Simulating Preemptive Scheduling with Timing-aware Blocks in Simulink

Andreas Naderlinger
Department of Computer Sciences, University of Salzburg
andreas.naderlinger@cs.uni-salzburg.at

***Abstract*—This paper introduces an extension of the modeling and simulation environment MATLAB/Simulink. It enables control and system engineers to consider software execution times, as well as the effects of scheduling and preemption inside software-in-the-loop (SIL) simulations. To this end, we present the concept of a Simulink block whose execution lasts for a finite amount of simulation time. During this time, the simulation engine continues to update the plant or other blocks with outputs that have already been calculated by the block. Execution time information is assumed to be known (or based on some random distribution). Source-level annotating the control software with target specific timing information enables a fine-grained and even a control-flow dependent simulation of the block. We outline the required synchronization with the simulation engine of Simulink. This timing-aware block consumes simulation time in the same sense as a task consumes CPU time on a target. We describe a mechanism to execute a set of such blocks with (potentially cyclic) data dependencies with a static priority scheduler inside Simulink, including support for preemption. The presented approach permits a development process, where a typical time invariant and platform agnostic model is incrementally transformed into a platform-specific one that makes the simulation more realistic.**

I. Introduction

Model-based development is a well established approach in software-intensive systems as found, for example, in the automotive or aerospace sector. In these domains, Simulink [1] is widely considered as the de-facto standard for modeling and simulation of real-time control applications. It is based on the synchronous block diagram (SBD) formalism that shows strong advantages over source code in terms of user (control engineers) acceptance and rapid prototyping capabilities, for example. Additionally, the affinity to the synchronous reactive (SR) paradigm makes SBDs particularly amenable to formal validation. The key assumption of the underlying programming model is that computation happens in zero time and that outputs are provided in reaction to inputs either instantly, without any observable delay, or with a fixed delay that is identical for every invocation [2]. This approach perfectly matches with control engineering theory that typically neglects sampling jitter and assumes constant latencies between reading inputs and writing outputs. Assuming that plant dynamics have been modeled sufficiently, a model-in-the-loop (MIL) simulation allows performance and stability characteristics of the system to be assessed.

Ultimately, however, a code generator turns all the blocks arranged to implement a control law into (typically C) code to be executed on a hardware platform. A hardware platform, which exhibits all the nastiness that was abstracted from in the model. Here, jitter caused by execution time variations or start-time delays may have considerable influence on the system behavior and may compromise its stability. Variations in execution time might stem from different paths through the control-flow graph. Those variations can be significant, since controllers typically operate in one of several modes according to some global system condition and change functionality, e.g., to compensate increased system loads at higher engine speed. Such effects are ignored to a large extent in models with pure SR semantics. In addition, the individual functionalities of a controller that may have been designed and tested independently must not be regarded in isolation on a target platform. Infinite computational resources assumed in simulation face limited capacity of the CPU that has to handle a number of competing tasks that are scheduled by some (probably preemptive) real-time operating system.

Only at the system integration, a system engineer decides on the mapping from controller functionalities to tasks and more recently also to cores such that the overall timing requirements are satisfied. Sources of nondeterminism include execution time variations, varying system load, or preemption. These effects are certainly neglected in Simulink MIL simulations and typically also in software-in-the-loop (SIL) simulations that might at best mimic execution times with constant delay blocks. Therefore, hardware-in-the-loop (HIL) simulations are required to be able to reason about the system behavior and to experiment with different configurations.

A. Contribution

In this paper we present an approach that brings software related concerns into Simulink models, in order to increase their expressiveness. Thereby we consider aspects that exceed typical SIL simulations, such as software execution times and preemption effects.

To this end, we introduce a new type of block, that we call *xTask block* for eXecution-time aware task block. It is based on Simulink's standard S-Function block, but behaves differently in several aspects. Most importantly, its simulation does not follow the zero-execution time behavior. Instead it lasts for a finite amount of simulation time and thereby spans several simulation steps. The exact amount of simulation time may be constant for all invocations or follow a random distribution, for example. An xTask block conceptually corresponds to an OS task. They can be configured to share computational resources

978-3-9815370-8-6/17/$31.00

with all the other xTask blocks within the same Simulink model, in the sense that their execution consumes simulation time and no two tasks can execute at the same time on a single CPU. In order to observe preemption effects among different xTask blocks, simulation time must not pass after the block has been executed, but during its execution, which stands in stark contrast to the regular Simulink SR approach. We also describe the underlying task state model that allows us to incorporate priority-based scheduling inside Simulink simulations.

A development process that uses our approach may start with a time invariant and platform-independent model and evolve to a platform-dependent one. Eventually it allows simulation results to be closer to the real behavior on a hardware platform as it already considers aspects that exceed pure control engineering concepts.

B. Related Work

The advantages of a model-based development are manifold, but its closeness to control theory and ignorance of hard- and software, particularly their effect on timing, leads to a mismatch between simulation results and behavior on an actual hardware platform once the system gets deployed. Several strategies have been proposed to overcome this mismatch. We differentiate between the following strategies:

1) Semantics-preserving Synthesis: Practical implementations of SR models must ensure that a task reads its input values at the occurrence of the triggering event and provides the output before the occurrence of the next event. For the single-processor single-task case (this includes SR models containing multiple blocks but all with the same sampling rate) this results in a simple read-compute-write loop. For the (preemptive) multitasking case, which is eligible for multi-rate models, preservation of the original semantics is difficult. A general discussion is given in [3]. Several papers (e.g., [4], [5]) propose strategies by introducing sample-and-hold elements that ensure correct semantics at the price of additional memory utilization and delays.

2) Platform Abstraction: The logical execution time (LET) [6] concept abstracts from physical execution that is influenced by the hardware, scheduling, system load, etc. It associates a task with a fixed logical time span, during which the task function is executed, optionally suspended, and resumed. I/O operations are considered zero execution time activities and are exclusively performed at the start and the end of the LET, respectively. An LET-based system shows equivalent behavior on any platform for which it has been proven to be schedulable, including distributed platforms [7] as well as simulation environments [8], [9].

3) Co-simulation: In this widely used approach the continuous plant is expressed in a Simulink model, for example, whereas the discrete controller part is specified with another environment that may be based on SystemC [10] or other discrete event (DE) simulators, such as the one described in [11]. Since the two aspects run in two different processes, a communication facility for data transfer and synchronization is required.

4) Platform-augmented SBD Models: Our approach ranks among this category. Approaches found in literature and practice (e.g., as third-party Simulink libraries) vary from simple delay blocks that shall mimic data-transfer delays, to more elaborate implementations like TrueTime [12]. TrueTime is a Simulink toolbox for simulation of distributed real-time control systems and controls CPU and network scheduling. It enables the simulation of a real-time kernel (represented as a Simulink block) that is able to schedule task implementations with different policies. In contrast to our approach, all TrueTime tasks are executed in the context of the kernel block, which consequently subsumes the whole control software. In our case, tasks are represented as individual blocks with their own input and output signals and they can appear at different positions in Simulink's sorted block order (see below). This makes our approach particularly flexible and composable, since tasks can be added to the model without requiring other parts of the model to be changed. We share with TrueTime that the execution time information is assumed to be known for a certain target and that it is part of the task function source code. While TrueTime requires the task function code to be in a special format with multiple entry points, we use a similar approach as the one described in [11] where the task function is instrumented by additional function calls for synchronization. Recent work includes TRES [13], a TrueTime-inspired and modular framework that adds an explicit block representation of tasks, schedulers and network messages to simulate various forms of impact on the timing behavior. While currently being more flexible than our approach, it requires structural changes to the model and assumes the same task format as TrueTime.

C. Simulink Background

Simulink is a modeling and simulation environment for dynamic systems, which may include a continuous representation of the physical plant together with discrete blocks that represent the controller functionality. The system dynamics are implemented as blocks connected by signals between output and input ports. An input port of a block is one of two kinds, *direct-feedthrough (DF)* or *nondirect-feedthrough (NDF)*. While DF inputs directly affect the output of the block at the same time instant (Mealy-type), NDF inputs only affect the block's state (Moore-type). During the simulation loop, the simulation engine repeatedly updates the state of the model by computing the system inputs, outputs, and states in the progress of time at consecutive steps determined by a solver. At each such step, the model can be viewed as an SR model [14]. The execution of the model requires the execution of the individual blocks in a particular order determined during initialization. This so-called *block sorted order* is most importantly determined by data dependencies and feedthrough characteristics of blocks. Two basic rules must hold: *(i)* a block must be executed before any of the blocks whose direct-feedthrough ports it drives and *(ii)* blocks without direct feedthrough inputs can execute in arbitrary order as long as they precede any block whose direct-feedthrough inputs they

drive [1]. Additionally, without violating the two rules above, block priorities assigned to individual blocks may influence the sorted order. To avoid non-deterministic behavior due to causality concerns, Simulink in general requires feedback loops to be resolved by blocks with NDF input.

The *S-Function* mechanism enables the extension of Simulink by custom blocks implemented in C. An S-Function appears as an ordinary block and is implemented as a set of callback functions, which the simulation engine executes at different stages during the simulation. Repeatedly during the simulation loop, firstly the *output function* (`mdlOutputs`) calculates and sets the block's output variables and secondly the *update function* (`mdlUpdate`) calculates and sets the block's state variables. This *function separation requirement* [15] of Simulink includes that an output function must not access NDF inputs. Note that the simulation engine invokes a particular kind of function for all the blocks in a model before invoking the next one. Hence, the blocks don't update their state until all outputs in the system have been set.

II. Tasks as Simulink Blocks

In this section we introduce the concept of xTask blocks, while implementation aspects and how to schedule multiple such blocks in Simulink are described later.

Every xTask block b_i within a model represents a task τ_i in the sense of an operating system, each having an associated task function that contains its implementation. Every input u and every output y to/from the task is modeled as an individual block input/output signal, respectively. The k^{th} job of a task τ_i, denoted as $\tau_i(k)$ corresponds to the k^{th} activation of block b_i that will be denoted as $b_i(k)$. The time instant of the activation will be denoted as $t_i^a(k)$. The characteristics of xTask blocks are in stark contrast with typical blocks and can be summarized as follows:

1) Non-synchronous Execution: As opposed to ordinary blocks, for an xTask block activation $b_i(k)$, the start time of the execution $t_i^s(k)$ and its end time $t_i^e(k)$ do not coincide. So the execution of such a block lasts for a finite execution time $T_i^x(k) > 0$ such that $t_i^e(k) = t_i^s(k) + T_i^x(k)$ and $t_i^e(k) \leq t_i^s(k+1)$ (the preemptive case is discussed below). In the simplest case, T_i^x is constant. However, execution times of the individual activations of b_i may vary, just like the individual jobs of a real-time task typically vary in their execution time. This is due to different input data, states of the task (resulting in different code paths taken) and the entire system including caches. Consequently, the observable output of the task (in our case the output of the block) is deliberately subject to jitter. While executed on a host machine, the obtained behavior of the code shall reflect the timing as if it were executed on a target platform. We use source-level annotation of tasks as an efficient method to incorporate timing information in the task representation [16] (see below).

2) Scattered IO Instants: By separating $t_i^s(k)$ and $t_i^e(k)$, also the time instants for reading inputs and providing outputs are affected. Intuitively, every input u is read at $t_i^s(k)$ while every output y is provided at $t_i^e(k)$. However, this appears to be an unnecessary restriction. Like access to IO variables may be scattered throughout a whole OS task function, input- and output operations of xTask blocks might be desirable also during the whole execution and are allowed at any time during $T_i^x(k)$ (with minor restrictions as explained below). This leads to a fine grained interaction with other blocks, where plant and controllers are always up-to-date.

3) Explicit Task Precendency: Typically, tasks are not independent, i.e., there may exist precedence relations and/or data dependencies. This suggests to categorically model inputs with nondirect-feedthrough characteristics for the following reason. As SR models are causal, a direct-feedthrough connection between two blocks implies a precedence relation such that, if their activation coincide at the same simulation time, the output providing block must be executed before the block that reads the input. In order to consider precedence constraints and data dependencies as two orthogonal aspects, input to xTask blocks must be non-direct feedthrough (NDF). This makes the block dependencies a directed acyclic graph (DAG) and avoids causality issues without introducing any timing errors.

4) Priority: Each xTask block b_i has a unique priority $i \geq 0$ associated that corresponds to the priority of the OS task. Priorities are reflected in the block sorted order of Simulink, with b_0 being the block with highest priority. Explicit priorities facilitate systems with tasks of different levels of criticalness.

5) Scheduling: The execution of $b_i(k)$ may start after its activation ($t_i^s(k) \geq t_i^a(k)$), for example, if the execution of another xTask block b_h with higher priority ($h < i$) prevents b_i from being started. This corresponds to the case where multiple tasks are scheduled under limited resources, like sharing the same CPU. This is an important property that implies that adding a high-priority task block to the model (or increasing the activation frequency of an existing block) influences the timing behavior of the other blocks, just as it would be the case on a real hardware platform.

6) Preemption: For improved responsiveness, operating systems may schedule tasks preemptively and hand control over to the task with higher priority, as soon as such a task becomes ready. Depending on the configuration, also a running xTask block can be suspended to allow a newly arriving task of higher priority to start execution.

7) Combined Output/Update Function: Simulink blocks with NDF input use the output function to write output ports and the update function to read input ports. While this function separation is required by Simulink, it is opposed to a single function with access to input and output variables in arbitrary order in an ordinary task implementation. Although an xTask block uses NDF inputs too, its functionality (including IO accesses) is implemented in a single function, its *task function*. A synchronization mechanism outlined in section II-B ensures the correct timing behavior while respecting Simulink's function separation requirement.

A. Execution-time Awareness

As outlined above, a proper modeling of software tasks that shall exceed pure functional aspects requires a fine-grained

consideration of execution times. We use source-level annotation [16] in order to incorporate timing information in the task representation. Therefore, the source code is (automatically) instrumented continuously with a form of a dedicated *delay* function. This function has a parameter that represents the execution time of a delimited code fragment for a certain execution platform. Those times might have been determined, for example, by profiling or static analysis.

Once activated during simulation, an xTask block gets scheduled to execute its time annotated task function. The timing annotations partition the code of a task function into a finite number of contiguous segments, each having an associated execution time. At segment boundaries, the block synchronizes with the simulation environment and consequently with the plant or other xTask blocks in the model. Only when the simulation time has drawn level, the block executes the next segment. For an accurate observation of IO behavior, at least every read operation that is influenced by the global system state and every write operation that influences this state is preceded by a delay function call.

B. Synchronization

The implementation of the xTask block is based on the S-Function block and its appendant API. We start with an outline of our general synchronization mechanism with the Simulink engine. It respects requirements of both Simulink itself and the xTask properties as listed in the previous section, such as non-synchronous execution, arbitrary IO instants, or the combined output/update function. Scheduling and the task state model to support preemption is described afterwards.

Each time-annotated xTask task function runs in its own execution context (thread). The delay function calls, which have been introduced in the source code, serve as synchronization points between a task and the simulation engine. They ensure that the individual code segments do not execute until the simulation time is in correspondence with the time annotation. Conceptually, the mechanism is based on a monitor (wait/signal) approach. While the task executes, the simulation engine has to wait until execution would exceed the current simulation time. While the task waits, the simulation engine continues execution of other blocks, such as the plant or other xTask blocks, and eventually increases simulation time. At some subsequent simulation step, both the output and the update callback function of the xTask block are candidates to signal the resumption of the task. This follows from the non-direct feedthrough characteristics of xTask blocks. The synchronization mechanism restricts IO access to the appropriate callback function (e.g., reading inputs is allowed only in the update function). It requires, however, that a task does not access an output port after it accessed an input port at the same simulation time instant. This needs to be ensured by the execution time instrumentation.

III. Scheduling & Task State Model

The synchronization mechanism above allows us to simulate non-synchronous activities and to observe the effect of execution time variations of individual tasks. In this section we present a mechanism to study the interaction of multiple tasks and their impact on each other. It demonstrates the applicability of Simulink for simulating the behavior of multiple tasks competing for the common and limited resource *CPU*.

The presented task state model is restricted to tasks without internal synchronization points (cf. OSEK basic tasks, AUTOSAR category 1 runnables). Tasks may, however, have precedence relations and/or data dependencies. In the following discussion we further assume that the task set is schedulable and all tasks are periodic (i.e., all activation times $t_i^a(k)$ of a task τ_i are statically known). Note that start times are generally not known, because the termination times of higher priority tasks are unknown in advance, as they are influenced, e.g., by the taken execution path.

Let $\mathbf{T} = (\tau_0 \dots \tau_n)$ be the list of all tasks τ_i in a system, ordered by their unique priority i with τ_0 being the highest priority task. Every task is characterized by a period π_i and an offset (phase) ϕ_i. We consider a Simulink model that contains for each task a corresponding xTask block b_i with a variable sample-time and a time-annotated version of the task function. Additionally, each xTask block has parameters for the priority, the period and the offset of its associated task.

Figure 1 shows the model of the state machine behind our scheduling approach that is based on OSEK's basic task state model [17] with the states $\{$suspended, ready, running$\}$. The actions to activate, start, preempt, or terminate a task result in the transitions $\{$ACTIVATE, START, PREEMPT, TERMINATE$\}$ and are known from many schedulers. However, we enable state transitions only at well-defined points that are aligned with the simulation engine. We introduce the following input signals to coordinate the task transitions with the simulation engine: $\{$tSim, OUTPUTS_0, …, OUTPUTS_n, UPDATE_0, …, UPDATE_n, SYNC_i, $\text{EOT}_i\}$. The signals OUTPUTS_i and UPDATE_i are enabled/disabled when the Simulink engine enters/leaves the callback function `mdlOutputs` and `mdlUpdate` of block b_i, respectively. Consequently, the OUTPUTS and UPDATE signals are mutually exclusive and appear in the order (OUTPUTS_0, …, OUTPUTS_n, UPDATE_0, …, UPDATE_n) at every time step (assuming that all blocks are executed at every time step). During an enabled OUTPUTS/UPDATE signal we say that the simulation engine is in the *outputs/update phase*. This differentiation is required as a consequence of Simulink's function separation requirement and the block sorted order (see section I-C). According to the current simulation time tSim, the scheduler decides to eventually resume with the next code segment of a tasks.

The two remaining signals SYNC_i, EOT_i, and the variables $\{\text{tProc}_i$, tCode_i, tSync_i, readyQ$\}$, which complete the definition of our state machine, are explained below.

Most scheduling activities of all tasks τ_i are performed at the beginning of the update phase, during the execution of `mdlUpdate` of the highest priority block b_0 (i.e., when UPDATE_0 is enabled): ACTIVATE (1), PREEMPT (2), START (3) and also transition (0). Only the transition with the TERMINATE action (5) and the self-loop transitions (4,6) in the running state of task τ_i are performed in the scope of their own

`mdlOutputs` or `mdlUpdate` callback functions (during an enabled OUTPUTS_i or UPDATE_i signal).

The number associated with an arrow in Figure 1 not only helps to identify the respective transition, it also represents the priority of a transition (as introduced in [18]) in the following sense. Every transition guard with a particular priority k is evaluated sequentially for every task τ_i before evaluating the n transitions with priority $k+1$. For example, we decide on the activation of every task (1) before the currently running task is potentially preempted (2).

As a consequence of the OUTPUTS_0/UPDATE_0 guard, the transitions 0/1,2,3 (represented with a thick arrow) may be performed for all tasks only during the execution of the output/update function of block b_0. The transitions 4,5,6/4,6 for task τ_i may be performed only in the outputs/update function of the corresponding block b_i, respectively. Dotted arrows represent transitions that are not performed within the context of the simulation engine, but in the context of the task.

The output phase is only relevant for the sole task in the running state (if there is one at all). First (transition 0), we increment its variable tProc_i by the step size Δt, to reflect that this very task had the processor since the last simulation step. Next (transition 4), we block the simulation engine during OUTPUTS_i, perform a context switch and resume with the execution of the task function. The function executes until hitting a synchronization call (indicated by the occurrence of a SYNC_i input event with synchronization time parameter tSync_i). The synchronization mechanism ensures that the task execution does not run to completion but keeps in sync with the simulation environment. It also guarantees that task outputs are only written during the outputs phase (i.e., within the `mdlOutputs` callback function). On every synchronization point, we compare the times the task was allowed to have the processor with the instrumented execution times in the code (summed up in tCode_i). Transition 6 indicates that eventually the task execution blocks and the context is switched back to the simulation engine. On a potential completion of the task function of task τ_i (indicated by the end-of-task input event EOT_i), the task is terminated and set to suspended state (transition 5).

At the beginning of the update phase (during UPDATE_0), we first (transition 1) decide on the activation of all suspended tasks. The suspended state is the initial state for every task. When the offset ϕ_i and period π_i property match with the current simulation time tSim, the task is added to the queue of ready tasks (readyQ) and its state is set to ready. It follows the potential preemption of the running task (transition 2), if a task in the ready queue has higher priority. Next (transition 3), if such a higher priority task was found, it is removed from the ready queue and its state is marked running. Then the running task may continue during UPDATE_i with the execution of its task function by switching to the task execution context again (transition 4) until the synchronization mechanism decides to block it and give control back to the simulation engine (transition 6).

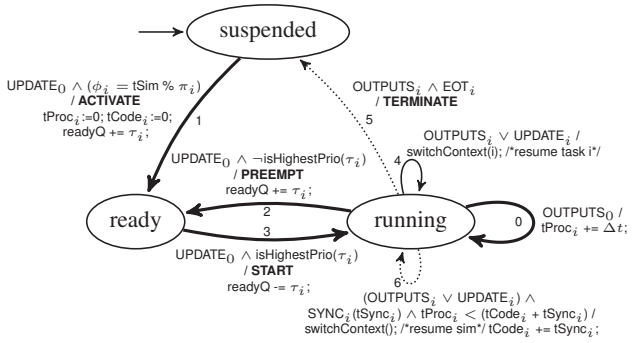

Fig. 1. State model of task τ_i

IV. Evaluation

We apply our approach in two different scenarios. In the first one, we use it to evaluate the stability of a system when adding new computational tasks. The second one demonstrates the practicability of the approach with synthetically generated tasks.

1) Automatic Transmission Controller Demo: We use a model that is based on Simulink's automatic transmission controller demo model *sldemo_autotrans*. It contains a plant that is in closed-loop with a Stateflow block implementing the shift logic, which we assume for system integration shall be executed as one of three tasks with known execution times under rate monotonic scheduling. In a first step, the Stateflow block has been replaced by an xTask block. Its task function is the time-instrumented version of the C-code representation of the original Stateflow block as generated by Simulink Coder with about 100 code segments in the sense of section II-A. In a second step, two further xTask blocks τ_0 and τ_2 have been added to observe their impact on τ_1 (see Fig. 2).

We assume the following task configuration: $\tau_0(\pi_0 = 0.2s, \phi_0 = 0.15s)$, the shift logic task $\tau_1(\pi_1 = 0.35s, \phi_1 = 0s)$, and $\tau_2(\pi_2 = 0.7s, \phi_2 = 0.2s)$.

Figure 3 shows the impact on the engine rpm signal. Signal *reference* shows the original signal (standard Simulink/Stateflow), signal *execTime* shows the rpms with the shift logic implemented as xTask block but without τ_0 and τ_2 in the model. The difference to the original signal reflects the instrumented execution times of the task that we chose to range up to $0.05s$. The third signal *preemption* shows the behavior when τ_0 and τ_2 are added to the model. Now the execution of the shift logic gets delayed and preempted by the higher priority task τ_0 which results in a degraded performance. Experiments showed that setting ϕ_0 to a value between $0s$ and $0.12s$ again leads to tolerable signal output.

2) Synthetically Generated Tasks: This scenario uses characteristics from typical industrial applications [19] where a large number of runnables (functions) with varying execution times are mapped to 9 tasks with harmonic periods $(1, 2, 5, 10, 20, 50, 100, 200, 1000ms)$. We use a configuration where each task is represented as an xTask block and each runnable is an empty but time-instrumented code segment

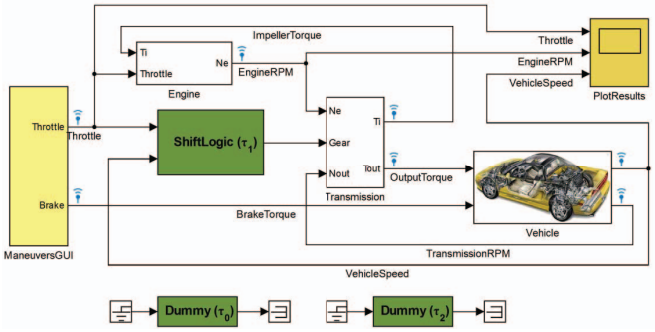

Fig. 2. Modified transmission controller demo (Scenario 1)

as explained in section II-A. Table I shows the results for simulating the model with three different numbers of runnables for 1 second in each case. The number of required context switches between the simulation engine and the different tasks and the number of task state transitions (preemptions, etc.) depend on the runnable execution times, which follow a random distribution. The accumulated times result in the given simulated target CPU load (with task switching overhead assumed to be zero). The last column represents the time it took to simulate the model with a variable-step solver on a 2.60 GHz host PC (including execution time randomization and logging). We consider this overhead to be acceptable for task/runnable sets of these sizes, if not neglectable in a context with complex plants and control tasks.

V. Conclusion

Typical real-time control software development is a co-design problem. Apparently, there is a gap between the control engineering aspect being considered in modeling and simulation tools such as Simulink and the real-time engineering aspect taking the hardware/software architecture of the target platform into account. This makes system integration difficult and expensive. The approach presented in this paper enables simulations that consider software execution times and scheduling effects of control tasks with minimal or no architectural changes in the models. To this end we discussed the concept of a new block that is based on a time-annotated function, which might stem, for example, from a synthesized Stateflow block. We showed the feasibility to simulate a set of such blocks executed under a rate monotonic scheduling in Simulink. Future work includes support for multi-core scheduling.

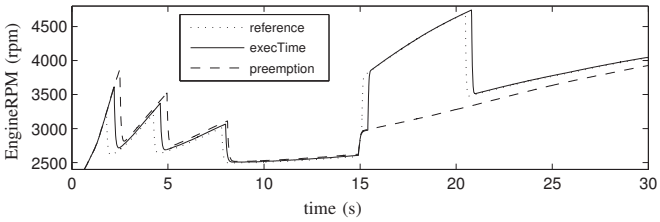

Fig. 3. *EngineRPM* signal of the case study (Scenario 1)

TABLE I
Evaluation with 9 tasks (Scenario 2)

#runnables	#runnable invocs.	#transitions: preempt. / total	#context switches	CPU load [%]	time [s]
425	42, 095	171 / 6,000	337,797	31.9	8.9
850	84, 190	489 / 6,636	676,465	63.6	17.1
1275	126, 285	992 / 7,502	1,015,823	95.5	26.4

References

[1] The MathWorks, "Simulink Reference, R2016a," 2016.

[2] C. Kirsch and R. Sengupta, *Handbook of Real-Time and Embedded Systems*. Chapman & Hall/CRC, 2007, ch. The Evolution of Real-Time Programming.

[3] S. Baruah, "Semantics-preserving implementation of multirate mixed-criticality synchronous programs," in *Proceedings of the 20th International Conference on Real-Time and Network Systems*, ser. RTNS '12. New York, NY, USA: ACM, 2012, pp. 11–19.

[4] M. Di Natale, L. Guo, H. Zeng, and A. Sangiovanni-Vincentelli, "Synthesis of multitask implementations of simulink models with minimum delays," *Industrial Informatics, IEEE Transactions on*, vol. 6, no. 4, pp. 637–651, Nov 2010.

[5] P. Caspi, N. Scaife, C. Sofronis, and S. Tripakis, "Semantics-preserving multitask implementation of synchronous programs," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 2, pp. 15:1–15:40, Jan. 2008.

[6] T. Henzinger, B. Horowitz, and C. Kirsch, "Giotto: A time-triggered language for embedded programming," *Proceedings of the IEEE*, vol. 91, pp. 84–99, January 2003.

[7] E. Farcas, C. Farcas, W. Pree, and J. Templ, "Transparent distribution of real-time components based on logical execution time," *SIGPLAN Not.*, vol. 40, no. 7, pp. 31–39, 2005.

[8] P. Derler, A. Naderlinger, W. Pree, S. Resmerita, and J. Templ, "Simulation of let models in simulink and ptolemy," in *Proceedings of the 15th Monterey Conference on Foundations of Computer Software: Future Trends and Techniques for Development*, ser. Monterey'08. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 83–92.

[9] A. Naderlinger, "Multiple real-time semantics on top of synchronous block diagrams," in *Proceedings of the Symposium on Theory of Modeling & Simulation - DEVS Integrative M&S Symposium*, ser. DEVS 13. Society for Computer Simulation International, 2013.

[10] F. Bouchhima, M. Briere, G. Nicolescu, M. Abid, and E. Aboulhamid, "A systemc/simulink co-simulation framework for continuous/discrete-events simulation," in *Behavioral Modeling and Simulation Workshop, Proc. of the 2006 IEEE International*, 2006, pp. 1–6.

[11] S. Resmerita, P. Derler, W. Pree, and K. Butts, *Real-time Simulation Technologies: Principles, Methodologies, and Applications*. CRC Pr. Int., 2012, ch. The Validator tool suite: filling the gap between conventional SIL and HIL simulation environments.

[12] D. Henriksson, A. Cervin, and K.-E. Arzen, "TrueTime: Real-time control system simulation with MATLAB/ Simulink," in *Nordic MATLAB Conf.*, 2003.

[13] F. Cremona, M. Morelli, and M. Di Natale, "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*, ser. SAC '15. New York, NY, USA: ACM, 2015.

[14] E. A. Lee and S. A. Seshia, *Introduction to Embedded Systems - A Cyber-Physical Systems Approach*, 1st ed. Lee and Seshia, 2010.

[15] B. Denckla and P. J. Mosterman, "An intermediate representation and its application to the analysis of block diagram execution," in *Proc. of the 2004 Summer Computer Simulation Conference (SCSC'04)*, 2004.

[16] T. Meyerowitz, A. Sangiovanni-Vincentelli, M. Sauermann, and D. Langen, "Source-level timing annotation and simulation for a heterogeneous multiprocessor," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '08. ACM, 2008.

[17] OSEK/VDX, "Operating system spec. 2.2.3," 2005.

[18] C. André, "Syncharts: a visual representation of reactive behaviors," I3S, Sophia-Antipolis, France, Tech. Rep. RR 95–52, 1996.

[19] S. Kramer, D. Ziegenbein, and A. Hamann, "Real world automotive benchmarks for free," in *Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems (WATERS)*, 2015.