Proceedings of the 2017 Winter Simulation Conference
W. K. V. Chan, A. D'Ambrogio, G. Zacharewicz, N. Mustafee, G. Wainer, and E. Page, eds.

SIMULATING EXECUTION TIME VARIATIONS IN MATLAB/SIMULINK

Andreas Naderlinger

Department of Computer Sciences
University of Salzburg
Jakob-Haringer-Straße 2
5020 Salzburg, AUSTRIA

ABSTRACT

Software-induced delays have a significant impact on real-time control system performance. While in extreme cases, jitter caused by execution time variations or start-time delays may compromise stability, such effects are often ignored in model-based approaches and simulations. We address this issue and discuss a Simulink block with a fundamentally new semantics that is used to encapsulate control functions as software tasks. It is particularly suited to model and observe software delays in complex control functions within software-in-the-loop (SIL) simulations. For this purpose, the controller source code is associated with platform-specific execution time information, which we assume to be available. We detail a synchronization mechanism between Simulink and such time-annotated task blocks implementing the control laws. It allows us to perform SIL simulations where controllers execute for a finite amount of simulation time and span multiple simulation steps at which they may interact with other controllers or the plant.

1 INTRODUCTION

Synchronous block diagrams (SBD) (Edwards and Lee 2003) are a popular approach to develop embedded control software in a model-based process. Appropriate tools feature the specification, simulation, and code generation of control algorithms. In this respect MATLAB/Simulink (The MathWorks 2016), which was used in this work, is the de facto standard in many domains. The advantages of such a model-based development are manifold (Badreddine and Hoadley 2010), but its closeness to control theory may bring also a drawback. Usually, 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. The key assumption of the underlying programming model, which is closely related to synchronous reactive (SR) programming, is that computations execute in zero time and outputs are provided instantly in reaction to inputs without any observable delay or after a constant delay (Kirsch and Sengupta 2007). By contrast, on a hardware platform variable software-induced delays may lead to a different system behavior. Delays and jitter may result in performance degradation or even in an unstable control system (Marti et al. 2001). Furthermore, scheduling and preemption are typically ignored during simulation. The aim of this work is to provide a means for simulating these effects on a fine grained level with SBD models. Jitter, i.e., deviations in periodicity of signals or events, comes in different flavors. In this paper we focus on jitter due to variable execution times of control algorithm implementations. Execution time of software tasks depends considerably on the path taken through the control-flow graph. Controllers typically have lots of mode-dependent behavior. Automotive applications, for example, change functionality at run-time to adapt computational demands according to the crank angle in order to avoid system overload at high engine speeds (Biondi, Di Natale, and Buttazzo 2015). Execution times of one and the same task may vary in the order of magnitudes leading to massive output jitter. In complex models, multiple state machines decide on some global system conditions and thus on the holistic computational load; characteristics, which are ignored to a large extent in models with pure SR semantics.

We envision a development that may start with a time-invariant and platform-independent model and evolve to a platform-dependent one. Eventually it allows simulation results to be close to the real behavior on a hardware platform as it already considers aspects that exceed pure control engineering concepts. To this end, we present a synchronization mechanism for Simulink that supports blocks that are not characterized by the typical zero execution time behavior. Instead, the execution of such blocks may last for a finite amount of simulation time and may span several simulation steps. We use this feature to simulate software-induced delays such as execution time variations and start-time delays. Insights gained from simulations that consider execution times can be used in approaches like jitter margin (Cervin et al. 2004) to assess the stability of controllers. The presented synchronization mechanism was recently applied to simulate a set of tasks under preemptive fixed-priority scheduling inside Simulink (Naderlinger 2017).

The remainder of the paper is organized as follows. We continue with a discussion on related work and the necessary Simulink background. Section 2 describes the core idea of non-synchronous block execution. The required synchronization algorithm is in detail presented in section 3 for both direct feedthrough (Mealy-like) and nondirect feedthrough (Moore-like) blocks. Finally, we conclude the paper in section 4.

1.1 Related Work

Co-simulation is a widely used approach in model-based development, where the continuous plant is expressed, e.g., in a Simulink model whereas the discrete part is specified and simulated with another environment that may be based on a discrete-event simulator like SystemC (Bouchhima et al. 2006) or others (Resmerita et al. 2012). The two aspects run as two independent processes that interact with each other, for example, via a handshake protocol for data transfer and synchronization. Our approach is different. We augment the SBD model itself by platform specifics. Approaches found in the literature and also found in practice (e.g., third-party Simulink libraries) are various. They rank from simple delay blocks that shall mimic data transfer delays in the simulation of distributed systems over source or target blocks that represent peripherals and are used for code generation up to more elaborate implementations, like TrueTime (Henriksson, Cervin, and Arzen 2003), that control CPU and network scheduling. TrueTime is a Simulink toolbox for simulation of distributed real-time control systems. It allows the simulation of a real-time kernel (represented as a Simulink block) that is able to schedule task implementations with different policies. It facilitates the simulation of control software close to its true temporal behavior and to consider different network protocols, but requires task functions to be in a special format with multiple entry points. The required code structure of the functionality code is basically a switch statement and each code segment with an associated execution time corresponds to one case block. On each invocation the switch condition variable is incremented such that the individual case blocks are executed sequentially. The execution times of the individual case blocks are hard-coded as their return values. In contrast, we use a similar approach as described by Resmerita et al. (2012) where the task function is instrumented by additional function calls for synchronization. Recent work includes TRES (Cremona, Morelli, and Di Natale 2015), which builds on TrueTime and introduces explicit representations of tasks and schedulers in the model. Both approaches require structural changes in the model, while our aim is to introduce execution times without adding new signals or blocks to the model. Stability analysis regarding jitter effects have been discussed, e.g., by Velasco et al. (2005). Our approach is closely related to MATLAB/Simulink. However, its core idea is applicable also for other simulation environments that pursue a similar simulation strategy with a static block execution order and use separate output/update functions (Denckla and Mosterman 2004).

1.2 Simulink Background

Simulink is a modeling and simulation environment for dynamic systems. Its principal model of computation is that of continuous-time, but it also supports discrete-time models by viewing discrete-time signals as piecewise-constant continuous-time signals (Lublinerman and Tripakis 2008). The system dynamics are implemented as blocks. In order to support hierarchical systems, models may be nested in macro blocks,

called subsystems. Multiple blocks form a network (as a directed graph) by directed links called signals between an output port of a (source) block and an input port of a (target) block. Input ports are either *direct feedthrough (DF)* or *nondirect feedthrough (NDF)*. While DF input ports directly influence the output of the block at the same time instant (*Mealy-like*), NDF input ports affect only the block's state (*Moore-like*). We consider blocks that have inputs of the same kind only and use the term *DF block* and *NDF block*, respectively.

During initialization, Simulink derives a fixed order in which blocks are executed, the so-called *block sorted (execution) order*, which is determined by the data-flow (i.e., the topological dependencies) and the feedthrough characteristics of blocks: *(i)* a block must be executed before any of the blocks whose direct feedthrough ports it drives and *(ii)* nondirect feedthrough blocks can execute in arbitrary order as long as they precede any block whose direct feedthrough inputs they drive. This results in a sorted order in which NDF blocks appear at the head of the list in no particular order followed by DF blocks in data-flow order (The MathWorks 2016). Additionally, without violating the two rules above, block priorities assigned to individual blocks may influence the sorted order. In 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. A solver determines the time steps between consecutive updates of the system. At each such step, a continuous time model can be viewed as a synchronous reactive (SR) model (Lee and Seshia 2010). In the SR model of computation, execution takes zero time. We consider a block b_i that is executed periodically with an offset (phase) ϕ_i and a period π_i. For $k \in \mathbb{N}_0$, $b_i(k)$ is the k^{th} instance of b_i being activated at time $t_i(k) = (\phi_i + k\pi_i) \geq 0$. Due to the zero-time semantics, at this point in time the activation of the execution (i.e., its triggering), its start, its termination and consequently all its input/output operations coincide. Also, data propagation from a source b_i that is directly connected to a destination block b_j happens in zero time, such that the input of the k^{th} instance of b_j is equal to the output of the last occurrence of b_i until $t_j(k)$.

In order to avoid non-deterministic behavior as a consequence of causality concerns in models that exhibit some kind of feedback character, Simulink restricts the dependencies at each step to be acyclic. So in general, block execution at a particular step must form a directed acyclic graph (DAG), which is a common constraint leaving enough expressiveness but easing the ability to argue about the models behavior. It is important to note that this does not require the network (blocks connected by signals) itself to be a DAG. Only the causality loop at execution needs to be broken. This is the case, if every cycle contains at least one NDF block, which serves as the starting point. A model for which this property holds is called *acyclic* to which we restrict ourselves in this paper.

The Simulink block library features numerous built-in blocks. Additionally, the *S-function* mechanism enables the extension of Simulink by custom blocks implemented, for example, 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. The two most important and prominent functions are the *output function* `mdlOutputs` and the *update function* `mdlUpdate`. The output function calculates and sets the block's output signals, whereas the update function calculates and sets the block's state variables. While all the blocks may access their input signals to update their state in the update function, only DF blocks are able to access them in the output function.

Figure 1 illustrates the main interaction points of the simulation engine with the S-function API for blocks with non-continuous state during a simulation run. Functions that are grayed out are optional and depend on block settings. Note that the engine invokes a particular kind of callback function for all the blocks in a model before invoking the next one. So, the blocks in a model do not update their state (in `mdlUpdate`) until all outputs in the system have been set (in `mdlOutputs`). We give a short explanation of the relevant remaining functions in section 3. After all blocks have been executed for a particular simulation step, the simulation engine increments the simulation time and resumes to the next step. Within a callback function, the S-function may access block specific or global information and data-structures via the SimStruct-API (The MathWorks 2016).

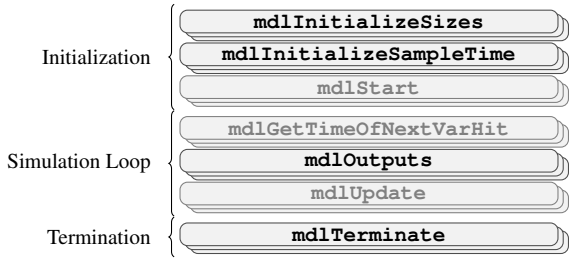


Figure 1: Engine interaction with the S-function API.

2 TIME-ANNOTATED CONTROL FUNCTIONS

This section describes the main characteristics of a custom Simulink block that is based on the standard S-function API but behaves fundamentally different. The execution mechanism that allows us to simulate such a behavior in Simulink without modifying the simulation engine is subject of the next section.

We consider a model for software-in-the-loop (SIL) simulation that contains blocks representing the physical plant as well as one block for each controller of the system. Each controller is implemented as a C function that could be a legacy function or the outcome of a code generation process that originated in a complex Simulink or Stateflow subsystem, for example. Basically, it is the same function that is later executed on a target system, maybe now only contained in some wrapper-function in order to satisfy calling conventions of the simulation engine. Again, according to SR semantics, each control function is executed in zero time. When executed on the target system, however, the execution of a control function is known to last for a certain amount of time that depends on several factors, including the global system state and its influence on the taken path through the control-flow graph of the function. We argue that a proper modeling of the software aspect requires the consideration of execution times inside SIL simulations and introduce the concept of *time-annotated control functions* in Simulink. A time-annotated control function is represented as a Simulink block that is implemented with the standard S-function API. In the following we use the term *task block* to refer to such a block as we assume that each block represents one task in the sense of a target operating system task.

In contrast to the standard Simulink semantics described above, for the k^{th} activation $b_i(k)$ of a task block b_i, the start time $t_i^s(k)$ and end time $t_i^e(k)$ do not coincide. Instead, $t_i^e(k) = t_i^s(k) + T_i^x(k)$, where the execution time $T_i^x(k) > 0$ and $t_i^e(k) \leq t_i^s(k+1)$. Thus the observable execution of the block may span multiple simulation steps and the execution time T_i^x may vary for each activation of the block to reflect the varying execution times on a target platform. Meyerowitz et al. (2008) and Resmerita et al. (2012) have shown that source-level annotation is an efficient method to incorporate timing information into the control functions. Timing annotations partition the code into a finite number of contiguous segments. Each segment has an associated execution time that indicates how long it would take to execute this code portion on a given platform. The same amount of simulation time should pass by for executing this very code portion in the simulation. One segment should cover at most a single edge in the control flow graph of the function. The actual execution times might have been determined, for example, by measurements or static analysis. The annotation may encode the timing information directly or simply represent a symbol to identify a particular code segment. The latter would allow one to randomly choose between given best- and worst-case execution times according to some distribution function. Code segments are delimited by dedicated source code instructions (`sync`) that prevent segments from being executed immediately one after the other. Let $\Sigma_i^k(s)$ be the sum of execution times that accumulate for the k^{th} activation of block b_i until the execution of some source code statement s. The time when statement s is executed is denoted as $t(s,k)$ and defined as follows: $t(s,k) = t_i^s(k) + \Sigma_i^k(s)$. A code segment executes in zero (simulation) time, i.e. for two statements s_1, s_2 in the same code segment, $t(s_1,k) = t(s_2,k)$. Thus, in this paper we consider time-annotated control functions to be executed in temporal isolation (Craciunas et al. 2013) in the sense that multiple independent control functions within the same model are executed concurrently,

without having any impact on each other w.r.t. their temporal behavior. However, control functions can also be configured to consume simulation time in the sense that no other control function is able to execute at the same time. This corresponds to scheduling a set of tasks on a single-core CPU (Naderlinger 2017).

The segmentation of the function can be performed automatically. The resulting segment boundaries serve as synchronization points between the control functions and the simulation environment and consequently also between multiple control functions. At these synchronization points (i.e., `sync` statements), the execution of the control function halts and lets the plant catch up. Clearly, the interval of synchronization points decides on the accuracy of the synchronization. Each access to an input or output signal should be immediately preceded by a synchronization statement (Resmerita et al. 2012). As a consequence, we can control the exact moments of communication with the environment and subsequently also between different controllers or parts thereof. This leads to a fine-grained interaction with other blocks, where plant and controllers are always up-to-date. With a minor restriction discussed in section 3.2, a control function may read/write its in-/output signals anytime within the interval $[t_i^s(k), t_i^e(k)]$, which is in stark contrast to the typical SR semantics of blocks.

3 SYNCHRONIZATION MECHANISM

To determine the behavior of the whole system over time, the model is solved at particular points in time, also referred to as *time steps*. At a particular time step, the simulation engine of Simulink executes the callback functions of the individual blocks in the order described above. They are executed synchronously (i.e., the engine waits for them to return), sequentially (one after the other), they run from the beginning of the function until they return, and no simulation time passes during their execution. A callback function is a *subroutine* (Knuth 1997) in the sense that each invocation returns only once and that it does not hold any state between two invocations. The behavior of a time-annotated control function as introduced above is different. Simulation time shall pass during its execution, and other controllers or ordinary blocks shall execute while the time-annotated function has still not returned yet. A time-annotated control function stops at the end of a code segment and resumes with the next segment at a subsequent time step. Conceptually, this corresponds to the behavior of *coroutines* (Knuth 1997) where the synchronization statement yields control to the simulation engine.

In the following we describe a mechanism to impose the behavior of time-annotated control functions upon callback functions. In particular it executes the time-annotated control function within the scope of a custom *task block* S-function (see above). The mechanism must satisfy the intended properties but also comply with Simulink requirements as, for example, the S-function API, the sorted block order, and the function separation requirement. From the perspective of the simulation engine, the changed semantics must be transparent, i.e., it executes the task block as an ordinary S-function by invoking its callback functions in the usual way. Furthermore, the simulation engine expects access to the SimStruct-API to happen only at specific, well defined, times. This includes, e.g., that the output of a block can only be changed, while the simulation engine is executing the output function of this particular block.

The approach presented in this paper is based on pthreads (IEEE Std 1003.1c-1995 1995). Each time-annotated control function runs in its own execution context (thread) with a separate stack and logically multiple such functions within a model execute concurrently. In order to confine the access to the SimStruct-API to the allowed callback functions and time points, the threads must be rigorously synchronized with the simulation engine. Failure to do so causes MATLAB to crash.

In the following we regard the simulation engine that synchronously executes the callback functions as a single and dedicated thread referred to as *Simulink thread*. Furthermore, each task block in a model has an associated private thread referred to as *task thread*. Depending on the feedthrough characteristics of a task block, we need two slightly different synchronization mechanisms. We start with the direct feedthrough case. Note that although the block is direct feedthrough, the output will have a delay that corresponds to the time instrumentation.

3.1 Direct Feedthrough

The direct feedthrough characteristics allow us to both read and write signals in the block's *output function* and no *update function* is required. This leads to a straightforward implementation of the synchronization mechanism. Figure 2 illustrates the synchronization mechanism between a direct feedthrough task block τ and the simulation engine of Simulink.

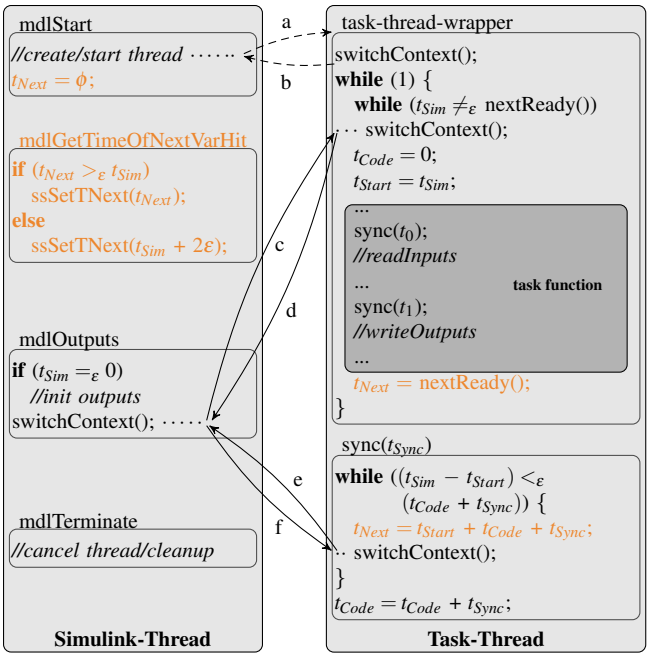


Figure 2: Direct feedthrough synchronization.

The Simulink thread (left part) executes the Simulink engine, which synchronously calls the S-function callbacks of the individual blocks (amongst others the callbacks of τ as listed in the figure). The interaction between the Simulink engine and the S-function API was outlined above. Code segments in orange color concern optimizations and are explained further below in section 3.1.1. The function `mdlStart` invoked in the initialization phase of the simulation creates and starts a thread dedicated to the execution of task τ (a). This task thread (right part of Figure 2) immediately blocks as a consequence of executing `switchContext` and control is given back to the Simulink engine (b). The function `switchContext` (not shown) uses a wait/signal approach based on a condition variable and is aware of the two threads. On each invocation from either side it suspends the current thread and resumes with the other one. Therefore, Simulink then resumes with the execution of the function `mdlStart` of other blocks that come afterwards in the block sorted order to complete the initialization phase. In the next phase (the simulation loop), the engine invokes `mdlOutputs`, where at simulation time 0 initial values of output ports are written. Then the Simulink engine blocks, as a consequence of `switchContext`, and control is given to the task thread which resumes execution (c). The task thread executes an endless loop, where each iteration corresponds to one invocation of the controller (which corresponds to one job of a task in operating system terms). This loop is an optimization to avoid the creation of a new thread for each job. While the task is not yet ready (in the first iteration this means that the task's offset is still greater than the current simulation time t_{Sim}), the thread blocks and control is given back to the Simulink engine (d). We assume that t_{Sim} is always kept up-to-date. The engine resumes simulation and eventually invokes `mdlOutputs` again at a later simulation time where it blocks and gives control back to the task thread. As long as the task is not ready, control is passed between the Simulink thread, which increases simulation time, and this very position in

the task thread (d). Once the task is ready, execution resumes with some housekeeping: variable t_{Code} stores the sum of the execution time that has passed while executing the code of the control function, t_{Start} is the simulation time at which the current job got started. The actual task function containing some control law implementation, for example, is executed next. For convenience, the code of the task function is part of the task-thread wrapper in the figure. The task thread executes the time-annotated task function instruction by instruction until a `sync` statement is hit. On each such call to `sync` the thread either resumes execution or blocks and the simulation engine resumes. The parameter of the `sync` function call (e.g., t_0) refers to the execution time of the code portion between the last reached `sync` statement (or since the start of the task function, in case of the first occurrence of the `sync` statement) up to this very point. Only when the simulation time has drawn level with the execution time of the code (e, f), the task increments t_{Code} and resumes further with the next instruction of the task function. Note that the context switch conditions may be checked on both sides in order to avoid unneeded switching overhead. In the direct feedthrough approach, there is no limitation regarding the order of accessing input and output ports and the time that passes in between those accesses. For example, it is allowed to alternate input and output access without time being consumed in between.

The task block is made available in the Simulink Library Browser and has four mask parameters that allow the user to specify the name of the task function as well as its period, offset, and priority (see later). It is implemented as an S-function and contains the synchronization mechanism. The implementation of the task function needs to be provided separately (compiled as a MEX file). A sample implementation for a task function *Task_1* is shown in Listing 1 together with additional hook functions, which are called by their respective namesakes (e.g., `mdlInitializeSizes` calls the `initializeSizes` hook to specify the numbers of in-/output ports and their dimensions). In this particular example, *Task_1* reads its input as soon as it is started. Also, it writes its output after 2 µs and upon completion, i.e., after 3 µs. The functions `syncInput`/`syncOutput` invoke the synchronization function with the supplied timing parameter (the third argument) and read/write the indicated input/output port (the first argument) using, for example, `ssGetInputPortSignalPtrs`/`ssGetOutputPortSignal` from the S-function API.

Listing 1: Sample task function.

```
1  #define S_FUNCTION_NAME Task_1 /* task name */
2  #include "task.h" /* includes for the sync. mechanism */
3
4  void start(Task *t, SimStruct *S) {} /* optional */
5  void terminate(Task *t, SimStruct *S) {} /* optional */
6  void outputs_t0(Task *t, SimStruct *S) {} /* optional */
7
8  void initializeSizes(Task *t, SimStruct *S) {
9    if (!ssSetNumInputPorts(S, 1)) return; /* one input */
10   if (!ssSetNumOutputPorts(S, 1)) return; /* one output */
11   ssSetInputPortWidth(S, 0, 1); /* input is 1-dimensional */
12   ssSetOutputPortWidth(S, 0, 1); /* output is 1-dimensional */
13 }
14 void taskFunction(Task *t, SimStruct *S) {
15   real_T in;
16   real_T *out;
17   /* read the input at time 0.0 */
18   in = *((InputRealPtrsType) syncInput(0, t, 0.0))[0];
19   /* perform computations that take 2us (not shown) */
20   out = (real_T *) syncOutput(0, t, 0.000002);
21   *out = 1; /* write the output at time 0.000002 */
22   /* perform computations taking another 1us (not shown) */
23   out = (real_T *) syncOutput(0, t, 0.000001);
24   *out = 2; /* write the output at time 0.000003 */
25 } /* end-of-task at time 0.000003 */
```

3.1.1 Keeping Pace: Variable- vs. Fixed-step Solver

The execution of the task function is slightly ahead of the simulation of the plant. At synchronization points, the execution of the task halts and lets the plant catch up. In order to assure that a particular code segment is executed at the correct time, we must be able to check the context switch condition of synchronization points at the expected times. The simulation engine must therefore execute the block at proper time instants. These times are determined by the Simulink solver, which supports two strategies:

For fixed-step solvers, the simulation engine executes with a constant *step size* and the sample times of all blocks must be integer multiples of it. Since the execution times are given relative to the previous synchronization point and since the execution path in the code is not known in advance, those times cannot

be determined upfront. To obviate missed synchronization times, for a task with period π, offset ϕ, and the set of all execution time parameters $\{t_0,...,t_n\}$ we define the offset of the block to be 0.0 and the sample time to be $gcd(\pi,\phi,t_0,\ldots,t_n)$. The main drawback of this approach is the poor performance because of a small step size that is expected to be typically in the range of microseconds or below. Note that a constant sample time leads to an enormous amount of useless attempts to resume the task if execution times are non-harmonic or vary in order of magnitude. Furthermore, analysis of the code is required to gather all execution times. As an additional restriction, execution times need to be static; they cannot be adapted during the simulation, for example, as a reaction to some global state change.

With a variable-step solver, the step size can vary from step to step. Intuitively, a variable-step solver and Simulink's possibility to set the time for the next execution of a block should allow us to execute the block only when needed, i.e., once for each `sync` statement that is hit during code execution. The next sample hit is defined by the execution times encoded in the parameters of the `sync` calls. However, Simulink's way of operation does not allow such an intuitive implementation. The API call to set the next sample hit (`ssSetTNext`) is only allowed to be called in the callback function `mdlGetTimeOfNextVarHit` while access to output ports, for example, is only allowed during the callback function `mdlOutputs`. According to Figure 1, the callback to set the next sample hit is called first in the simulation loop. Consequently, no next execution times are known at this point. We introduce an intermediate step to be executed after every `sync` statement. A conservative execution time parameter for the intermediate step is a constant value in the order of the machine epsilon. The implementation of this approach is shown in Figure 2 with the required code in orange color. In the worst case, this approach requires twice as many invocations as actually needed. While several optimizations to this strategy are possible, they would require code analysis or a mechanism to turn back simulation time, and we do not expect any significant performance gain.

3.2 Nondirect Feedthrough

Despite their straightforward implementation, a main limitation arising from DF task blocks concerns circular dependencies. Simulink does not allow direct feedthrough blocks to be connected in a circular fashion and reports an algebraic loop error. As a consequence, data dependencies between DF task blocks must form a directed acyclic graph. While delay blocks, such as a Unit Delay block, added between task blocks would break illegal loops, they also distort the timing behavior and cannot be used effectively. A further limitation concerns the influence of data-dependencies on task precedency. In Simulink, a data dependency between direct feedthrough blocks influences the sorted order of blocks as explained in section 1.2. Blocks that provide values have to appear before direct feedthrough blocks that consume those values. This order decides on the precedence of blocks and thus on the precedence of tasks. In order to allow cyclic dependencies between time-annotated control functions and to be able to specify their precedence relation independently of data dependencies, we describe a synchronization mechanism for task blocks with nondirect feedthrough. According to The MathWorks (2016), *blocks without direct feedthrough ports appear at the beginning of the list in no particular order* and there are no restrictions regarding cyclic dependencies. As Simulink implies no particular order of NDF blocks, we may use their priority parameter to dictate their order. This is necessary to simulate the execution of a set of task blocks under a given scheduling scheme (Naderlinger 2017), which is however out of the scope of this paper.

Since no illegal cyclic dependencies between blocks can occur in the presence of NDF blocks, there are no restrictions on the block connections and consequently on the data-dependency relations between the individual time-annotated control functions in the system. However, NDF blocks require a different synchronization strategy, due to the following Simulink restriction: In an NDF block, input values are accessible only in the S-function callback `mdlUpdate` (The MathWorks 2016), while changes to outputs are propagated only in the `mdlOutputs` callback. A detailed discussion on this *function separation requirement* has been presented by Denckla and Mosterman (2004).

This function separation is opposed to a typical controller- or task-function as a single course of action with access to inputs and outputs in arbitrary order. To overcome this mismatch, we present a

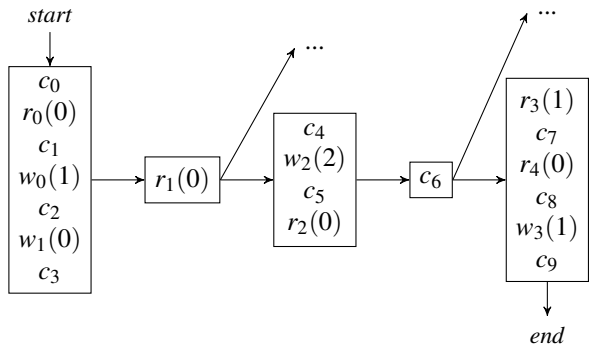
synchronization mechanism between Simulink and NDF task blocks in Listing 2. Changes compared to the DF synchronization mechanism shown in Figure 2 are marked in orange color again.

Listing 2: Nondirect feedthrough synchronization.

```
1  void mdlStart(...) {
2    //create and start thread
3    t_Next = φ;
4  }
5
6  void mdlOutputs(...) {
7    if (t_Sim =ε 0)
8      //write initial values
9    scope = OUTPUT;
10   switchContext();
11 }
12
13 void mdlUpdate(...) {
14   scope = UPDATE;
15   switchContext();
16 }
17
18 void mdlGetTimeOfNextVarHit(...) {
19   if (t_Next >ε t_Sim)
20     ssSetTNext(t_Next);
21   else
22     ssSetTNext(t_Sim + 2ε);
23 }
24
25
26 void mdlTerminate(...) {
27   //cancel thread and cleanup
28 }
29
30 void taskfunction(...) {
31   ...
32   syncRead(t_0);
33   //read inputs
34   ...
35   syncWrite(t_1);
36   //write outputs
37   ...
38 }
39
40 void *taskThreadWrapper(void *arg) {
41   switchContext();
42   while (1) {
43     while (t_Sim ≠ε nextReady())
44       switchContext();
45     t_Code = 0;
46     t_Start = t_Sim;
47     taskfunction();
48     t_Next = nextReady();
49   }
50 }
51 void sync(double t_Sync) {
52   while
53   ((t_Sim − t_Start) <ε (t_Code + t_Sync)) {
54     t_Next = t_Start + t_Code + t_Sync;
55     switchContext();
56   }
57   t_Code = t_Code + t_Sync;
58 }
59
60 void syncRead(double t_Sync) {
61   sync(t_Sync);
62   while (scope ≠ UPDATE)
63     switchContext();
64 }
65
66 void syncWrite(double t_Sync) {
67   sync(t_Sync);
68   if (scope ≠ OUTPUT)
69     //raise run-time error or
70     //introduce ε-timing error
71 }
72
73
```

On the side of the Simulink callback functions (synchronization from the simulation thread), we are required to implement the update function `mdlUpdate` for the block (line 13). Further, a variable `scope` (lines 9, 14) is introduced to distinguish if the simulation thread currently is invoking the output or the update function. After setting the scope, we switch context to the task function thread and wait in both cases. The `switchContext` function is unchanged as well as the wrapper on the task thread side and the `sync` function, which decides when to switch back to the simulation thread and when to proceed with the task execution. In addition to the function `sync`, which is typically inserted after each basic block of the task function during the code instrumentation, we introduce the two functions `syncRead` and `syncWrite`, which are inserted before accessing input values or writing output values, respectively. These functions ensure the correct timing according to the time parameters (t_0 and t_1 in the sample task function) by calling the `sync` function and also that Simulink's requirements *read during update and write during output* are met. As outlined in Figure 1, at every sample hit Simulink first executes the output function and second executes the update function of a block. Therefore, for a read operation, once the simulation time has caught up such that t_{Code} is incremented and the `sync` function returns, we keep switching back to the simulation thread until we are called from the *UPDATE* scope (line 62) at the same simulation time. In contrast, the `syncWrite` function is required to be called within the *OUTPUT* scope. If the `sync` statement in `syncWrite` ever happens to return in the *UPDATE* scope (i.e., the possibility to write the output at this time was already missed) a run-time error will be raised. In essence, an output operation must not occur after an input operation at the same simulation time instant. This needs to be ensured by the execution time instrumentation (i.e., the synchronization instructions). As an alternative to a run-time error, other strategies can be pursued (for example, a small timing deviation can be introduced). Note that synchronization calls with a zero time parameter are allowed, for example, to execute a bunch of read operations at the same time while still following the advice of synchronizing each and every input/output operation. Note that whether or not an input/output operation immediately follows another one cannot be determined statically in general as this depends on the control flow at run-time.

Figure 3a shows the control flow graph of a sample task function for one particular job. Code portions that are not reached in this very run are omitted. We use c_i to indicate code segments without access to input or output variables, $r_j(t)$ represents a read operation from an input, and $w_k(t)$ a write operation to an output variable, where t is the execution time of the preceding code segment ($i, j, k \in \mathbb{N}_0$). Figure 3b shows the sample hits for this job and lists which code portions are executed in which scope. There are several calls with execution time 0 (e.g., to indicate empty code segments) and the total execution time accumulates to 5 µs. For an assumed period of also 5 µs, the code portions c_0, r_0, and c_1 immediately follow c_9. The synchronization mechanism ensures timely execution of read operations in the update phase and write operations in the output phase. All the model's task block input/output operations that are designated to be performed at a particular time instance execute in a strict *write before read* order, which is a result from the output/update function separation.



(a) Control flow graph.

time [µs]	output	update
0.0	c_0	r_0; c_1
$0.0+2\varepsilon$	-	-
1.0	w_0; c_2; w_1; c_3	r_1; c_4
$1.0+2\varepsilon$	-	-
3.0	w_2; c_5	r_2; c_6
$3.0+2\varepsilon$	-	-
4.0	-	r_3; c_7; r_4; c_8
$4.0+2\varepsilon$	-	-
5.0	w_3; c_9; (c_0)	(r_0; c_1)
...	...	...

(b) Simulation times and scopes.

Figure 3: Sample control function.

In the following, we informally argue about algorithmic properties of the mechanism in Listing 2:

Temporal correctness. The precondition for a correct timing in reading inputs and writing outputs as specified by the time parameters is the timely invocation of the block at well-defined time instants. This is ensured by the proper setting of the sample time in case of a fixed sample time or by setting the next sample hit in case of the variable sample time approach as outlined in section 3.1.1.

Simulink conformity. The consideration of the scope in `syncRead` and `syncWrite` ensures that Simulink's requirements regarding access to inputs/outputs are met (function separation). Furthermore, the setting of sample hits is confined to the admissible callback function (i.e., `mdlGetTimeOfNextVarHit`). Assuming a correct time instrumentation, it further follows that simulation time passes between a `syncRead` and the write access to an output in a subsequent `syncWrite` invocation. This ensures that this write access can be performed within some successive `mdlOutputs` callback function.

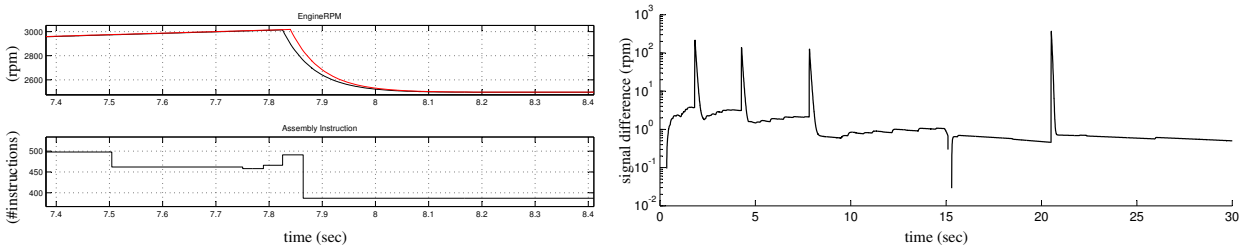
Termination. We consider the loops in the synchronization mechanism to argue that it makes progress and finally terminates for runs with finite simulation times: The while loop (line 43) in function `task-ThreadWrapper` and the while loop (52) in function `sync` exit and resume in the thread execution as soon as the simulation engine has made enough progress. Progress is ensured by setting the next sample hit in `mdlGetTimeOfNextVarHit` (being called at every sample hit) to a time in the future. The adjustments of t_{Next} in (54) and in (3) for starting the first and in (48) for all following executions provide progress greater than an ε-step at every second sample hit. The while loop (62) in function `syncRead` exits as soon as the next invocation of `mdlUpdate` changes the context to $UPDATE$. Finally, at the end of the simulation time `mdlTerminate` cancels the task thread and ends the last loop (42) such that the execution terminates.

Optimality (w.r.t. sample hits). For a fixed sample time approach, the minimal number of sample hits N during a simulation of duration T depends on T, offset ϕ, period π, and the set of all execution time

parameters $\{t_0,...,t_n\}$, to be determined statically: $N = T/gcd(\phi,\pi,t_0,...,t_n)$. For the variable sample time approach, N depends only on the number of executed synchronization calls. In theory, each synchronization call being executed triggers a single follow-up sample hit whose time is determined by the time parameter. However, since this parameter is unknown at the time Simulink allows the next sample hit to be set, one intermediate step is required per synchronization call. If a program analysis has been performed, further optimizations are possible (see section 3.1.1).

3.3 Automatic Transmission Controller Example

The presented approach was applied to the *sldemo_autotrans* example that comes with Simulink. The model contains an automatic transmission controller where the logic is implemented as a Stateflow block. We instrumented the source code generated by the Simulink Coder with synchronization calls that also contained information on the code execution complexity in terms of *number of assembly instructions per basic block*. The number of executed instructions per task job ranged from 385 to 529 for the default maneuvers (and 27 at start-up). The values for a short time frame of about one second of simulation time are shown in the lower part of Figure 4a. For the simulation the values were multiplied by a constant time factor such that the maximum observed execution time of the controller consumed about 50% of its 4 *ms* period. A close up view of the comparison of controller output signal EngineRPM is shown in the upper part of Figure 4a: the black signal shows the original Stateflow signal, and the red signal shows the signal when considering the varying software-induced delays. Figure 4b shows the modified behavior displayed as the difference of the EngineRPM signal between the original controller and the one that considers execution time. The maximum difference (386 rpm) on the logarithmic y-scale is about 8.5% of the original signal at time 20.52. Results of simulating the transmission controller as one of three tasks under preemptive scheduling using the presented approach are described in (Naderlinger 2017).



(a) EngineRPM signal and assembly instructions.

(b) EngineRPM signal comparison.

Figure 4: Automatic Transmission Controller.

4 CONCLUSION

We presented a modification of the classical synchronous reactive behavior in synchronous block diagrams. Its basic principle allows us to consider jitter effects of control tasks caused, for example, by execution time variations or start-time delays already in the simulation. The approach is capable of considering also scheduling and preemption aspects. To this end, we discussed the idea of a *time-annotated control function* implemented as *task block* whose activation-, start-, and end-time instances do not necessarily coincide. A synchronization mechanism for such a block was implemented for MATLAB/Simulink.

REFERENCES

Badreddine, B., and D. Hoadley. 2010. “Modeling & Code Generation for Powertrain Control Monitoring”. In *Society of Automotive Engineers (SAE)*. Warrendale, Penn.

Biondi, A., M. Di Natale, and G. Buttazzo. 2015. "Response-time Analysis for Real-time Tasks in Engine Control Applications". In *Proceedings of the ACM/IEEE Sixth International Conference on Cyber-Physical Systems*, ICCPS '15, 120–129. New York, NY, USA: ACM.

Bouchhima, F., M. Briere, G. Nicolescu, M. Abid, and E. Aboulhamid. 2006, Sept. "A SystemC/Simulink Co-Simulation Framework for Continuous/Discrete-Events Simulation". In *Behavioral Modeling and Simulation Workshop, Proceedings of the 2006 IEEE International*, 1–6.

Cervin, A., B. Lincoln, J. Eker, K.-E. Arzen, and G. Buttazzo. 2004. "The Jitter Margin and its Application in the Design of Realtime Control Systems". In *In Proceedings of the IEEE Conference on Real-lime and Embedded Computing Systems and Applications*.

Craciunas, S. S., C. M. Kirsch, H. Payer, H. Röck, and A. Sokolova. 2013. "Temporal Isolation in Real-Time Systems: the VBS Approach". *International Journal on Software Tools for Technology Transfer* 15 (3).

Cremona, F., M. Morelli, and M. Di Natale. 2015. "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. New York, NY, USA: ACM.

Denckla, B., and P. J. Mosterman. 2004. "An Intermediate Representation and Its Application to the Analysis of Block Diagram Execution". In *Proceedings of the 2004 Summer Computer Simulation Conference*.

Edwards, S. A., and E. A. Lee. 2003, July. "The Semantics and Execution of a Synchronous Block-diagram Language". *Sci. Comput. Program.* 48 (1): 21–42.

Henriksson, D., A. Cervin, and K.-E. Arzen. 2003. "TrueTime: Real-time Control System Simulation with MATLAB/ Simulink". In *Nordic MATLAB Conf.*

IEEE Std 1003.1c-1995 1995. "POSIX.1c, Threads extensions".

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

Knuth, D. E. 1997. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc.

Lee, E. A., and S. A. Seshia. 2010. *Introduction to Embedded Systems - A Cyber-Physical Systems Approach*. Lee and Seshia.

Lublinerman, R., and S. Tripakis. 2008. "Translating Data Flow to Synchronous Block Diagrams". In *2008 IEEE/ACM/IFIP Workshop on Embedded Systems for Real-Time Multimedia*.

Marti, P., J. Fuertes, G. Fohler, and K. Ramamritham. 2001, Dec. "Jitter Compensation for Real-Time Control Systems". In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS 2001)*, 39–48.

Meyerowitz, T., A. Sangiovanni-Vincentelli, M. Sauermann, and D. Langen. 2008. "Source-level Timing Annotation and Simulation for a Heterogeneous Multiprocessor". In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '08, 276–279. New York, NY, USA: ACM.

Naderlinger, A. 2017, March. "Simulating Preemptive Scheduling with Timing-aware Blocks in Simulink". In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, 758–763.

Resmerita, S., P. Derler, W. Pree, and K. Butts. 2012. *Real-time Simulation Technologies: Principles, Methodologies, and Applications*, Chapter The Validator Tool Suite: Filling the Gap between Conventional Software-in-the-loop and Hardware-in-the-loop Simulation Environments. CRC Pr. Int.

The MathWorks 2016. *Simulink Reference, R2016a*.

Velasco, M., P. Marti, R. Villa, J. M. Fuertes, J. Ayza, and M. Monroig. 2005. "A Probabilistic Approach to the Stability Analysis of Real-Time Control Systems". In *16th IFAC World Congress*.

AUTHOR BIOGRAPHIES

ANDREAS NADERLINGER is an Assistant Professor of the Department of Computer Sciences at the University of Salzburg, Austria. He holds a Ph.D. in computer science from the University of Salzburg. His research interests include real-time and embedded systems, modeling and simulation, and the logical execution time paradigm. His e-mail address is andreas.naderlinger@cs.uni-salzburg.at.