TT" subfix indicates the TT MoC. We have applied single factor Analysis of Variance (ANOVA) for each MoC in separate to verify whether the distinct levels of the experiment have influence on time overhead. Once the factor was considered significant, we have performed multiple comparison test through the Tukey test method, which compares all pairs of means [5].

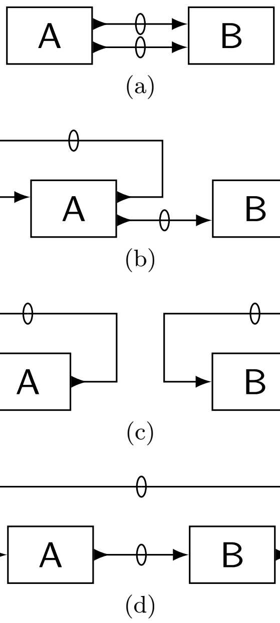


Figure 5: Topologies used for Experiment 2.

In the case of DE, the obtained p-value in ANOVA is lesser than 2×10^{-16} indicating that the difference between topologies has statistical significance. Therefore, in the case of DE, topology does have influence on time overhead. The multiple comparison test indicates that there is no special overhead caused by topologies employing feedback (topologies B, C, and D).

In the case of TT, the obtained p-value by ANOVA is lesser than 2×10^{-16} indicating that the difference between topologies has statistical significance. Therefore, in the case of TT, topology does have influence on time overhead. The multiple comparison test indicates that topologies presenting feedback have a larger time overhead than Topology A, which has no feedback. Despite being statistically significant, which means that the difference is not caused by random errors in the experiment, the difference between all topologies in TT can be considered small in many practical situations (since it is lesser than $1\mu s$).

5. CASE STUDY

For demonstrating the feasibility of the proposed approach, we have modeled and implemented a Smart Home environment. The system can monitor and control the temperature, humidity and CO_2 concentration. Sensors and actuators were wirelessly interconnected inside the room. Each of them was modelled as an actor for a given MoC.

5.1 Implementation

The Smart Room control system was implemented both as DE and as TT using the EPOSMote II, a sensor mote capable of communicating using IEEE 802.15.4 standard. The MoC period and the deadline of each actor were set to 240 ms. A deadline miss detector was used to check that all actors met their deadlines during the system execution. The system was divided into three subsystems, one for each variable being controlled.

The temperature subsystem is composed by three actors: a sensor, a controller and an actuator. The first is interacts with a temperature sensor and forwards the measured data to the controller. Based on the received samples of temperature and the desired temperature, the controller uses a heuristic to decide how the actuator operates. Finally, the actor that interfaces with the actuator interprets the commands sent by the controller and communicates with the temperature actuator.

The level of humidity inside the room can vary according to the number of people in the room and air flow from the outside. The humidity control subsystem was able to keep the humidity at a comfortable level inside the room. This subsystem was in charge of reading the humidity sensor and interfacing with the air dehumidifier.

The CO_2 control subsystem can keep the CO_2 comfort levels at a given rate. Its topology is similar to the temperature control subsystem; three actors comprising a sensor, a controller and an actuator. The actuator can insert fresh air coming from the outside to the room, lowering the CO_2 concentration inside.

5.2 Results

The results are based on data gathered during four hours of system functioning. Two people were using the room, and the outside temperature was 20° C. We have set the desired temperature in the room to 24° C with a tolerance of 0.5° C. The temperature measured inside the room is depicted in Figure 6. As seen, the measured temperature

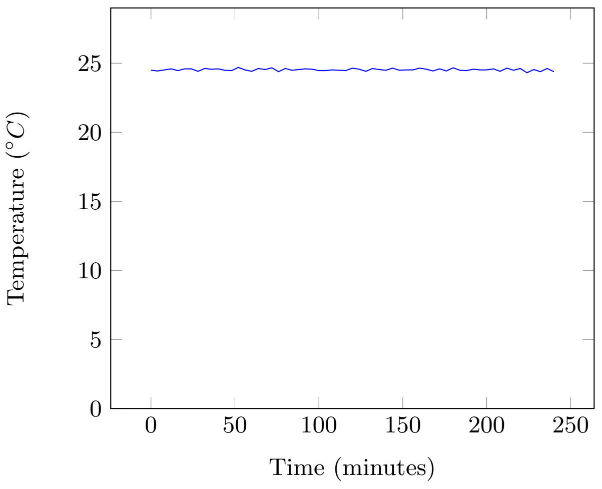


Figure 6: Smart Room temperature during a four hour period.

approaches the desired temperature during the period of operation. In order to save electrical energy, the system shuts down the temperature actuator after staying in the tolerance zone for a given amount of time. Due to this mechanism, the measured temperature oscillates near the desired temperature. Fine-tuning these parameters might reduce these oscillations.

6. CONCLUSIONS

Many MoCs are implemented ad-hoc along with the applications (e.g. executive cyclic that implement synchronous reactive MoC), or rely on tool-based solutions that are limited on generating functional blocks, or in proprietary infrastructures for specific MoC (e.g. TTTech TT). This paper presents an open-source run-time support system that supports the implementation of MoCs targeting CPS applications. Its current design supports DE and TT MoC and uses RTOSs abstractions to implement MoC concepts.

We have evaluated the RTSS's time overhead with different topologies and number of elements. For both experiments, we have employed ANOVA to detect whether there was some difference on the studied levels. Additionally, we have developed regression models that estimate time overhead, given the number of ports, actors, and interfaces used in the system design. The analysis shows a small overhead due to the infrastructure (in the order of hundreds of microseconds). The proposed RTSS was deployed on a case study comprising a Smart Room controlled by embedded devices, demonstrating the feasibility of the RTSS while employed to develop a real CPS embedded application.

7. REFERENCES

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

[2] 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.

[3] F. Maraninchi and T. Bouhadiba. 42: Programmable models of computation for a component-based approach to heterogeneous embedded systems. In *Proceedings of the 6th International Conference on Generative Programming and Component Engineering*, GPCE '07, pages 53–62, New York, NY, USA, 2007. ACM.

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

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

[6] 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.

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

[8] 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.