Simulating a Multicore Scheduler of Real-Time Control Systems in Simulink

Wei Li, Ramamurthy Mani, Pieter J Mosterman, and Teresa Hubscher-Younger
MathWorks
4 Apple Hill Drive, Natick MA 01760, USA
{Wei.Li, Ramamurthy.Mani, Pieter.Mosterman, Teresa.Hubscher-Younger}@mathworks.com

ABSTRACT
Today's real-time control systems are becoming highly distributed, concurrent, and complex. Performance of architectural components such as operating system schedulers and networks must be considered while validating controller design. A framework is presented based upon Simulink® and SimEvents® for simulating the effects of multicore operating system scheduling in the context of control system models. This framework leverages the discrete-event simulation engine underlying SimEvents and a new authoring framework based on MATLAB®.

INTRODUCTION

Control Engineering has moved through a number of distinct phases as it has evolved into an engineering discipline. In the first phase, building control functionality into a system was an artisanal matter. By clever design of a system, an objective behavior could be controlled. For example, the water clock (clepsydra) of the antiquity used a floater to maintain a constant flow of water from a vessel. In the second phase, separate control devices, such as the flyball (centrifugal) governor, introduced during the industrial revolution, were designed and integrated into the controlled system. In the third phase, post World War II, the feedback control architecture was established to analyze and systematically design control behavior that was then implemented by dedicated physical components. In the fourth stage that has started to emerge over the past decade, the control functionality is designed for a logic platform, that is, an abstraction of the underlying physics. Run-time execution scheduling separates the control functionality in logic from the physical components that realize the corresponding behaviors.

A key challenge that the separation of functionality and physics introduces is the characterization of control performance in the face of dynamic execution scheduling. For example, a software task may implement certain control functionality without static determination of the computing resources that ultimately execute the task. Specifically, on a multi-core processor, it may not be known a priori when a task is executed and on which core [1]. Similarly, in a broader scope, on a multi-processor system a runtime engine determines when tasks are to execute and on which resource [2].

While scheduling a resource such as a microcontroller so it can be shared has the advantage of increased utilization, control applications come with rigorous time constraints that must be honored to achieve desired control performance. To optimize the overall system design, a trade-off analysis must be made based on the implication that scheduling functionality on shared resources has on the ability to satisfy control performance objectives.

The work presented here enables a separate but integrated formulation of control functionality and of the scheduling method during the control system design. The strict separation enables quick design space exploration by experimentation with different scheduling methods (e.g., rate monotonic) and resource configuration (e.g., number of cores on a microprocessor). Additional shared resources can be modeled explicitly to enable, for example, modeling of access to peripherals.

The work is integrated in a larger tool suite for Model-Based Design that is based on Simulink® [3]. The integration allows immediate analysis of the simulated effects of a scheduler on control performance within an overall control system design, which enables direct assessment whether the performance conforms to system requirements. Moreover, if necessary, better control performance can be attained by modifying the control functionality or the system design in the original modeled representation.

The solution builds on a discrete-event modeling environment with a corresponding execution engine for dynamic-event scheduling. The dynamic even scheduling is integrated with static event scheduling of an execution engine for models that are based on periodic tasks [4]. In addition to the integration with the periodic control behavior, supervisory control is supported by a state transition modeling environment with its execution engine that is integrated as well. The work presented here illustrates the supported performance analyses of a control strategy subject to different implementation choices.

Previous solutions in this area based on MATLAB® and Simulink include TrueTime [5] and TRES [6]. Both provide functionality to model and simulate multi-task scheduling and networks in embedded control systems. A TrueTime Kernel block simulates a single-core computer node with a generic real-time kernel, A/D and D/A converters, external interrupt inputs, and network interfaces. A limitation of

TrueTime is that the model of a real-time control task must be written as either MATLAB or C++ code. This results in constrained access to Simulink controller design blocksets. In addition, for an existing Simulink model where controllers are defined as graphical blocks with execution and timing specifications, this approach requires substantial re-architecting and is often not practical.

TRES provides a modular framework that adds real-time task and network models to existing Simulink models. However, adding a task or a network message requires insertion of complex modeling constructs, and therefore is error prone while incurring possible scalability issues. In addition, TRES simulates event-driven behaviors by using Simulink variable-step sample time and zero-crossing technologies. Comparing this with a true discrete-event simulation engine, the TRES approach has a non-trivial performance overhead.

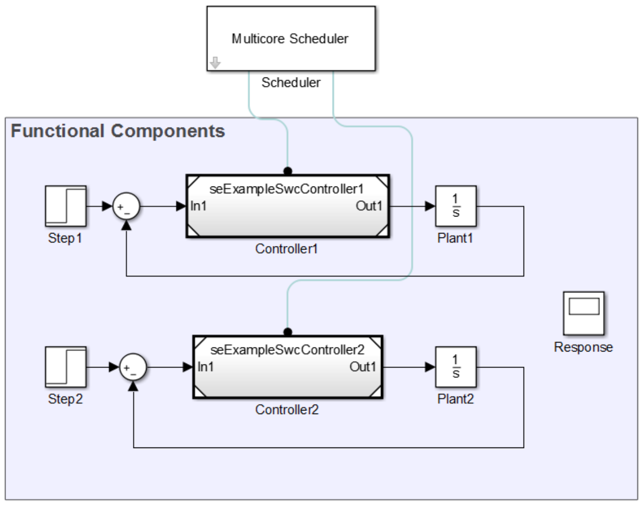


Figure 1. An example multicore control system with scheduler model

SIMUATING A MULTICORE SCHEDULER OF REAL-TIME CONTROL SYSTEMS

A framework in the Simulink environment is proposed for the graphical modeling of a multicore scheduler with resource contingencies. The framework is implemented by a Scheduler block that is provided as a SimEvents$^{®}$ model component with discrete event semantics [7, 8]. This framework enables the simulation of scheduling effects (e.g. latencies) and explore the task execution design space by analysis of the impact of scheduling choices on performance of a given control design. The framework is overlaid on the control algorithm and enables seamless migration to the algorithm implementation to deployed software via automatic code generation technologies. Figure 1 shows an example Simulink model using this framework consisting of a Scheduler block (*Scheduler*) and associated Controller algorithmic components (*Controller1* and *Controller2*). The Scheduler implicitly relies on the definition of the notion of a **Task** which is a real-time task with the following general properties:

- **ID**: Unique identifier of a task.
- **Period**: Time period between sequential task invocations when the task is instantiated and enabled for execution.
- **Priority**: Numeric value indicating relative importance of a task.
- **Segments (or subtasks)**: A set of individually schedulable executables of a task.

Each task segment is defined by:

- **Task function**: Implemented as a Simulink Function subsystem containing blocks implementing corresponding Controller computation.
- **Execution duration**: Worst-case execution time for a task segment to complete, if it is executed on a processor without interruption
- **Resource requirement**: Resources (other than processor) required to execute a task segment. A resource can be a peripheral piece of equipment such as a shared memory location or hard disk.

In the example of Figure 1, *Controller1* is a simple proportional control algorithm of the form shown in Figure 2. However, it is implemented within Figure 1 as three Task functions: *t1_read*, *t1_run*, and *t1_write*. This decomposition helps model the implicit sensor read and actuator write that would be present in an actual Controller implementation.

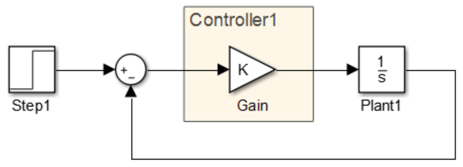


Figure 2. Controller model without task specification

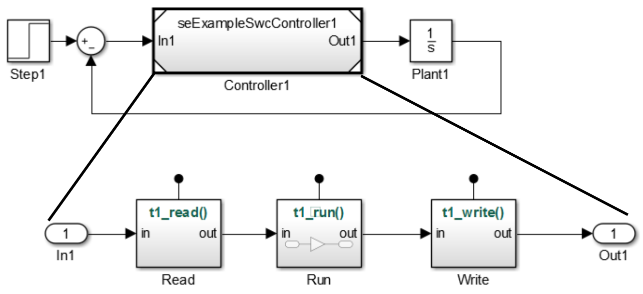
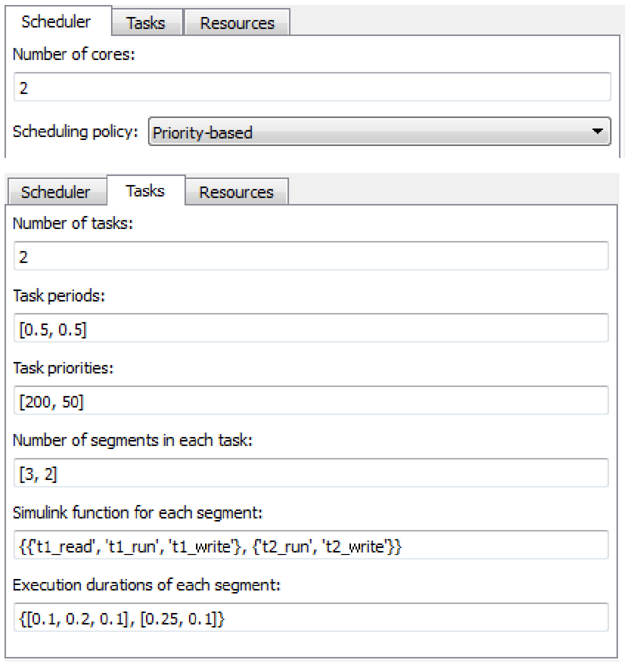


Figure 3. Simulink$^{®}$ Function defining a controller task with three segments

The scheduler itself models a homogeneous multicore system and is defined by the following properties that are presented as simple parameters of the block as shown in Figure 4.

- **Number of cores**: Number of cores available on the real-time control system.
- **Scheduling policy**: Rule and algorithm of scheduler in determining task execution sequences. Some standard policies are fixed-priority scheduling and round-robin scheduling and are pre-configured in the Scheduler block. Other policies can be added by expressing them in MATLAB.
- **Mutually exclusive resources**: A set of resources each identified by a numeric index, representing resources (other than processors) required to execute a task.

The SimEvents Scheduler block allows users to specify Tasks as block parameters also as shown in Figure 4.


Scheduler | Tasks | Resources

Number of cores:
2

Scheduling policy: Priority-based

Scheduler | Tasks | Resources

Number of tasks:
2

Task periods:
[0.5, 0.5]

Task priorities:
[200, 50]

Number of segments in each task:
[3, 2]

Simulink function for each segment:
{{'t1_read', 't1_run', 't1_write'}, {'t2_run', 't2_write'}}

Execution durations of each segment:
{[0.1, 0.2, 0.1], [0.25, 0.1]}

Figure 4. Example parameter configuration of SimEvents® Scheduler block

These parameters define a homogeneous multicore system with two cores. The scheduler is setup to perform a priority-based policy. The system has two control tasks, with:

- Task 1 for *Controller1* is configured with a period of 0.5 second, priority of 200 and 3 segments, whose functions and execution durations are (*t1_read*, *t1_run*, *t1_write*) and (0.1, 0.2, 0.1) second respectively.
- Task 2 for *Controller2* is configured with a period of 0.5 second, priority of 50, 2 segments, whose functions and execution durations are (*t2_run*, *t2_write*) and (0.25, 0.1) second respectively.

The implementation of the Scheduler block builds on a high-level discrete-event based language that is supported by Simulink. In turn, this language is MATLAB based and enables the implementation of additional scheduling functionality (e.g., different policies).

Note that the modeling framework is more generic and not strictly tied to the use of this Scheduler block. As an alternative to the Scheduler block the scheduling functionality may also be captured by a graphical model as shown in Figure 5. In this model, all the blocks that model the closed-loop control systems are the same. Only the Scheduler block is replaced by a SimEvents implementation that relies on discrete-event blocks. Therefore, the proposed framework makes the implementations of the scheduler and the controller model replaceable without affecting one another. This additional power for expressing the scheduler using textual, graphical, or even state-chart notation [7] allows for great modeling flexibility.

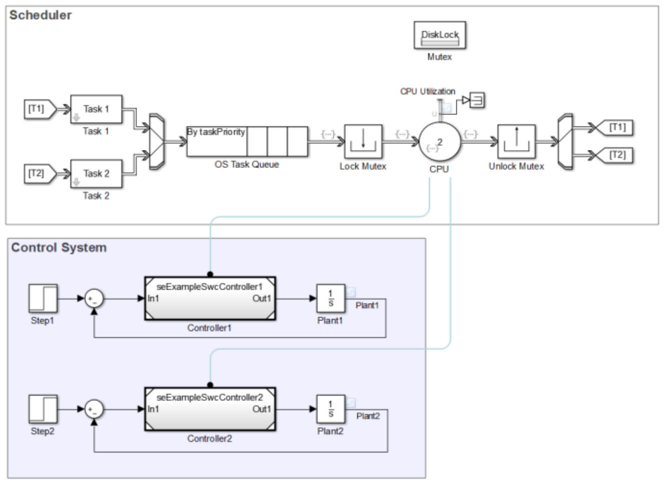


Figure 5. Scheduler implementation replaced without changing controller models

An important contribution of the framework is the strict separation of the algorithmic part from the computational implementation on a given architecture. The control algorithm that captures the functionality is seamlessly mapped onto the execution components that are part of the architecture of the run-time environment. For instance, one can simply split the three functions of *Controller1* into two tasks instead of one by simply manipulating the mapping shown in the dialog of Figure 4. This separation is a key feature of the framework that enables the modification and exchange of the control algorithm and of the scheduler independent from one another, which, in turn, supports design space exploration at different phases of the control system design within one control design environment.

HIGH-LEVEL DISCRETE-EVENT MODELING LANGUAGE IN MATLAB®

The Scheduler block of the previous section is created using the MATLAB Discrete-Event System block of SimEvents.

MATLAB Discrete-Event System is a high-level MATLAB construct for discrete-event system modeling. It abstracts a discrete-event system as an entity-flow system where discrete objects (entities) are stored, transferred, and processed. Main modeling elements include:

- **Entity**: An entity is a discrete object that contains persistent data (attributes) and exhibits certain run-time behavior. MATLAB Discrete-Event System provides syntax to specify entity types. Similar to concepts in object-oriented languages, an entity type defines a class of entities that share common data specifications and run-time methods.
- **Storage**: Entity storage is a MATLAB construct that can store multiple entities. Using MATLAB Discrete-Event System, you can define a discrete-event system containing multiple entity storage constructs. Each storage construct can be parameterized to have a certain capacity and use one of the commonly used queueing policies (e.g. First-In-First-Out or priority-based).
- **Event**: MATLAB Discrete-Event System provides syntax to create and schedule events. It also provides programming interfaces to define how the system responds when an event occurs. For example, timer events can be created and scheduled. When the timer is due, the interface that was implemented as timer event action will be invoked. The action implementation is provided with event context information (e.g. timer name, associated entity, simulation clock time) to perform desired updates to the system.

The language also supports modeling system interfaces, which enables composition by integrating multiple systems.

- **Port**: A MATLAB Discrete-Event System can have multiple input ports and output ports. A port can either be an entity port (allowing entities to enter/exit), or a data port (allowing input/output of values). This enables MATLAB Discrete-Event System blocks and other SimEvents blocks to be connected in a Simulink model.
- **Parameter**: A MATLAB Discrete-Event System can have multiple parameters. These parameters specify both the structure of the system (e.g. number of ports/entity storage), and run-time behaviors (e.g. task execution durations). Some parameters (called tunable parameters) can be changed, while the simulation is running.
- **State**: A MATLAB Discrete-Event System can have multiple state variables. A state variable can be either a simple data element such as a double-precision number or a MATLAB object such as another MATLAB Discrete-Event System. This enables the implementation of a "system of systems" where a component itself is a well-defined discrete-event system.

To model a discrete-event system using a MATLAB Discrete-Event System, a set of programming interfaces are to be implemented. Table 1 shows some interfaces that enable the definition of entity types, ports and storage.

Interface	Description
getEntityTypesImpl	Define entity types of discrete-event system
getEntityPortsImpl	Define input ports and output ports of discrete-event system
getEntityStorageImpl	Define entity storage elements of discrete-event system

Table 1. Programming interfaces for entity types, ports and storage specification

Table 2 shows some programming interfaces for defining event actions.

Interface	Description
entryImpl	Action when entity enters storage
exitImpl	Action before entity exits from storage
generateImpl	Action upon entity creation
destroyImpl	Action upon entity destruction
timerImpl	Action when timer completes

Table 2. Programming interfaces for event actions

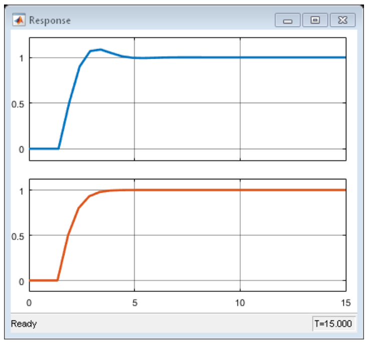
Scheduler block as a MATLAB® Discrete-Event System

The Scheduler block of the previous section is created as a MATLAB Discrete-Event System. It implements the multi-core scheduler as follows:

- A task instance is modeled as an entity that is periodically created, executed, and destroyed at the end of execution. Properties and runtime states of a task instance are stored as data of the entity.
- The task queue of the operating system and the multicore processor itself are modelled as two entity storages.
- Task instances (entities) are created periodically inside the task queue. They are stored in an ordered sequence as required by the scheduling policy.
- The scheduling algorithm is realized by scheduling events on the task entities. Events cause tasks (entities) that are ready to execute to be forwarded from the task queue to the processor (the 2nd entity storage).
- An executing task stays in the processor for the time period as specified by the *execution duration* parameter. Such execution duration is simulated by

using a *timer* event. When the timer completes, its event action executes the corresponding task function (a Simulink Function).

- A task instance is returned to the task queue if it has more segments to run, otherwise it completes and is destroyed.



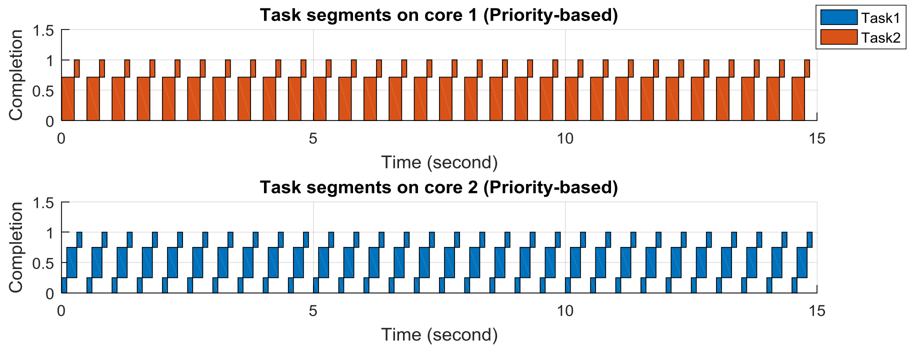


Figure 6. Task schedules and controller performance (two cores, priority-based scheduling)

SCHEDULER SIMULATION SENARIOS

A key advantage of the introduced framework is the ability to experiment with different schedulers without having to rework the control algorithm functionality. This is illustrated by applying a number of different tasking and scheduling profiles to the example of Figure 1.

Comparing Different Core Allocations

The Scheduler block allows assigning an arbitrary number of cores and exploring how that impacts system performance. In the first scenario two cores have been assigned to execute the two control tasks. Figure 4 illustrates parameters of the Scheduler block for this setup.

With sufficient processing capacity, both closed-loop control systems perform acceptably when the set point is changed from 0 to 1 (see Figure 6). Figure 6-10 each includes a response diagram (on the left-hand side) and a timing diagram (on the right-hand side).

- The response diagram shows the response of the two controllers with Controller1 in the top graph, and Controller2 in the bottom graph.
- The timing diagram shows utilization of cores and resources. A colored bar indicates execution of a segment of task. The position of a bar on the horizontal axis indicates time of start and completion of the segment. The position of the bar on the vertical axis indicates normalized task completion before and after executing the segment.

The timing diagram of Figure 6 indicates that tasks are processed concurrently with cores having medium and balanced utilizations. Notice that Task2 is assigned to Core1 because it has higher priority and, therefore, is assigned before Task1.

In comparison, when only one core is available, the performance of Controller1 (mapped to a low-priority task) degrades because of task overruns (see Figure 7). For example, the first instance of Task1 does not complete until time 1.45s because its execution was preempted by two instances of the high-priority task (Task2). Such overrun causes only 3 instances of Task1 to complete execution between time 0 and 5, while 10 instances have been scheduled.

Notice that the performance of control task 2 remains unchanged. This is because the scheduler applies a priority-based policy where processing capacity is maximally assigned to high priority tasks.

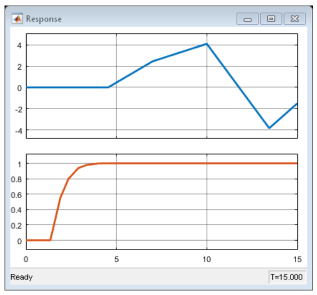
Comparing Different Scheduling Policies

At this point, if the Scheduler switches to use a round robin scheduling policy, the control system performs differently (see Figure 8). Compared to the previous case where processing capacity remains the same, the control of Plant 1 becomes stable, with the cost of degrading the performance of the control of Plant 2. This change is because the round-robin policy evenly assigns the processing capacity among all tasks.

Comparing Different Resource Allocations

At this point, the design study was returned to the scheduler configuration of Figure 4, and different resource allocation schemes explored. A resource is added that must be shared by tasks in a mutually exclusive fashion such as a file handle (see Figure 9). As indicated by the timing diagram, although concurrent execution is allowed with two cores, the tasks are processed in a sequential fashion. Only one core is in use. This is because a task must wait for the required resource before it can start to run.

Such resource contingency can be eliminated by assigning more resources. Thus, the scheduler is configured to use 2 resources and each task is allowed to have a dedicated resource (see Figure 10). With each task having its own resource, tasks are now processed concurrently.



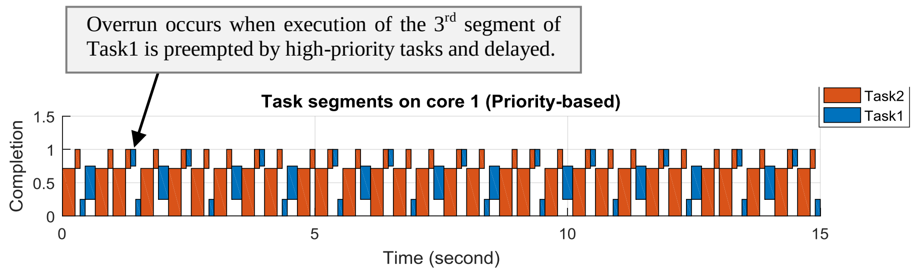
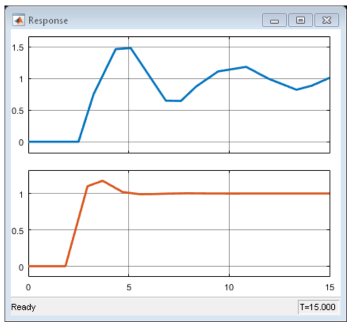


Figure 7. Task schedules and controller performance (one core, priority-based scheduling)



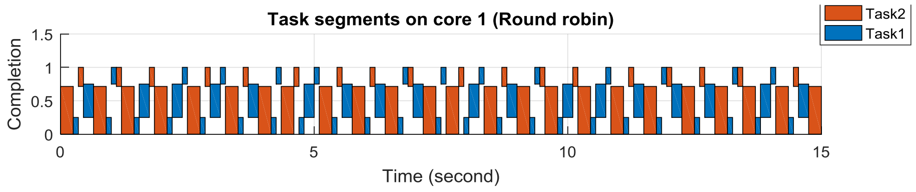
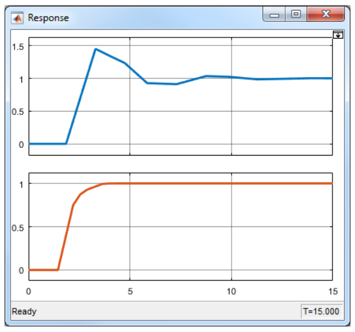


Figure 8. Task schedules and controller performance (one core, round-robin scheduling)



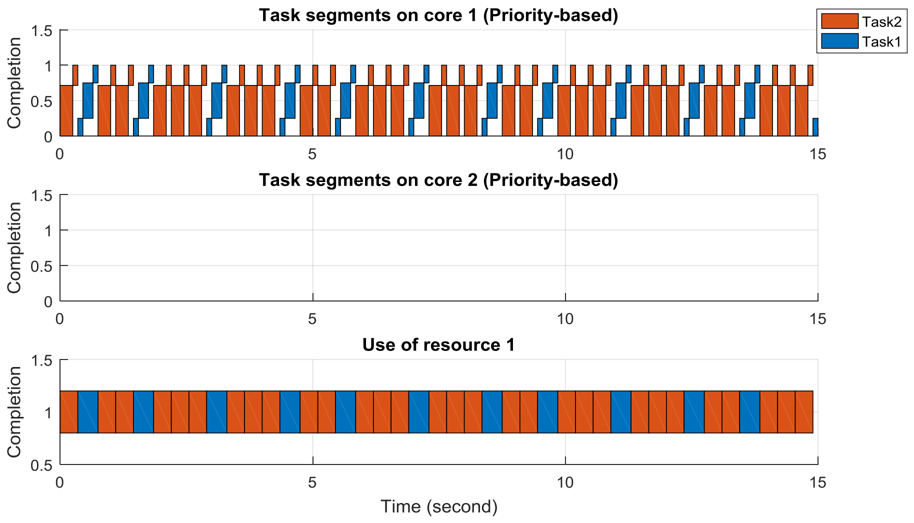
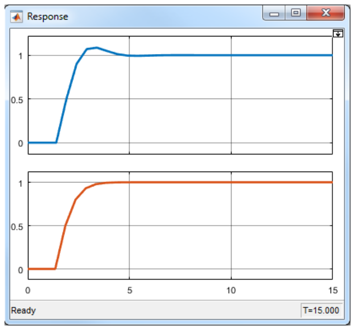


Figure 9. Task schedules and controller performance (two cores, tasks share resource, priority-based scheduling)



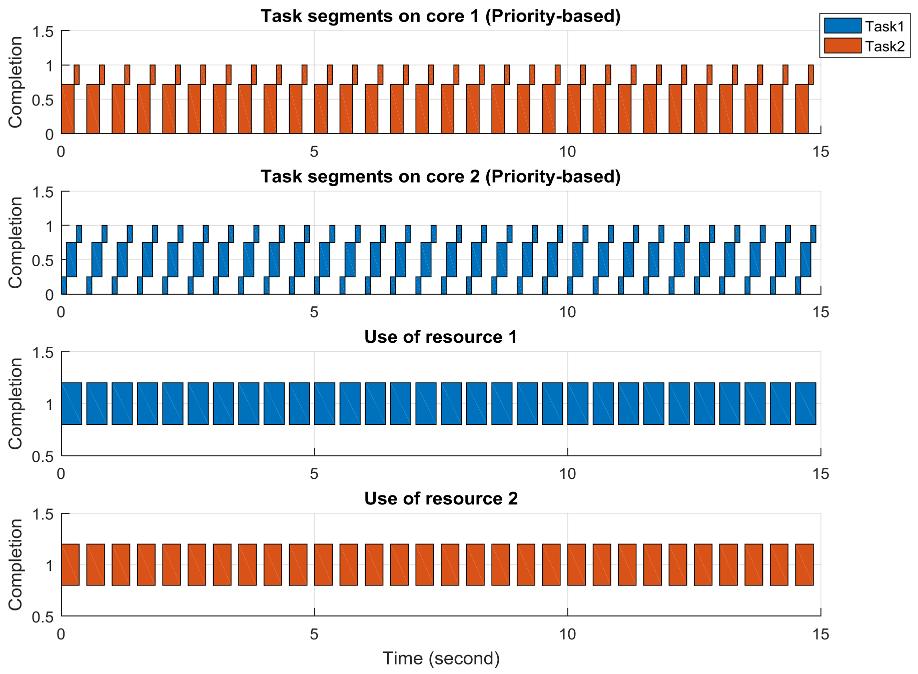


Figure 10. Task schedules and controller performance (two cores, no resource sharing, priority-based scheduling)

CONCLUSION

This paper describes some new advances in Simulink for modeling architectural components of real-time multicore control applications, including:

- A new Scheduler block for modeling and simulation of scheduling effects (e.g. latency, overruns, resource contingencies) of real-time, multicore control applications.
- An integrated and extensible framework in Simulink for co-simulation of functional components (controllers and plants) and architectural components (scheduler and networks) of a distributed control system.
- A high-level language in MATLAB for discrete-event system modeling and simulation. Syntax and semantics of this language are well suited for modeling dynamic system architectures.

High performance discrete-event simulation of the above features is supported by an optimized discrete-event simulation engine [9].

REFERENCES

1. Astrom, K. J. and Wittenmark, B. *Computer-controlled systems: theory and design*. Courier Dover Publications, 2011.
2. Barbalace, Antonio, et al. Performance comparison of VxWorks, Linux, RTAI and Xenomai in a hard real-time application. In *Proceedings of Real-Time Conference, 2007 15th IEEE-NPSS*. IEEE (2007).
3. Simulink MathWorks, *Simulink® User's Guide*, MathWorks®, Natick, MA, March, 2016.
4. Mosterman, Pieter J. Hybrid Dynamic Systems: Modeling and Execution. In *Handbook of Dynamic System Modeling*. Fishwick, P.A. (editor), Chapter 15, CRC Press (2007),15–23.
5. Cervin, A. and Arzen K.E. TrueTime: Simulation tool for performance analysis of real-time embedded systems. In *Model-Based Design for Embedded Systems*. Nicolescu, G. and Mosterman, P.J. (editor), Chapter 6, CRC Press (2010), 145-178.
6. Cremona, F., Morelli, M., and Natale, M. Di. TRES: A Modular Representation of Schedulers, Tasks, and Messages to Control Simulations in Simulink. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing, SAC '15*, ACM Press (2015), 1940–1947.
7. MathWorks, *SimEvents® User's Guide*, MathWorks®, Natick, MA, March, 2016.
8. Develop Custom Scheduler of a Multicore Control System. http//www.mathworks.com/help/simevents/examples/develop-custom-scheduler-of-a-multicore-control-system.html.
9. Clune, Michael I., Mosterman, Pieter J., and Cassandras, Christos G. Discrete Event and Hybrid System Simulation with SimEvents. In *Proceedings of the 8th International Workshop on Discrete Event Systems*, 2006.