TRES: A Modular Representation of Schedulers, Tasks, and Messages to Control Simulations in Simulink

Fabio Cremona, Matteo Morelli and Marco Di Natale
Institute of Communication, Information and Perception (TeCiP)
Scuola Superiore Sant'Anna
56124 Pisa, Italy
{fabio.cremona, matteo.morelli, marco.dinatale}@sssup.it

ABSTRACT

Model-based development of CPS is based on the capability of early verification of system properties on a model of the controls and the controlled physical system, and the capability of producing automatically an implementation of the model. Unfortunately, in the development of complex distributed or highly concurrent systems, the scheduling and communication delays may significantly affect the behavior of the controls. We present a framework for adding the model of schedulers, tasks and messages to Simulink models and to verify by simulation the impact of scheduling and execution times delays on the performance of the controls. Our toolset is highly modular and extensible and allows application to existing models with limited changes and even the automatic synthesis of the task and message model from an external specification.

1. INTRODUCTION

Simulink is a graphical modeling environment implementing a Synchronous-Reactive (SR) Model of Computation (MoC) for multidomain Model-Based Design (MBD) and simulation. Continuous-time, discrete-time and discrete-events systems can be defined in the model, making Simulink suitable for the design of Cyber-Physical Systems (CPSs), where the controller (discrete-time) and the plant (continuous-time) must be modeled and simulated together.

CPSs are often distributed and execute on nodes connected by a wired or wireless networks. The Simulink model of the controls is purely functional and allows the modeling and simulation of the control function executing in logical time, with the assumption that all reactions complete within the next event. However, in a real system implementation, control functionality executes in finite time, and messages are queued waiting for transmission on network(s), giving rise to latencies and jitter that may change the performance of the controls.

To represent computation and communication delays in Simulink, a possible solution is provided by the TrueTime framework from the University of Lund [6]. TrueTime integrates a real-time scheduling and a network simulator in a Simulink custom block, enabling the co-simulation of the control functions considering the software (task) and message implementation details and the scheduler and resource management policies. TrueTime supports the Earliest Deadline First (EDF) and Fixed Priority (including Rate Monotonic, RM) scheduling policies [5] for single-core systems and several network standards, including Ethernet, the CAN bus and FlexRay, but also wireless networks such as 802.11b WLAN and 802.15.4 ZigBee. Presently, multi-core architectures are not supported. TrueTime is implemented as a single custom Simulink block specified as a Matlab or C++ S-Function and implements, in a monolithic fashion, the controller functionality, the task model, and the scheduler.

In this paper, we present a modular framework for the co-simulation of control functionality and controlled system dynamics with real-time scheduling policies, communication mechanisms and real-time controllers. Similar to TrueTime, it is based on Simulink, allowing the co-simulation of any discrete-time controller model and continue time plant. However, our framework provides a modular and extensible architecture and allows for three key features that can be considered its main contributions.

- the addition of a task implementation model and a scheduler to an existing Simulink model with limited and localized changes. The proposed modeling structure clearly separates the controller model (functionality) from the model of the task, the scheduler, the communication mechanisms and the other attributes of the execution platform.
- the modular integration of a (possibly third-party) real-time scheduling simulator and network simulator, through simple and generic interfaces. For completeness and self-consistency, we provide an adapter to an existing open source scheduling and resource management simulation project RTSim [12], and the open source network simulator framework OMNeT++ [21].
- the possibility of the automatic generation of the task, scheduler, network and message blockset from a (SysML) specification using model-to-text transformation rules to generate the blockset using a Matlab script.

Our framework is freely available as open source from `http://retis.sssup.it/tres`. The paper is organized as follows. In section 2, we introduce the model assumptions and provide a short summary of the Simulink semantics, its execution model and the interactions between the simulation

SAC'15 April 13-17, 2015, Salamanca, Spain.

http://dx.doi.org/xx.xxxx/xxxxxxx.xxxxxxx

engine and our framework. In section 3, we describe the structure of the framework, with the custom blockset and the internal structure of the adapter towards the scheduling simulator. In section 4 we provide an example that shows a use case in which different task models and scheduling policies impact the performance of the controls, as shown by the simulation results. Finally, in section 5, we provide our conclusions and a discussion of future work.

1.1 Related Work

A large variety of simulation tools are available from the industry and the academia. Most of them are geared towards the use in a specific domain. NS-3 (Network Simulator) [16] and OMNeT++ [17] are freely available discrete-event computer network simulators. They supports several multimedia communication protocols and are extensible for the inclusion of new ones. In [15] a simulation environment for CAN-Ethernet networks is presented as example of extension to OMNeT++. Simulation of CAN and FlexRay automotive network communications protocols is available from many commercial tools, including those from Vector [22].

A very large number of projects target the evaluation of scheduling policies and the analysis of task implementations (more than 6 million hits when searching the keywords *real time scheduling simulator* in Google). A necessarily incomplete list includes Yartiss [8], Storm [20], ARTISST [9], Cheddar [19], Schesim [14], Stress [4].

TrueTime [6] is a *freeware*[1] Matlab/Simulink-based simulation tool that has been developed at Lund University since 1999. It provides models of multi-tasking real-time kernels and networks that can be used in simulation models for networked embedded control systems. TrueTime is used by many research groups worldwide to study the (simulated) impact of lateness and deadline misses on controls.

Several research works investigate the consequences of computation (scheduling) and communication delays on controls. An overview on the subject can be found in [3]. Recent works on this subject include [7]. Also, an analysis of control activation models based on events rather than periodic triggers (and the possible improvements with respect to the CPU resource) are discussed in [2] and [13].

The TrueTime Kernel block simulates a *computer node* with a generic real-time kernel, A/D and D/A converters, external interrupt inputs and network interfaces. The block is configured via an initialization script (usually written in Matlab code), where a specific API is used by the designer to create tasks, timers and interrupt handlers and define the scheduling policy and the communication resources.

In TrueTime, the model of task code is represented by *code functions* that are written in either Matlab or C++ code. A TrueTime developer has two options: hand-code the control logics and lose availability of Simulink (control) toolboxes, or call external discrete-time Simulink models from within the code functions using a mechanism based on the MATLAB built-in operator `sim()` with several limitations: signal-generator blocks that use the simulation time and blocks for which is not possible to specify the sample rate (e.g., the Discrete Derivative block) cannot be used. In addition, data connections among Simulink models need to be implemented in code using a purposely offered API and the application of a TrueTime Scheduler to an already existing Simulink model of controls requires substantially rewriting, mixing the controller functionality, and models of the task set, the scheduler, and the physical execution platform. Because of the monolithic architecture and the number of code artifacts that are needed for system configuration (e.g., initialization script and code functions), the current TrueTime implementation is hardly compatible with an automatic model generation and a M2M transformation flow.

SimEvents [18] is a commercial toolbox developed by The MathWorks, providing a discrete-event simulation engine and component library for Simulink. It enables event-driven communication modeling between Simulink components to analyze and optimize end-to-end latencies, throughput, packet loss, and other performance characteristics. The component library allows the designer to customize processing delays, prioritization, and other operations to represent systems that range from manufacturing processes, to hardware architectures and sensor/communication networks. Because of its generality, SimEvents has no explicit notion of task, real-time scheduler, network protocol or hardware component. They need to be built as libraries using the elementary queue and server blocks.

2. BASIC CONCEPTS AND ASSUMPTIONS

Simulink implements a Synchronous-Reactive (SR) model of computation integrating continuous-, discrete-time and discrete events (triggered) subsystems and blocks. Continuous-type blocks process continuous-time signals and produce as output other continuous signal functions according to the block description, typically a set of differential equations. The equations that compute the dynamics of the continuous part are integrated by a solver at fixed or variable time steps. A variable step solver is invoked at those points in time that are relevant for the dynamics of the system they compute (major steps and zero-crossing points).

Discrete-Time Simulink blocks are activated at periodic-time instants and process input signals, sampled at periodic instants, producing a set of periodic-output signals and the state updates. Finally, triggered blocks are only executed on the occurrence of a given event (a signal transition or a *function call*).

A fundamental part of the model executable semantics is the rules dictating the evaluation order of the blocks. Any block for which the output is directly dependent on its input (i.e., any block with *direct feedthrough*) cannot execute and produce outputs until the blocks driving its inputs execute. The set of topological dependencies implied by direct feedthrough blocks defines a partial order. When the simulation starts, blocks are ordered and a total order of execution compatible with the partial order of execution is determined. When a block is triggered (activated), inputs are sampled and output update and state update functions computed in sequence to produce the system outputs.

The Simulink engine computes the states and outputs of the system at time points (steps) from the simulation start time to the finish time. The length of time between steps is called *step size*. *Variable step* solvers divide the simulation timespan in *major* and *minor* time steps. The solver produces a result at each major time step. Any point in time that is relevant for the dynamic of the controlled or the controller system corresponds to a major step. For example, all the triggering instants of discrete-time (controller) subsystems correspond to major steps. A minor time step is a subdivision of the major time step used to improve the ac-

[1] `http://www3.control.lth.se/truetime/LICENSE.txt`

curacy in the computation of the system dynamics and find the point in time when continuous-time system have a *zero-crossing* point, that is a point when some of the state variables cross a zero threshold (a significant change of state).

Simulink System Functions (S-Functions) are a mechanism for extending the set of predefined Simulink blocks. An S-function is a description of a Simulink block written in Matlab, C, C++ or Fortran. Interactions between the Simulink simulation engine and custom blocks occurs through a predefined set of API functions.

Figure 1 shows the simulation cycle at runtime with the major and minor steps and the points in the cycle in which the simulation engine invokes the API functions specified for the S-function blocks. Among those, the `mdlOutputs` is used to update the outputs of the custom block, the `mdlUpdate` to update the internal state of the custom block and `mdlZeroCrossing` to define the signals that determine the zero-crossing points and possibly force them (set a time instant for a future major step). The same task can be achieved by using the `mdlGetTimeOfNextVarHit` that allows to define a future time instant for a major step.

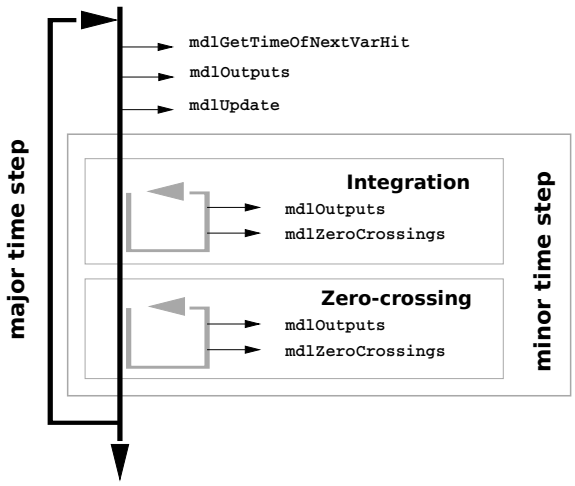


Figure 1: Simulation cycle in Simulink.

Real-Time scheduler Simulator.

A Real-Time scheduler simulator is a Discrete Event System (DES) implementing an event handling mechanism (typically with a queue). It reacts to tasks arrival events and dispatches the currently active tasks from the ready queue according to a fixed or dynamic priority-based scheduling algorithm. Tasks arrival events can arrive asynchronously or periodically and are ordered in the event queue in ascending order following (*i*) the event occurrence time (*ii*) a causality order for those with equal occurrence time. To preserve causality, a task is dispatched only when all the events at the current time have been processed. At any point in time, the next scheduling event can be the termination of the task currently in execution, or the arrival event of a task, that can possibly cause a preemption (if the new task has higher priority) and a context switch.

In an RT simulator, tasks execute according to a model of (time-consuming) computations. Our framework assumes the same model as in TrueTime (which is also suited to the typical code generation process for Simulink models). The execution of a task is split in preemptable units called *segments*, informally corresponding to the execution of a function called by the task main code. Each segment is identified by an execution time (possibly randomly generated according to a given distribution) and all segments in a task are executed in a sequence.

When the RT simulator is integrated with Simulink, segments map one-to-one to subsystems and the execution order of the segments in a task must match the order of execution imposed by the model semantics. The time duration of each task segment, corresponds to the execution time of the corresponding code function implementing the sub-system (and possibly generated from it in an automatic code generation flow). Formally, the execution model is the following (a strict subset of the Simulink semantics).

- $\mathcal{V} = \{S_1, ..., S_{|\mathcal{V}|}\}$, the set of *functional sub-systems* in the simulink model. Subsystem blocks can be continuous time (representing the controlled system or plant) or discrete time controller blocks, describing the controller logic. A functional controller subsystem S_i is characterized by a worst case execution time γ_i for its generated execution code on a given platform. We assume a subsystem reads or samples all its inputs when it starts executing, and generates the outputs when it completes execution. A subsystem S_i may have input and output ports. We assume the designer partitions the controller (discrete-time) model into single rate subsystems and that all input ports carry signals with a uniform sampling period t_i. The result of the block computation is a set of signals with the same rate, produced on the output port. The sampling period t_i of the input signal is also the activation period of the block.
- $\mathcal{E} = \{l_1, ..., l_{|\mathcal{E}|}\}$ is the set of *links*. A link $l_i = (S_h, S_k)$ connects an output port of subsystem S_h (source) to an input port of S_k (sink).
- A precedence relation may exist between a pair of subsystems S_i and S_j. The notation used is $S_i \prec S_j$.

On the side of the task model, supported by the RT simulator, we have

- $\mathcal{T} = \{\tau_1, ..., \tau_{|\mathcal{T}|}\}$ is the set of tasks. Each task τ_i has an activation period T_i or activation event e_i and an optional priority π_i (or other scheduling attributes). Periodic tasks have zero offset, thus they start at the same time instant $t = 0$.
- A mapping relation $mt(S_i, \tau_j, k)$ is defined between a controller subsystem S_i and a task τ_j meaning that the code implementing S_i is executed in the context of task τ_j with order index k. A mapping relation is only possible if the execution rate of S_i and τ_j are the same (the constraint could be relaxed allowing for integer divisors). If two subsystems are mapped onto the same task, the mapping order index must match their partial order of execution. If they are on different tasks, the execution order must be guaranteed by the priorities assigned to the tasks.
- tasks are scheduled for execution on a single- or multi-core platform according to a defined scheduling policy.

The major steps of the Simulink simulation must include all the periodic activation times of tasks as well as the aperiodic events that lead to the activation of other tasks. Every

time a major step occurs, the block implementing the real-time scheduling simulator is invoked and processes (if there is any) the task arrival event and any other event that is active at the same time. The task activation instant corresponds to the activation of the first task segment. Next, the real-time scheduler determines the tasks to be set in execution and the execution time for their current segments, determining the point in time when the current segments are expected to complete. These times are set as future major step times in the Simulink simulation using the zero crossing or next var hit API calls.

The actual start and completion times of the task segments must correspond to the times in which the corresponding subsystems reads or sample their inputs and produce their outputs. To guarantee this execution semantics, the activation of each subsystem must be changed from periodic to function activated (Figure 2), and a latch barrier must be added on all its outputs. The signals activating the subsystem (and its input sampling) and the output latch are generated by the task blocks upon the beginning of the execution and the completion of the corresponding task segment. Zero-execution time assumption is replaced by the finite-execution time assumption, enabling the simulation of the scheduling mechanisms upon the Simulink model.

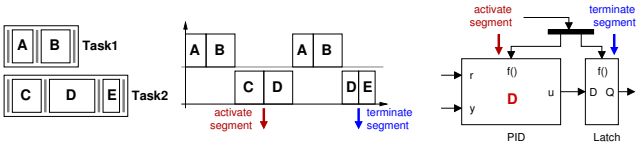


Figure 2: The execution of subsystems modeled through segments.

Figure 2 shows the activation mechanism of a block (D in the example). When a task segment starts executing, the corresponding block is activated. However, the output value is not instantaneously available for the other blocks in the system. The segment execution can be interrupted an its execution time can take longer than the original segment lenght since the task can be descheduled to execute higher priority tasks. To handle this, the output signal u is latched and enabled as output only when the segment terminates the execution. The segment completion results in the generation of the terminate signal that produces the signal update at the output of the latch.

Network Simulator.

Like a Real-Time scheduler Simulator, a Network Simulator is a DES simulating nodes exchanging messages over a network infrastructure with a given communication protocol. The simulator defines the timed events related to the transmission and arrival of messages by the networked nodes. The communication (MAC) protocol is the core attribute of the communication network. It defines the set of rules according to which messages are selected for transmission on shared physical links and ultimately determines the latency of messages together with the attributes that define the network speed and reliability. It is therefore important that a Network Simulator supports a large set of protocols and can be easily extended to include new protocols.

Formally, a network can be described with:

- $\mathcal{M} = \{m_1, \cdots, m_{|\mathcal{M}|}\}$, a set of messages.
- $\mathcal{N} = \{n_1, \cdots, n_{|\mathcal{N}|}\}$, a set of networks.
- A mapping relation $mm(m_i, n_j)$ is defined between messages and networks, meaning that message m_i is transmitted over the network n_j.
- A mapping relation $ml(l_i, m_j)$ is defined between links and messages, meaning that the data of the signal exchanged over l_i are mapped onto message m_j. Each link l_i can be mapped onto at most one message. many-to-one mappings are allowed (signal multiplexing). If l_i is not mapped to any message, then its implementation consists of local communication (typically a shared variable).

The synchronization between the Network Simulator and Simulink, is orchestrated by our Simulink Network custom block.

The major steps of the Simulink simulation must include all the periodic activation times of messages, as well as the aperiodic events that lead to aperiodic transmissions of messages. Every major step, the Network block is invoked and processes the possible message transmission and arrival events and determines when the currently sent messages are expected to be received. These time instants are set as future major steps in the Simulink simulation.

When a signal is sent by a message over a network, the signal is sampled and the message is assembled and put into a ready queue of messages ready to be sent. When it reaches the receiver, the message is disassembled and its signal values are available to be read. The time required to deliver the message depend on the underlying protocol and the network traffic. To guarantee this execution semantics, each networked link is replaced with a double latch barrier as highlighted in bottom of Figure 3. The Zero-communication delay assumption is replaced by the finite-communication delay enabling the simulation of the network exchange mechanism over the network.

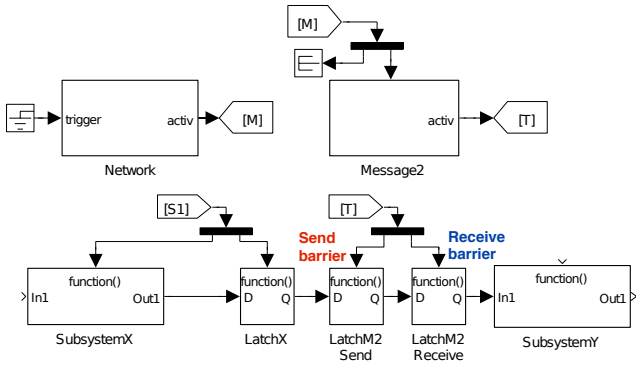


Figure 3: The transmisison of information signals packed into messages under the control of the network block.

When a message is sent, the send barrier is activated and the corresponding link sampled. The signal is not instantaneously propagated to the receiver block but gated by a second latch that blocks the signal until the message is actually delivered to the receiver node. When the message is delivered, the arrival barrier is activated, and the signal is made available to the other blocks in the system.

3. ARCHITECTURE

3.1 Execution Model

Our framework adds the capability of simulating real-time task execution of Simulink models on single- and multi-core platforms through the two custom blocks `Kernel` and `Task` and the capability of simulating the communication delay on Simulink signals (links) when they are implemented as messages exchanged over a network through the two custom blocks `Network` and `Message`, implemented as C++ S-Functions. The block `Kernel` models an event-based real-time kernel and the scheduler inside it, simulating the execution of tasks and interrupt handlers on a single- or multi-core computer node according to a given scheduling policy. Each task (or handler) executed by the `Kernel` is modeled with one instance of the block `Task`. Task execution consists of the serialized execution of the functions implementing the subsystems. These functions are segments associated with an execution time. The block `Network` models an event-driven network in which messages are exchanged between nodes (Simulink subsystems) according to a network protocol. The messages exchanged over the network are modeled with the block `Message`.

3.1.1 Real-Time Scheduler

Figure 4 shows the structure of the blocks `Kernel` and `Task` and the interactions among them. Figure 2 and the example in Figure 6 show how a block `Task` controls the activation and termination events of the subsystems that are executed by it.

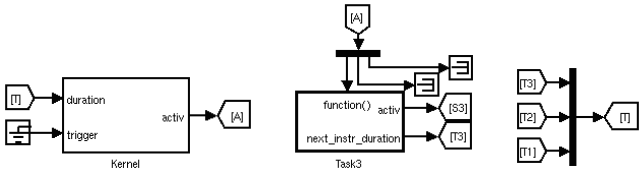


Figure 4: `Kernel` and `Task` blocks.

Each block `Task` is a triggered subsystem, executed on the occurrence of a function call event received on its port `function()`. Its output interface consists of two ports: `activ` and `next_instr_duration`. The first one is an array of function call events with size equal to twice the number of subsystems managed by the task. This port is used to issues activation and termination events to each Simulink subsystem implementing one of the segments. The second port outputs a scalar signal representing the duration of the next segment executed by the task. Each time `Task` is triggered, it (*i*) issues the termination signal for the previously executed segment (if any), (*ii*) outputs the activation signal for the current segment, (*iii*) updates its internal data structure to point to the next segment and (*iv*) transmits the execution time of the new segment to the block `Kernel`. Whenever there is no other segment to be executed, `Task` sends a *task completion* signal on the port `next_instr_duration`. At the next activation, it issues the termination signal for the last segment, sets its internal pointer to the current segment to the first segment and sends back to the `Kernel` block the segment duration.

The duration of segments executed by `Task` is set through the block mask dialog. The dialog box has one field for specifying the name of a Matlab workspace variable. The variable is a cell array of strings (pseudo-instructions) that conform to a specific syntax. Each string describes the computation time (i.e., the duration) of a segment, which can be fixed or random. Supported probability distributions that describe random computation times include the uniform distribution between a minimum and a maximum value, the exponential and the Dirac delta distributions (an example is shown in the following code section).

```
task_code = {'fixed(7)'; 'delay(unif(3,8))'};
```

The block `Kernel` has two input ports: `duration` and `trigger`. On the `duration` port receives an array of values, one for each `Task` block, with the indication of the duration of the next segment to be executed. On the second port, it receives the array of activations signals of aperiodic tasks (from external sources). The block has one output port, named `activ`, which is used to signal to each task the execution of the current segment.

The block `Kernel` uses a zero-crossing function to require future activations (from the Simulink engine) in correspondence of scheduled events but it is also activated synchronously with the arrival of aperiodic tasks.

The block `Kernel` is responsible for keeping the scheduling simulation aligned with the system simulation. At each activation, it checks for any aperiodic requests. If there is any, it activates the corresponding aperiodic tasks in the RT scheduler simulator. Next, it advances the RT scheduler simulator by looking for the next event in the simulator's event queue. Two types of events are relevant for the simulation: the *segment completion* and *task completion*. In case an event of the first type occurs, `Kernel` reads the input signal on the port `duration` at the index corresponding to the task that completed the segment, and dynamically creates a new instruction and insert it in the corresponding task. Finally, once it detects which task has generated the event, it sends an activation signal to the `activ` port to trigger the corresponding `Task`. If the event *task completion* is detected, `Kernel` simply reset the internal state of the corresponding task clearing the past history of the executed segments.

A number of parameters configure the (simulated) kernel and are set through the `Kernel` mask dialog. Three scheduling policies, namely Deadline Monotonic (DM), Fixed-Priority (FP) and Earliest Deadline First (EDF), are readily selectable from a context menu. Optionally, the designer can provide the type and the configuration parameters (if any) of other scheduling policies, provided that they are supported by the underlying RT scheduling engine. The designer can specify the number of cores on which the task execution is simulated. The current implementation of `Kernel` with RT-Sim, supports multi-core architectures with global scheduling policies. The mask dialog has one edit field for specifying the name of the Matlab variable (cell array) that describes the type and the timing properties of the (heterogeneous) task set (an example in the following code section).

```
task_set = {'PeriodicTask', 0.004, 0.004,  0;
  ...
           'PeriodicTask', 0.006, 0.006,  0};
```

Tasks can be periodic or aperiodic and timing properties include interarrival time, relative deadline and initial offset. Optionally, task priorities (for tasks scheduled according to FP) and core affinities can be specified for each task. Finally, in case several external RT simulators are supported, a context menu enables the designer to select the one that is used to perform the simulation.

3.1.2 Network

Figure 3 (top) shows the structure of the `Network` and `Message` blocks. `Message` blocks are driven by `Network` block with a dataflow port `state`. Block `Network` communicate to each `Message` the actual state of the simulated message over the network. The state can assume two possible values: *send* or *receive*. The block `Message` uses the function call output port `activ` to trigger the function triggered barriers shown in Figure 3 (bottom) depending on the *send* or *receive* state received from the `Network` on the `state` port.

When a message must be sent over the network, the `Network` block sends the *send* state to the appropriate `Message` block. Then, the `Message` block triggers the send barrier as shown in Figure 3. Once the message is delivered, the `Network` block will change the state signal (related to the delivered message) to *receive* and the `Message` block will trigger the arrival barrier as shown in Figure 3.

3.2 The RT Simulator API Layer

The blocks `Kernel` and `Network` rely on external engines to perform the processor scheduling and Network simulation. Their implementations are not tied to a specific tool or framework, and are designed according to the basic principles of object-oriented programming to provide an easy integration with external scheduling and network simulators.

The design of the RT Simulator API layer is based on the observation that every RT simulation framework basically consists of two high level components: an event handling system and, on top of it, the actual scheduling simulator. The first one defines a set of objects that provide a representation of events and event queues, and provides an API that enables the creation and deletion of events and their insertion in and extraction from the event queue. The second one uses the definitions of event (typically after specialization) and event queue and realizes the actual real-time scheduling simulation functionality. It implements the concepts of task (e.g., periodic, aperiodic), scheduling policy (e.g, DM, FP, EDF), and kernel (e.g., single- or multi-core, with a scheduler and a resource manager). The RT Simulator API layer abstracts these concepts and enables the development of the `Kernel` S-Function so that it depends upon a set of interfaces classes, rather than upon their concrete implementations (i.e., a specific RT simulation framework).

The RT Simulator API layer defines three virtual classes that are used by the `Kernel` S-Function, namely `Kernel`, `SimTask` and `Event`. These define a narrow interface for the kernel, task and event data structures offered by the external RT simulator. The *object adapter pattern* [11] is used to bind the abstract interfaces, used by the S-Function (client), with the third-party software represented by the RT simulator. Each class defines a set of operations as pure virtual functions.

The following is the set of virtual methods to be specialized by the refinement of the `Kernel` class realizing the adaptation to the real-time scheduling simulator. Another set of virtual methods is defined for the adapter classes for tasks and events (not shown because of space constraints).

```
// List of virtual methods to be specialized
// for the adaptation to the Real-Time
// scheduling simulator
virtual void initializeSimulation(const double,
   const double* const*) = 0;
virtual void processNextEvent() = 0;
virtual yaks::Event* getNextEvent() = 0;
virtual int getTimeOfNextEvent() = 0;
virtual int getNextWakeUpTime() = 0;
virtual void getRunningTasks() = 0;
virtual void activAperiodicTask(const int) = 0;
```

Their implementation is demanded to the object adapter, which calls adaptee operations to actually carry out the requests. Figure 5 shows the interaction of the custom `Kernel` block with the simulation loop.

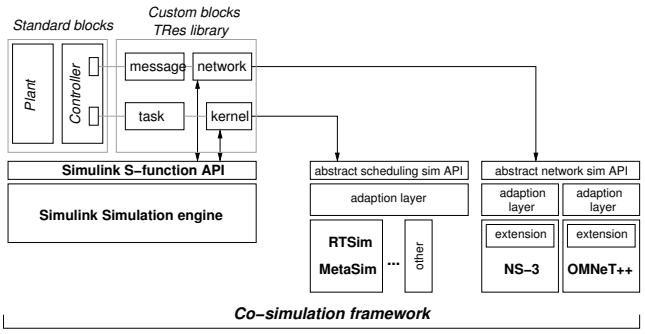


Figure 5: The system co-simulation of the plant and functional controls with the task and network scheduling parts.

A Network Simulator framework is also based upon an event handling mechanism (discrete event simulator) with, on top of it, a message handling mechanism (send/receive) exchanging messages in an OSI conceptual model [1]. The Network API layer defines three virtual classes that are used by the `Network` S-Function: `Network`, `SimMessage` and `Event`. The *object adapter pattern* is still used to bind the abstract interface with the third-party Network simulator.

```
// List of virtual methods to be specialized
// for the adaptation to the Network simulator
virtual void processNextEvent() = 0;
virtual yaks::Event* getNextEvent() = 0;
virtual int getTimeOfNextEvent() = 0;
virtual int getNextWakeUpTime() = 0;
```

Similar to the processor scheduler, our framework includes the adapter towards one existing network simulator. In this case, we realized and tested the connection towards the very popular OMNet++ package, thereby showing the proposed extensibility and adaptation. A CAN protocol implemented on top of OMNet++ [15] has been used as the first example of an embedded communication protocol.

In order to create an instance of a class without making the S-Function (both `Kernel` and `Network`) depend upon the concrete class, the *factory method pattern* [11] is used. Each adapter defines a method called `createInstance()`, which takes a `std::vector` of `std::string` objects as input argument. The `std::string` objects describe a specific configuration for the adapter to be instantiated and are provided by the user through the S-Function mask. For example, a specific configuration for a `Kernel` may define the scheduling policy, the task set, the number of cores, etc. Adding adapters for new external RT scheduling simulators is easy and just requires to register the factory method of the new adapter class to a generic factory class. It does not require any modification to the code of the factory.

4. EXAMPLE

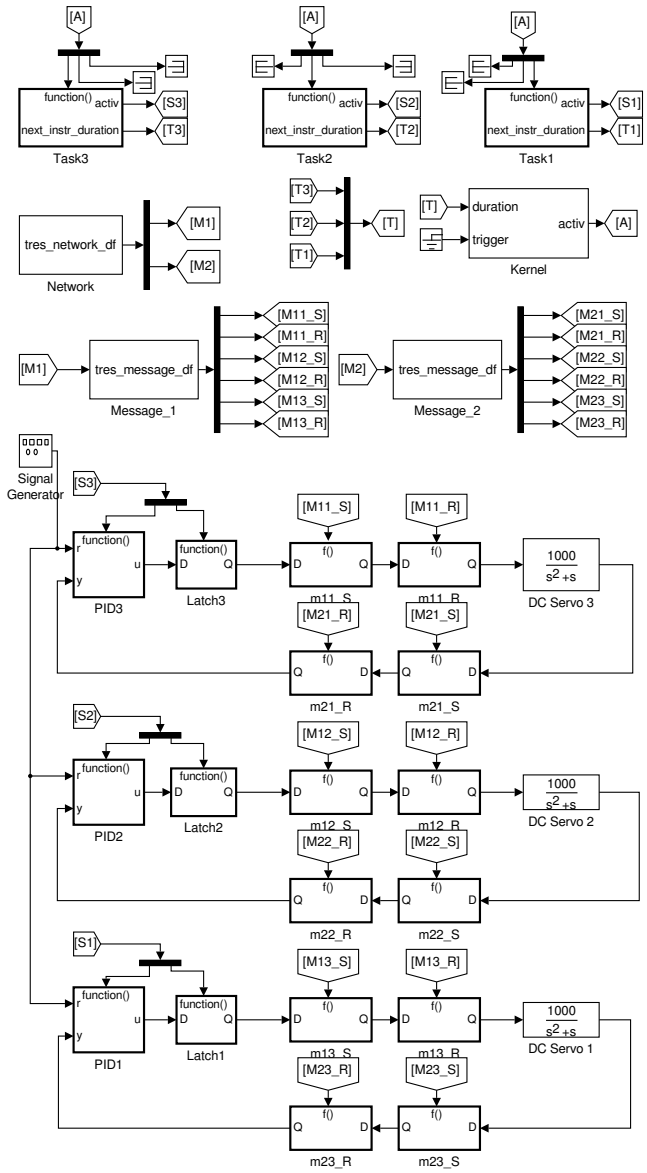
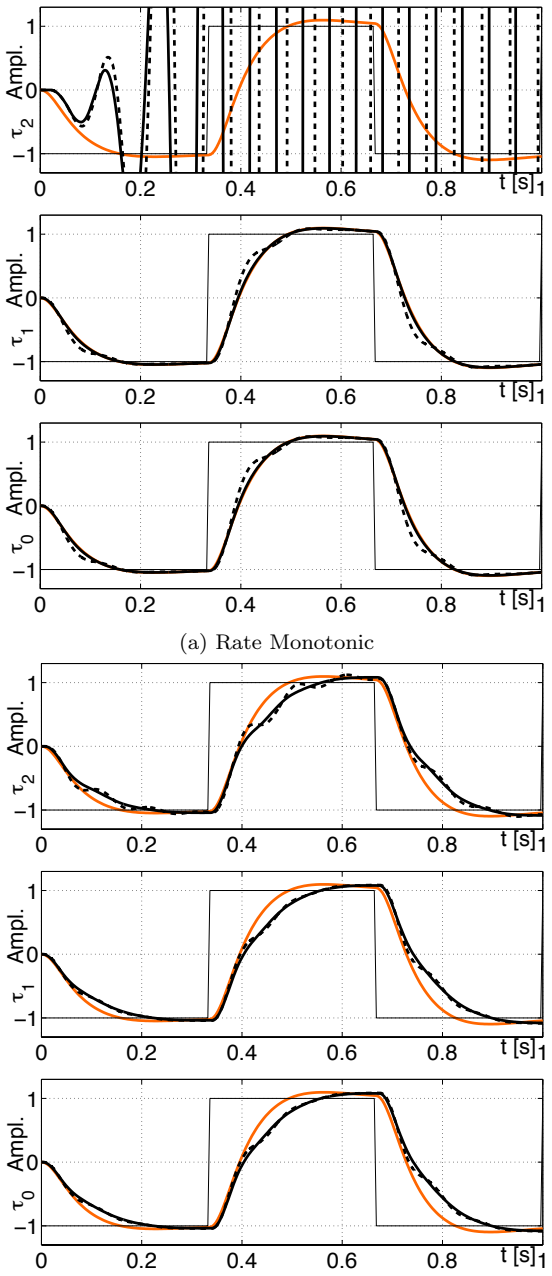


Figure 6: The three servos example adapted from [6].

A case study showing a possible implementation of three servo PID controls, adapted from the TrueTime example library is used to illustrate the capability and the output of the framework. The example consists of three *DC servos* with their continuous-time dynamics (the plant) modeled by a continuous time system and controlled by three (discrete-time) PID controllers with the same coefficients (same functional behavior). The three controller subsystems are mapped for execution on three tasks (τ_0, τ_1, τ_2 each running one segment). The task periods are $T_0 = 4$, $T_1 = 5$ and $T_2 = 6$ *milliseconds* respectively. The example is shown in Figure 6 with the appropriate `Kernel` and `Task` blocks. Tasks specification are supplied as *workspace variables*. The PID controllers (contrary to the TrueTime example) are implemented as Simulink subsystems. Each subsystem is executed in a segment with an execution time of $2ms$, giving a total CPU load higher than 100% (an overload condition). The behavior and the implementation assumptions are the same as in the TrueTime example.



(a) Rate Monotonic

(b) Earliest Deadline First

Figure 7: The motor output positions for the three servos using RM and EDF.

In addition, the position of the motors is read by sensors and sent to the PID using a periodic CAN message m_1. Another periodic CAN message m_2 collects all the command signals from the controls and forwards them to the motors. The period of the two messages is 4 ms , m_1 (the message with the sensor data) has higher priority. The CAN message blocks are shown at the top of Figure 6, right below the task blocks and their simulated transmission is controlled by the Network block (same line, on the left of the figure).

We simulated the example on several runs, with and without networking (and the associated sampling and transmission delays) and using different scheduling policies, such as

the Rate Monotonic (RM, static priority assignment) and Earliest Deadline First (EDF, dynamic priority) policies. The results are shown in Figure 7. The top three graphs show the output of the motors for the three PID controllers when the corresponding task implementations are scheduled using RM. Task priorities are assigned in order to the three tasks, $\pi_0 = 1, \pi_1 = 2, \pi_2 = 3$. The reference signal for the motors is a square wave. As a reference, the output of the controls when the computation and communication delays are not considered (zero logical time execution: a normal Simulink run executing without our framework blocks) is shown as a thick light line. The output of the model with the computation delays only (no networking) is a thick black line and the output when also the message delays are considered is shown as a dashed line.

The top graph in the figure shows how the lowest priority task (τ_2) is losing too many deadlines and the control is not stable when considering the scheduling delays alone (of course, the same is true when communication delays are considered). The use of an EDF scheduling policy with the abortion of tasks trespassing the deadline results in a completely different set of results, as shown on the bottom three graphs of Figure 7. None of the outputs is unstable and, despite some performance degradation (this time the scheduling delay tends to be spread among the three tasks) the motion of the motors is controlled.

5. CONCLUSIONS AND FUTURE WORK

The framework for scheduling simulation in Simulink is part of a larger project aimed at system-level modeling, timing analysis and automatic generation of an implementation for real-time distributed systems [10]. The overall project consists of a merger of MBD and MDE methodologies. The functional model of the physical controlled system and the controller functions is created in Simulink and validated by simulation. Next, a functional abstraction is imported in Papyrus (SysML). The distributed execution architecture, including CPUs, networks, devices, operating systems, resource managers and protocol stacks, is then modeled in SysML and the functional model is mapped onto the execution architecture creatng a model of the software (including task) and message implementation. After mapping, the worst case execution times of functions (implementing subsystems) and tasks are estimated, and the implementation model is then validated against timing constraints and by verifying that the latencies and jitter added by the scheduling and communication delays do not exceedingly deteriorate the performance of the controls.

6. REFERENCES

[1] Iso/iec 7498-1:1994.

[2] A. Aminifar, E. Bini, P. Eles, and Z. Peng. Designing bandwidth-efficient stabilizing control servers. In *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*, pages 298–307. IEEE, 2013.

[3] K. J. Åström and B. Wittenmark. *Computer-controlled systems: theory and design*. Courier Dover Publications, 2011.

[4] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Stress: A simulator for hard real-time systems. *Software: Practice and Experience*, 24(6):543–564, 1994.

[5] G. C. Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*, volume 24. Springer, 2011.

[6] A. Cervin, D. Henriksson, B. Lincoln, J. Eker, and K.-E. Årzén. How does control timing affect performance? *IEEE control systems magazine*, 23(3):16–30, 2003.

[7] A. Cervin, M. Velasco, P. Martí, and A. Camacho. Optimal online sampling period assignment: theory and experiments. *Control Systems Technology, IEEE Transactions on*, 19(4):902–910, 2011.

[8] Y. Chandarli, F. Fauberteau, D. Masson, S. Midonnet, M. Qamhieh, et al. Yartiss: A tool to visualize, test, compare and evaluate real-time scheduling algorithms. In *Proceedings of the 3rd International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, pages 21–26, 2012.

[9] D. Decotigny and I. Puaut. Artisst: An extensible and modular simulation tool for real-time systems. In *5th IEEE ISORC Symposium*, pages 365–372, 2002.

[10] M. Di Natale, M. Bambagini, M. Morelli, A. Passaro, D. Di Stefano, and G. Arturi. Enabling model-based development of distributed embedded systems on open source and free tools. *WATERS workshop at Euromicro ECRTS*, 2012.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[12] G. Lipari. Real-time system simulator `http://rtsim.sssup.it`.

[13] C. Lozoya, P. Martí, M. Velasco, J. M. Fuertes, and E. X. Martin. Resource and performance trade-offs in real-time embedded control systems. *Real-Time Systems*, 49(3):267–307, 2013.

[14] Y. Matsubara, Y. Sano, S. Honda, and H. Takada. An open-source flexible scheduling simulator for real-time applications. In *15th IEEE ISORC Symposium*, 2012.

[15] J. Matsumura, Y. Matsubara, H. Takada, M. Oi, M. Toyoshima, and A. Iwai. A simulation environment based on omnet++ for automotive can–ethernet networks. *Analysis Tools and Methodologies for Embedded and Real-time Systems*.

[16] NS-3. Discrete-event network simulator for internet systems `https://www.nsnam.org`.

[17] OMNeT. An extensible, modular, component-based c++ simulation library and framework for building network simulators `http://www.omnetpp.org`.

[18] SimEvents. A discrete-event simulation engine and component library for simulink `http://www.mathworks.it/products/simevents/`.

[19] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Cheddar: a flexible real time scheduling framework. In *ACM SIGAda Ada Letters*, volume 24, pages 1–8. ACM, 2004.

[20] R. Urunuela, A. Deplanche, and Y. Trinquet. Storm a simulation tool for real-time multiprocessor scheduling evaluation. In *IEEE ETFA Conference*, 2010.

[21] A. Varga et al. The omnet++ discrete event simulation system. In *Proceedings of the European Simulation Multiconference (ESM 2001)*, 2001.

[22] Vector. `http://vector.com`.