*Proceedings of the 2019 Winter Simulation Conference*
*N. Mustafee, K.-H.G. Bae, S. Lazarova-Molnar, M. Rabe, C. Szabo, P. Haas, and Y.-J. Son, eds.*

# HARNESSING CONCURRENCY IN SYNCHRONOUS BLOCK DIAGRAMS TO PARALLELIZE SIMULATION ON MULTI-CORE HOSTS

Andreas Naderlinger

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

## ABSTRACT

Model-based and simulation-supported engineering based on the formalism of synchronous block diagrams is among the best practices in software development for embedded and real-time systems. As the complexity of such models and the associated computational demands for their simulation steadily increase, efficient execution strategies are needed. Although there is an inherent concurrency in most models, tools are not always capable of taking advantage of multi-core architectures of simulation host computers to simulate blocks in parallel. In this paper, we outline the conceptual obstacles in general and discuss them specifically for the widely used simulation environment Simulink. We present an execution mechanism that harnesses multi-core hosts for accelerating individual simulation runs through parallelization. The approach is based on a model transformation. It does not require any changes in the simulation engine, but introduces minimal data propagation delays in the simulated signal chains. We demonstrate its applicability in an automotive case study.

## 1 INTRODUCTION

The development of embedded and real-time systems is a challenging and complex engineering task. Today, the prevalent development approach is model-based. The ability to simulate the behavior of the individual control tasks together with a model of the plant on a desktop PC leads to a short development cycle and reduced costs. Code generators turn the modeled controller functionality into production quality code. MATLAB/Simulink is a widely used tool for the development of safety-critical embedded control systems across multiple domains. Supplementary toolboxes provide additional functionalities for individual use cases. The *Computer Vision and System Toolbox*, for example, supports i.a. camera calibration, object detection/tracking, motion estimation, or various video processing techniques, which makes it particularly attractive for the development of advanced driver assistance systems (ADAS).

A main advantage of the model-driven approach is its ability to immediately test the control algorithms on a host PC without requiring the target platform (including the hardware, real-time operating system, etc.) or the actual system that needs to be controlled, such as a vehicle. Clearly, the more fine-grained and realistic the plant has been modeled the more accurate the simulated behavior will be. Typical test strategies start with a model in the loop (MIL) testing where the plant is simulated with the control algorithm represented as numerous Simulink blocks. In a next step, the control part of the model is code generated and this very C code is brought back to the model as a single or a few heavyweight block(s). This testing method, known as software in the loop (SIL) testing, is an important step before deploying the system to a hardware in the loop (HIL) testbed or to the actual target platform. Additionally, simulating a SIL model tends to be faster than simulating the original MIL model. On this level, integration tests that combine multiple such controllers in a single model ensure the correct behavior of greater parts of the system. This

978-1-7281-3283-9/19/$31.00 ©2019 IEEE 

is where the control engineer can easily configure and adapt a particular system on the desktop computer and observe the resulting impact on the whole ensemble. The simulation of such a model poses high computational demands on the simulation environment. Unfortunately, to the best of our knowledge, the principal block execution strategy of a Simulink model is strictly sequential and consequently bound to a single thread of execution (see the related work section for some exceptions). This means that a recent multi-core host PC, for example, with 4 cores each capable of two independent threads is far from realizing its full potential, although there is an inherent concurrency in most of the models. Note that individual GUI elements, such as a Scope or Video Viewer block, are already able to run on separate cores.

In this work, we outline the fundamental obstacle for the simulation engine to parallelize the execution of a single simulation run. We present an approach that harnesses the concurrency present in synchronous block diagrams to simulate individual parts of the model in parallel. In contrast to previous work, our mechanism allows us to directly use the Simulink simulation engine itself to simulate the blocks instead of executing the generated code in a customized simulator. The remainder of this work is structured as follows: We continue with a discussion on related work and some Simulink fundamentals. Section 2 describes the core concept of our approach together with the model transformation process that enables concurrent and consequently parallel block execution by partly shifting the execution strategy from the simulation engine to the block-level. In Section 3 we exemplify different parallelization scenarios. In Section 4 a case study of an ADAS system is presented before future work is discussed and the paper is concluded.

### 1.1 Related Work

To date, a large amount of scientific papers has dealt with the parallelization of model-based descriptions, such as Simulink models. However, most of them covered the process of software synthesis from models so that the resulting source code can be efficiently and correctly executed by a real-time operating system on a (mostly embedded) multi-core target. Efforts in this direction include, for example, (Canedo and Faruque 2012), (Tuncali et al. 2015), (Cha et al. 2011), and (Kumura et al. 2012). They discussed block-to-task/core mapping and scheduling as well as buffering protocols. Even for single-core targets, the preservation of the original semantics of synchronous block diagram models, such as Simulink, has shown to be non-trivial when multi-rate models are mapped to software tasks (see Baruah 2012; Caspi et al. 2008). Canedo et al. (2010) described an automatic splitting mechanism of Simulink models, where individual parts are code generated and executed in a custom simulation environment. According to Görür and Çalli (2017) who presented a transformation-based approach for parallelizing Scilab/Xcos models, the first approach for parallelizing Simulink models was proposed almost 20 years ago by Ozard and Desira (2000). They distributed a model on a multi-processor system under the restriction that the parallelized blocks are identical and don't interact except at one time step. In a very recent MATLAB release (version 2018b), the *DSP System Toolbox*, a Simulink add-on for signal processing systems, supports parallel execution of blocks in the *Dataflow domain* (a new model of computation in Simulink). Also, with some limitations, the same version enables the distribution of co-simulation components (so-called Functional Mock-up Unit (FMU) blocks) on different cores. Another recent FMU-based approach for parallelizing multiple different simulators is described in (Saidi et al. 2019).

To the best of our knowledge, the presented approach is the first work that reconciles parallel computations with the general sequential block execution in synchronous block diagrams and that enables the parallel execution of individual blocks in Simulink simulations. It has been tested with various MATLAB versions back to R2014a and is expected to be compatible with even older releases.

### 1.2 Background

Simulink is an add-on to MATLAB and provides a modeling and simulation environment for dynamic systems. Being based on the causal block diagram formalism (Denckla et al. 2005), Simulink is closely related to the synchronous reactive programming model (Halbwachs 1993). A Simulink model is a block

diagram that consists of blocks implementing the functionality. Data-dependency relations between blocks are described by signals that connect input with output ports of (receiver) and (supplier) blocks, respectively. Input ports are either direct-feedthrough (DF) or nondirect-feedthrough (NDF). While DF input ports can instantly influence the output of the block, NDF input ports can only affect the block's state. Essentially, a block with solely NDF input ports represents a Moore machine, while a block that has at least one DF input port represents a Mealy machine.

During initialization, the simulation engine derives a fixed order, called the *block sorted (execution) order*, which is based on the data-dependencies between blocks as expressed by signals and the feedthrough characteristics of the ports they input. In this very order, the blocks are executed during the subsequent simulation loop, in which the simulation engine repeatedly updates the state of the model by computing the system inputs, outputs, and states as time progresses. A solver determines the time steps between the 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). A detailed description of the operational semantics of Simulink's simulation engine is given in (Bouissou and Chapoutot 2012). Conceptually, at each step all the blocks in the model execute simultaneously and instantaneously, i.e., all operations and computations are performed in zero (simulation) time. Effectively, the simulation engine interacts with the blocks by an API that, amongst others, comprises two important callback functions: the *output function* calculates and sets the block's output signals, whereas the *update function* calculates and sets the block's state variables. While a block may access all of its input signals in the update function to update its state, it is only allowed to access DF input ports in the output function.

## 2 TOWARDS PARALLELIZATION OF SIMULINK MODELS

The ultimate goal of this work is to speed up simulations by executing individual parts of a model in parallel. We hereby consider a single simulation run. This is in contrast to known approaches, where an overall acceleration is enabled by running multiple simulations in parallel (e.g., using the `parsim` command of the Parallel Computing Toolbox).

We start this section with a general discussion on the principal presence of concurrency in causal block diagrams, such as Simulink models. We contrast this to the strict sequential execution mechanism of Simulink's simulation engine, which cannot turn the conceptual concurrency into a parallel execution due to its operational semantics. To still support concurrent and ultimately parallel execution, we propose a modified execution strategy on a block-implementation level, while the execution strategy of the simulation engine itself stays unchanged. To avoid creating false expectations, such a parallelization is not possible without changing individual signal traces, by introducing minimal data propagation delays in simulated signal chains. However, as we will argue below, these changes likely only have minor implications on the simulated behavior or might not even be observable on the system output level.

### 2.1 General Remarks and Potential Assessment

Support for parallelism typically introduces some sort of run-time overhead. Depending on the specific implementation, this overhead can be substantial. In the case of blocks performing computationally lightweight operations, parallelizing their execution likely results in a degradation of the overall performance. However, in the case of heavyweight blocks, a parallelization can be particularly useful. Typically, this is the case for software in the loop (SIL) simulations, where all the individual blocks belonging to a single functional unit are turned into a code-representation, which is then executed as a single block. Finding a reasonable granularity for parallelization is thus a crucial prerequisite for the presented approach.

Parallelization is only one of the possible approaches to improve performance. The Simulink User's Guide lists various hints and techniques that may help to speed up simulation such as switching solver type, choosing sample rates that are multiples of each other, or lowering accuracy requirements. This list should be the first reference point to identify potential flaws in the model and for improving simulation

performance. However, especially large models with multiple compute-intensive algorithms, such as video processing or radar cross section analysis, are ideal for a parallelized simulation.

In view of this, data-dependencies are a further important factor that decide on the potential of parallelization and thus on the potential acceleration that can be achieved for an individual simulation run. In addition to forward dependencies, Simulink signals may also represent backward dependencies. In the case of backward dependencies, a receiver block depends on computations performed in the last iteration of a supplier block, e.g., in a feedback (closed-loop) control system. Typically, only forward dependencies limit the ability to parallelize the execution of a model. Besides the data-dependency aspect, which we address in Section 3.1, we will show that further factors (e.g., block periods, offsets, or the order in which blocks are executed) have a fundamental impact on the potential acceleration in Section 3. The latter is mainly due to the operational semantics of the simulation engine.

## 2.2 Concurrency in Causal Block Diagrams

Most causal block diagrams exhibit concurrent aspects. We use the term *concurrent* to express the fact that blocks can be executed in any order at each simulation step without influencing the behavior. This is the case, for example, for the set of blocks with solely nondirect-feedthrough, i.e., the blocks of type Moore: *"Blocks without direct-feedthrough ports appear [...] in no particular order"* (MathWorks 2018). Furthermore, two blocks without data-dependencies can also execute in one order or the other.

The *level of concurrency* is a property that is inherent in the model. Figure 1 shows three sample block diagrams together with their respective dependency graphs. We assume that blocks $A, B, b_0, \dots b_3$ are of type Mealy, i.e., their output depends on their state and current input (direct-feedthrough). The block $1/z$ in case (c) represents a unit delay and thus its output does only depend on the state but not on its current input (type Moore; nondirect-feedthrough). In case (a) there is only one valid order of executing the blocks and hence no concurrency. In case (b) the signal paths involving blocks $A$ and $B$ are completely independent. While there is again only one valid order for each path, both strands can be executed in any order; one after the other or intertwined, i.e., concurrent. Due to the delay block in case (c), in every iteration block $B$ reads the output of block $A$ from the last iteration. Therefore scenario (c) breaks the data-dependency allowing a concurrent execution of the two strands $(b_0, A)$ and $(1/z, B, b_1)$. Note that blocks $A$ and $B$ may be compound blocks (e.g., virtual subsystems) and consist of direct-feedthrough blocks $A1, A2, \dots$ and $B1, B2, \dots$, respectively. Thus, forward dependencies and feedthrough characteristics decide on the level of concurrency.

Beyond that, the *level of parallelism* is a matter of implementation and depends on the environment, in which MATLAB is executed and which involves the hardware as well as the software configuration (e.g., it is limited by the number of cores and is influenced by the thread/core affinity setting). Clearly the level of parallelism is furthermore bounded by the level of concurrency.

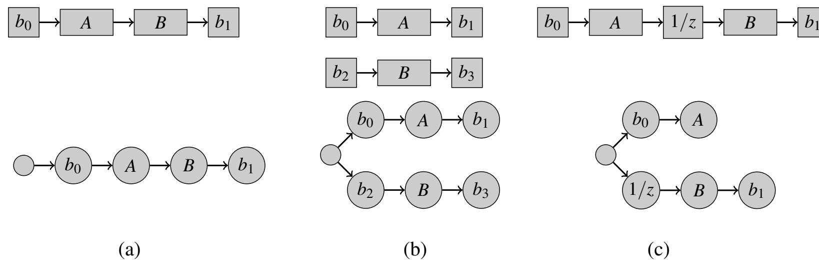


Figure 1: Block diagrams vs. dependency graphs.

### 2.2.1 Concurrency among Blocks vs. Sequential Simulation Engine

Although in principle some blocks could be executed *concurrently*, in general the simulation engine of Simulink executes all blocks within a model purely *sequentially*. As outlined in Section 1.2, at consecutive points in time (called *simulation steps*) during the simulation loop, the simulation engine computes and updates the model's state. For this, the engine prompts all blocks, whose sample time parameter is an integer fraction of the current step, to compute their outputs and to update their internal state. This is done by invoking both the *output* and the (optional) *update callback function* of the block. These calls are performed both *synchronously*, i.e., the engine waits for the calls to return, and *sequentially* in a fixed order derived during the initialization phase of the simulation. After finishing the last relevant block, the simulation time is increased to the next step. Consequently, with respect to the simulation time, all operations for a particular step are performed instantaneously (consuming zero simulation-time). This is a perfectly fine implementation of the SR model, however, depending on the computational costs associated to the execution of a block, it may not be the most efficient one.

## 2.3 Enabling Parallelism: The Core Idea

To overcome this sequential execution and introduce true parallelism for the block execution, we propose a novel concept to execute blocks in Simulink. An important aspect to note here is that no adjustments are made to the simulation engine. Modifications are only made on the block level. In the following we assume a scenario of compute-intensive direct-feedthrough blocks, each implementing its functionality in a single top-level procedure, called *step function*. The respective step function is invoked, whenever the simulation engine triggers the respective block via the *output* function of the block API. Therefore, it may in principle perform operations to read inputs, write outputs, and adjust the state in an arbitrary order, with no *update* function required. However, in the following we assume that the step functions follow a strict read-execute-write (REW) schema (International Electrotechnical Commission 2013), i.e., all inputs are read at the beginning of the function while all outputs are written at the end. In between, there is no access to inputs or outputs. In the subsequent Section 2.3.1, we argue that in a system with SR semantics this is not a restriction. We denote the three code sections of a block as the sequence of the operations $(r,e,w)$.

In our approach, when the simulation engine invokes the step function of a block, the sequence of operations $(r,e,w)$ is not executed atomically. Instead we separate the time for reading inputs ($r$) from the time for writing outputs ($w$). We denote these two simulation time instances as $t_r$ and $t_w = t_r + \Delta$, respectively. The time delay $\Delta$ can be decided for each block individually in the range $(0,T]$, where $T$ is the sample time of the block. Within the interval $[t_r,t_w]$, the compute-intensive operation $e$ is executed asynchronously, which gives us the opportunity to execute the $e$ operation of multiple different blocks in parallel. This altered execution strategy essentially has two effects. On the one hand, the wall clock time to complete a simulation can be considerably reduced. On the other hand, delaying of the write operation shifts individual signal traces and can change the overall system behavior. Figure 2 contrasts the sequential (a) and the concurrent (b) execution with respect to simulation time (st) and wall clock time (wct).

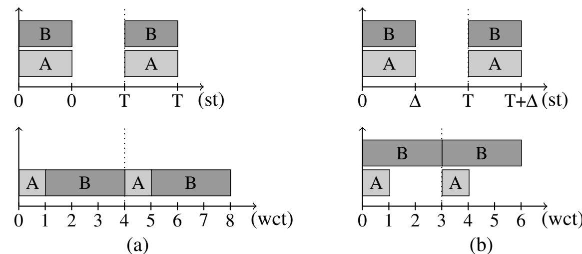


Figure 2: (a) Sequential execution, (b) concurrent execution with delayed output.

It is important to note, that it is not possible to execute operations of multiple blocks concurrently without introducing the time delay for two reasons: Firstly, the $r$ and $w$ operations must be synchronized with the simulation engine to ensure deterministic simulation results. Secondly, the step function of each block is executed at most once for each time step. Therefore, we cannot initiate the $e$ operation for all blocks in a first round and wait for its completion in a second round before writing the outputs at the same simulation time.

However, delays are a natural modeling element in synchronous block diagrams, for example, to avoid illegal cyclic dependencies (algebraic loops), or to mimic the expected execution times for executing the functionality on the target hardware (Naderlinger 2017a). Furthermore, controllers are typically designed to tolerate delays and jitter. Therefore, we argue that the resulting delays are likely to be compensable, when considered properly. It is important to note, that the execution strategy of the blocks in the simulation is independent of the scheduling of the code on the target platform (see Section 1.1).

### 2.3.1 Read-Execute-Write (REW) Cycles and Logical Execution Time (LET)

The presented approach is particularly useful for the *read-execute-write (REW)* task model, as typically found in programmable logic controller (PLC) applications (International Electrotechnical Commission 2013). Reading all inputs at the start of the execution and writing all outputs at the end maximizes the time frame for input/output-independent execution and thus the workload that can be parallelized. The AUTOSAR standard (AUTOSAR 2019), for example, also makes use of this model (calling it *implicit* communication) in order to increase determinism. Notably, the REW model is not a requirement. However, only a single contiguous code segment without access to inputs and outputs can be executed asynchronously for each block. This motivates an automated source-code transformation to REW-form before applying our parallelization approach. Within a zero-execution-time environment such as Simulink such a transformation can be applied without implications on the behavior of the simulation. Due to the synchrony hypothesis of the underlying SR semantics, any function can be rewritten to adhere to REW-form by first converting its code into static single assignment (SSA) form (Rosen et al. 1988). Therefore, the above assumption is not a restriction but comes with the benefit of simplifying presentation. More importantly, it also maximizes the code portion that will be subjected to concurrent execution, as will be shown below.

The logical execution time (LET) model (Henzinger et al. 2003) pushes this separation of computations and input/output operations even further. The code structure follows the REW-form. Additionally, as the LET model associates each task with a constant (logical) execution time, the delayed output of the proposed approach is a perfect fit. By defining the delay of all blocks to be equal to the LET of their associated computational unit, the parallelization of the model will not lead to a modified behavior.

## 2.4 Formal Description of the Model Transformation

Our approach is based on a model transformation as described below. During simulation, the described semantics of the transformed system are established by a synchronization mechanism that ensures both the compliance with the requirements of the simulation engine and the correct order of operations.

Let $E$ be a simulation environment subsuming a certain hardware/software setup. This environment hosts a simulation engine, which follows the synchronous reactive (SR) paradigm, together with a solver. A *sequential simulation configuration* $S^s$ is described by the tuple $(B^s, P, m^s, L, \tau^s_\Omega, \sigma^s, E)$, where $B^s$ is a set of blocks, $\sigma^s$ is a total order relation on $B^s$, $P$ is a set of ports, $L$ is a set of links (connections between ports), and $\tau^s_\Omega \in \mathbb{R} \cup \infty$ is the simulation end time. Furthermore, the function $m^s : P \to B^s$ defines a mapping between ports and blocks. For all blocks $b_i \in B^s$, let $(t_i^k)_{k=0,1,...} \in \mathbb{R}^+ \cup \{0\}$ be the monotone sequence of (simulation) time instances, called *sample hits*, where the simulation environment triggers $b_i$ by executing its step function $sf_i$, which is a function in the computer programming sense (procedure). For $b_i$ being a periodically triggered block, $t_i^k = o_i + k \cdot T_i$, where $o_i$ is the offset and $T_i$ the sample time of $b_i$. With $\sigma_i^s$ we denote the index of $b_i$ in the sorted block order established by $\sigma^s(b_i)$.

The sample hits of the simulation configuration $S^s$ is the set of sample hits of all blocks in $B^s$ in the simulation interval $[0, \tau_\Omega^s]$. At every such sample hit $t$, the simulation engine executes the step function of every block $b_i \in B^s$ where $t \in (t_i^k)_{k=0,1,\ldots}$. It does so purely sequentially, in the order defined by $\sigma^s$. The block's step function $sf_i$ implements an arbitrary long but finite sequence of operations $(op_j^i)_{j=1,2,\ldots}$ with $op_j^i \in \{r,e,w\}, \forall j \in \mathbb{N}$. At this, $r$ and $w$ represent an operation to *read* an input signal and *write* an output signal, respectively. The symbol $e$ refers to an *execute* operation (or computation) that does not involve access to input or output signals.

We define $B^{s'} \subseteq B^s$ to be the set of blocks which are subjected to a transformation process described below to enable a potentially parallel execution of their functionality. As outlined in Section 2.3.1, without loss of generality, we can assume that each execution $k$ of the step function of a block $b_i$ in $B^{s'}$ results in a strict sequence of the three sequentially executed operations $(r_i^k, e_i^k, w_i^k)$, where $r_i^k$, $e_i^k$, and $w_i^k$ subsume all read, execute, and write operations in $(op_j^i)_{j=1,2,\ldots}$, respectively.

We introduce a work load indicator $w$ for every operation to be performed at a particular simulation time. This work load represents the wall clock time that is required to execute a particular operation at a specific simulation time instant. This is in contrast to the (simulation) time cost of executing an operation, which – as a result of the synchrony hypothesis inherent to the simulation engine – is always zero. For example, $w(sf_i, t_i^k)$ is the real time cost to atomically execute $sf_i$ at simulation time $t_i^k$ (i.e., without interruption) on the environment $E$. At this, we assume that the workload of read and write operations is zero, thus $w$ corresponds only to the execute operation of the block's step function at a particular time instant.

We define a transformation process $\mathcal{P}(S^s, p_{sc}) = (B^c, P, m^c, L, \tau_\Omega^c, \sigma^c, E)$, which establishes a *concurrent simulation configuration* denoted as $S^c$. The set of blocks $B^c = B^s \setminus B^{s'} \cup B^{c'}$, where $B^{c'} = p_{sc}(B^{s'})$. The transformation function $p_{sc} : B^{s'} \to B^{c'}$ defines a one-to-one mapping of sequentially executed blocks $\{b_1, \ldots, b_n\} \in B^{s'}$ to concurrently executed blocks $\{c_1, \ldots, c_n\} \in B^{c'}$, which are totally ordered by $\sigma^c : B^c \to \mathbb{N}$. More precisely, it are the execute operations of the blocks in $B^{c'}$ that can be executed concurrently and, as a further consequence, in parallel.

Since $p_{sc}$ defines a bijection, and since the set of ports $P$ and the set of links $L$ are taken over from the sequential configuration, the structural representation of $S^c$ is identical to $S^s$. The mapping from ports to blocks is defined by the function $m^c : P \to B^c$ such that $\forall p \in P, b \in B^s, c \in B^c : m^c(p) = c$, iff $c = p_{sc}(b)$ and $b = m^s(p)$.

As a prerequisite for concurrent execution, let $\Delta(c_i, k) : (B^{c'} \times \mathbb{N}) \to \mathbb{R}^+$ associate a positive real number to the k$^{th}$ execution ($k \in \mathbb{N}$) of a block $c_i \in B^{c'}$, which represents a time delay (in the domain of the simulation time) with the following effect: For $b \in B^{s'}$ and $c = p_{sc}(b) \in B^{c'}$, if $(t^1, t^2, \ldots)$ is the sequence of sample hits of block $b$, the sequence of sample hits of block $c$ is given by $(t^k, (t^k + \Delta(c,k)))_{k=1,2,\ldots}$ with $\Delta(c,k) \leq (t^{k+1} - t^k), \forall k \in \mathbb{N}$. We assume in the following that the delay associated to a block is constant over all executions $k$, and denote the delay of a block $c_i$ with $\Delta_i$, such that for a periodically triggered block with period $T_i$ and offset $o_i$, the sequence of sample hits is $(o_i, (o_i + \Delta_i), (o_i + T_i), (o_i + T_i + \Delta_i), (o_i + 2 \cdot T_i), (o_i + 2 \cdot T_i + \Delta_i), \ldots)$.

At time instances $(t^k)_{k=1,2,\ldots}$ the step function executes the operations $(r^k \tilde{e}^k)$, while at time instances $(t^k + \Delta)_{k=1,2,\ldots}$ the step function executes the operations $(\gamma(e^k) w^k)$. With $\tilde{e}$ we denote the asynchronous execution of operation $e$. The operation $\gamma(e)$ denotes a synchronization operation that waits for $e$ to complete. As a consequence, for the periodic case with $\Delta_i = T_i$, the sequence of operations of block $c_i$ at time instance $t^1$ is $(r_i^1 \tilde{e}_i^1)$ and is $(\gamma(e_i^{k-1}) w_i^{k-1} r_i^k \tilde{e}_i^k)$ for all following time instances $t^2, t^3, \ldots$.

For all periodic blocks $b_i \in B^{s'}$, let $\alpha_i = |(t_i^k)| = (\left\lfloor \frac{\tau_\Omega^s - o_i}{T_i} \right\rfloor + 1)$ denote the number of triggerings of $b_i$ in $S^s$ in the interval $[0, \tau_\Omega^s]$. In the concurrent case additional sample hits are required to execute the write operations for all blocks with $\Delta \neq T$: let $c_j = p_{sc}(b_i)$, then $|(t_j^k)| = \alpha_i \cdot (\left\lfloor \frac{\Delta_j}{T_j} \right\rfloor + 1)$. Therefore, to achieve the same number of complete step function executions in $S^s$ and $S^c$, the simulation stop time needs to be extended such that $\tau_\Omega^c = \tau_\Omega^s + max(\Delta_i)_{i=1,\ldots,|B^{c'}|}$. Additionally, no step function in $S^c$ may start execution after $\tau_\Omega^s$.

Starting from this transformation process and the proper choice of the delay parameters for the individual blocks, further refinements of $S^c$ may involve the adjustment of offsets, periods and the sorted block order defined through $\sigma^c$. These parameters have a substantial effect on the parallelization potential and the behavioral differences between $S^s$ and $S^c$, as illustrated in Section 3.

### 2.5 Implementation and Resulting Wall Clock Timing

We have implemented an execution mechanism for concurrent configurations as described above for MATLAB/Simulink based on S-Functions (MathWorks 2018). Thereby, each transformed block executes the step function in its own thread of execution (based on pthreads). The assignment of threads to cores is decided by the host operating system. Complementarily, a synchronization mechanism ensures the compliance with the requirements of the simulation engine and the order of operations.

For the remainder of this section as well as for the illustrative scenarios in Section 3, we assume the *level of parallelization* to be equal to the *level of concurrency* (which presupposes that the number of cores is greater or equal than the number of blocks). Furthermore, we assume the workload of each individual block $c_i$ to be constant during all its invocations and denote it with $\mathcal{w}_i$. Note that these are not real requirements. Also, we ignore any overhead from the simulation engine or the synchronization mechanism. Thus, the wall clock time definitions below actually define lower bounds.

Let $(t_i^k)_{k=1,2,\ldots}$ be the complete sequence of sample hits of block $c_i \in B^{c'}$. The wall clock time *wct* to execute a read operation $r$ and a write operation $w$ for a simulation time $t_i^k$ is defined as follows:

$$wct_i(op,t_i^k) = \begin{cases} wct_i^{max}(t_i^k) & \text{if } op = r \\ max(wct_i(r,t_i^k - \Delta_i) + \mathcal{w}_i, wct_i^{max}(t_i^k)) & \text{if } op = w \end{cases}, \text{ where}$$

$wct_i^{max}(t_i^k) = max(0, wct_h(op,t_h^l), wct_j(op,t_j^l))$ for $op \in \{r,w\}, l \in \mathbb{N}, \sigma_h < \sigma_i < \sigma_j$, where $t_h^l \leq t_i^k$, and $t_j^l < t_i^k, \forall h,j : c_h, c_j \in B^{c'}$.

Thus, the wall clock time, when a particular operation at particular simulation time is performed, is a result of the concurrent configuration including periods, offsets, delays, and work loads on a given hardware/software environment together with the sorted block order.

## 3 PARALLELIZATION SCENARIOS

This section demonstrates the potential of concurrent execution that is provided by our approach. Therefore, we assume (only for illustration) that the number of blocks is less than the number of cores to execute their functionality. Still there are configurations that impede parallelization. We show subtle effects resulting from supposedly minor changes in the configuration and therefore indicate pitfalls that come with our approach which result however from the restrictions of the simulation engine. In the following, we exemplify the approach with different scenarios for parallelization that are motivated by different use cases.

Unless noted otherwise, we assume that the considered blocks $\{b_1, b_2, \ldots\}$ are independent, i.e., they have no data dependencies. Furthermore, we assume that they are ordered by their index in the sorted block order, i.e., $\forall b_i, \sigma(b_i) = i$. To shorten the descriptions, we omit the index for the parameter representing the sample time, offset etc., if the value is the same for all but explicitly mentioned blocks.

We consider in each scenario the sequential simulation configuration in the half-open interval $[0, \tau_\Omega^s)$, i.e., there are no block executions at the final simulation time step. To be fair in the performance comparison, we choose an adequate simulation stop time in the concurrent simulation configurations and consider the equal number of block triggerings.

We contrast the wall clock times $\mathcal{W}^s$ and $\mathcal{W}^c$ it takes to simulate the listed simulation configurations in the sequential and the concurrent case, respectively. Thereby we only consider the computational work load of the blocks and assume the overhead of the simulation environment and the synchronization mechanism

to be zero. In the sequential case, the wall clock time to simulate the system in the interval $[0, \tau^s_\Omega)$ is given by $\mathcal{W}^s = \sum_i(\left\lceil \frac{\tau^s_\Omega - o_i}{T_i} \right\rceil \cdot w_i)$.

We use the constant $\varepsilon$ to represent a small constant (e.g., twice the machine epsilon), to minimize the behavioral difference due to the delayed write operations of the output signals. Furthermore, we use the constant $r$ as an additional factor for the work loads, in order to indicate the different time domains: simulation time (st) and wall clock time (wct).

**Scenario 1.** This scenario describes the case, where the blocks under consideration have identical execution patterns, i.e., they share the same sample time $T$, offset $o$, and delay $\Delta$ parameters. Only the execution times $w$ may differ.

Let's assume, $\tau^s_\Omega = T = 1, o = 0, \Delta = \varepsilon$ for all blocks $\{b_1, \ldots, b_5\}$. Furthermore, $w_1 = 1r, w_2 = 1r, w_3 = 2r, w_4 = 4r, w_5 = 6r$. $\mathcal{W}^s = \sum_i w_i = 14r$ (Figure 3a). In the concurrent case, all blocks may start to execute at time 0 and finish at time $\Delta = \varepsilon$. Thus, $\mathcal{W}^c = max_i(w_i) = 6r$ (Figure 3b,c). Please note, that the wall clock time, when an individual block $b_i$ writes its output (indicated by $\diamond$), depends on the sorted block order and is equal to $max_{j \leq i}(w_j)$.

Please further note that when changing, for example, the offset $o_4$ to 0.5, $\mathcal{W}^c$ will increase to $10r$ (Figure 3d). This is due to the fact that $b_4$ cannot start execution at time $o_4$ until all blocks have written their outputs at time $\Delta$, which is only after $6r$, when $b_5$ has finished. Consequently, $b_4$ finishes at $10r$, which means after additional $4r$ compared to the previous concurrent configuration.

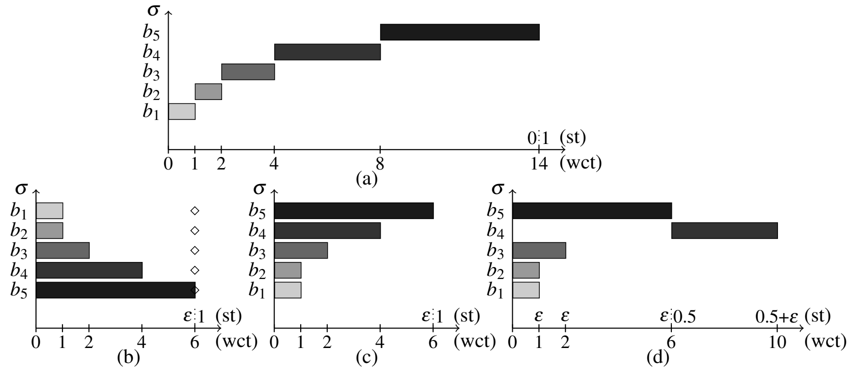


Figure 3: Scenario 1.

**Scenario 2.** In this scenario, all blocks under consideration have harmonic periods and share the same work load parameter. For example, $o = 0, \Delta = \varepsilon, w = 1r$ for all blocks $\{b_1, \ldots, b_5\}$, $T_1 = 2, T_2 = 4, T_3 = 8, T_4 = 16, T_5 = 32$, and $\tau^s_\Omega = 32$ results in $\mathcal{W}^s = \sum_i(\left\lceil \frac{32}{T_i} \right\rceil \cdot 1r) = 31r$ and $\mathcal{W}^c = \sum_i(\left\lceil \frac{32}{min(T_i)} \right\rceil \cdot 1r) = 16r$.

**Scenario 3.** In this simple scenario we will exemplify that in the concurrent case the sorted order influences the wall clock time $\mathcal{W}^c$. We consider the two blocks $b_1$ with $T_1 = 2, \Delta_1 = 2, w_1 = 4r$ and $b_2$ with $T_2 = 8, \Delta_2 = 2, w_2 = 8r$. In a simulation run with $\tau^s_\Omega = 8$, $\mathcal{W}^s = 24r$ (Figure 4a). For $\sigma(b_1) < \sigma(b_2)$, $\mathcal{W}^c = 16r$ (Figure 4b), while for $\sigma(b_2) < \sigma(b_1)$, $\mathcal{W}^c = 20r$ (Figure 4c).

**Scenario 4.** Scenario 4 exemplifies the impact of the delay parameter. We consider $b_1$ with $T_1 = 2, w_2 = 2r$, $b_2$ with $T_2 = 4, w_2 = 4r$, and $b_3$ with $T_3 = 8, w_3 = 8r$. In a simulation run with $\tau^s_\Omega = 8$, $\mathcal{W}^s = 24r$ (Figure 5a). In a first concurrent case, where $\Delta = \varepsilon$ for all blocks in order to obtain a minimal input-output-delay, $\mathcal{W}^c = 16r$ (Figure 5b). In a second concurrent case, with $\Delta_1 = T_1 = 2, \Delta_2 = T_2 = 4$, and $\Delta_3 = T_3 = 8$, $\mathcal{W}^c = 8r$ (Figure 5c).

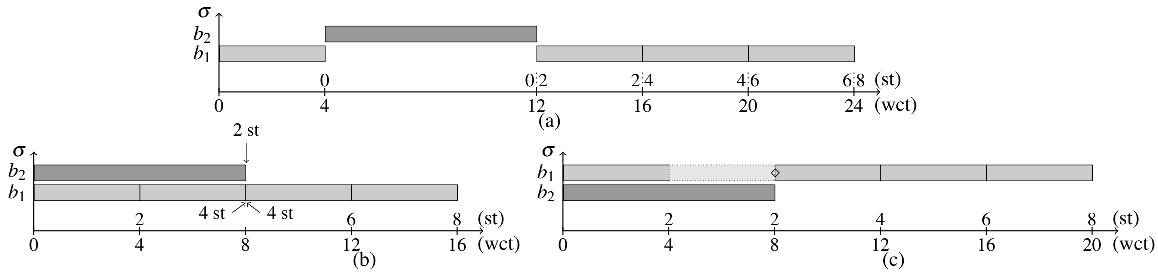


Figure 4: Scenario 3.

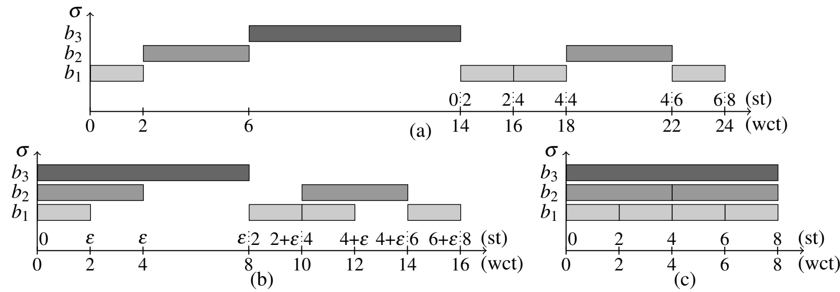


Figure 5: Scenario 4.

### 3.1 Block Dependencies and Delay-Propagation

In the model transformation process described in Section 2.4, the offsets of the blocks in the sequential configuration are taken over to the concurrent configuration. For blocks with data dependencies, this typically results in an increased latency in signal propagation. This is due to the delayed writing of output values. We consider two blocks $b_1$ and $b_2$ with $T_1 = T_2$, $o_1 = o_2$ and $b_1$ providing an output value to $b_2$. In the sequential case (with SR semantics), at each sample hit time $t$, $b_2$ reads the value just provided by $b_1$ at the same simulation time. In the concurrent case, however, $b_1$ has not yet updated the output (to be performed at time $t + \Delta_1$) leading to an outdated input for $b_2$. This may lead to violated end-to-end timing requirements as, for example, data age constraints in cause-effect-chains (Becker et al. 2016).

By adjusting the offset parameters in the concurrent case, this can be avoided. Figure 6 schematizes a model with multiple signal chains. Based on a sequential configuration with a common sample time and zero offsets, for example, in the concurrent configuration the offset of a block may be set to the accumulated delay of its source blocks in the signal chain. Consequently, for a chain of length $n$, the end-to-end latency can be as small as $n \cdot \varepsilon$. Parallelization is then performed in the vertical direction.

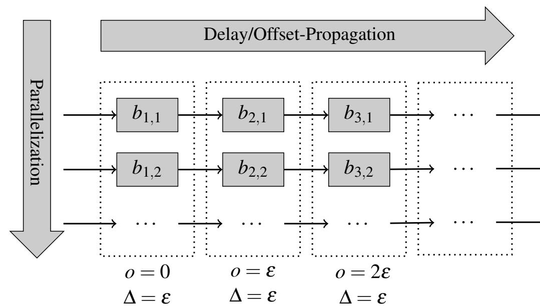


Figure 6: Parallelization of blocks with data-dependencies.

## 4 CASE STUDY

The described parallelization approach was applied to a consolidated Simulink model consisting of three advanced driver assistance systems (ADAS). The three systems are part of the Computer Vision System Toolbox, which is available as a Simulink add-on. The consolidated ADAS model comprises a traffic sign recognition system (TSR), a lane departure warning system (LDW), which warns the driver when the vehicle moves towards the lane lines, and an additional road tracking system that detects and tracks road edges when lane markings are not present. The individual assistance systems have been adapted to work on a common video input. The merged model with three controllers (1 ADAS, MIL) serves as the baseline for our performance comparisons shown in Table 1. The sample time for all three controllers is $0.0\dot{3}$ seconds to account for the frame rate of the input video (30fps) and was taken from the three original models. Afterwards, C code was generated by the Simulink Coder tool. This code base was then used in the model for sequential simulation (1 ADAS, Sequential), where each controller was implemented as a single S-Function block. The sequential model was then translated to a concurrent model (1 ADAS, Concurrent), where the individual controller S-Functions have been replaced by their corresponding concurrent counterpart with a common delay parameter of $\Delta = 0.00001$ seconds. Besides the block for reading the input traffic video sequence, the model only contains one block per controller (as well as terminator and video viewer blocks for debugging purposes, but which have been disabled for the performance measurements). For each measurement the model was simulated over 11 seconds (simulation time) which corresponds to the length of the input sequence. The simulation host PC was a Windows machine with an Intel i7 quad-core with 8 logical processors and 2.6 GHz. The numbers (in seconds) were averaged over three simulation runs. The code generated sequential model (Sequential) already shows some performance gains compared to the baseline model (MIL) containing about 2.000 individual blocks. By applying the presented approach and executing the three controllers in parallel (Concurrent) the wall clock time to simulate the system is reduced further. To see the full benefit of the parallelization, the complexity of the models was increased by adding multiple instances of the controller blocks, leading to models with 6, 9, 12, 15, and 30 controllers, respectively. As reported by the Windows Resource Monitor, the total CPU consumption of the MATLAB process was about 20% in both the MIL and the sequential case, and about 90% in the concurrent case. Especially the data for simulating 6 controllers on a system with 8 logical processors showing a speed-up factor of 2.5 suggests that the involved overhead for this configuration was high. Nevertheless, on average the wall clock time for the simulation was reduced to a third compared to sequential simulation.

## 5 CONCLUSION AND FUTURE WORK

We presented an approach to simulate individual parts of a Simulink model in parallel. This allowed us to take full advantage of the multi-core architecture of today's host computers. Our approach is based on a model transformation and does not require any changes in the sequential simulation engine of Simulink. The latencies that are introduced by delaying output signals are likely compensable by the control algorithms or, depending on the structure of the model, are not even observable at all. Future work will involve the use of a nondirect-feedthrough approach, which enables both complete control of the sorted order of concurrent blocks and a liberation of cyclic dependencies (algebraic loops). However, this requires a more elaborate synchronization mechanism. We plan to combine the parallelization approach with a mechanism used for platform-aware SIL simulations (Naderlinger 2017b), which enables the ex ante analysis of scheduling effects of target platforms on a simulation host.

Table 1: Performance comparisons.

| | 1 ADAS | 2 ADAS | 3 ADAS | 4 ADAS | 5 ADAS | 10 ADAS |
|---|---|---|---|---|---|---|
| Controllers | 3 | 6 | 9 | 12 | 15 | 30 |
| MIL [s] | 10.94 | 19.1 | 28.12 | 40.56 | 49.61 | 97.97 |
| Sequential [s] | 6.59 | 12.17 | 20.15 | 25.71 | 32.37 | 63.51 |
| Concurrent [s] | 3.78 | 4.86 | 6.57 | 7.96 | 9.32 | 17.54 |

## REFERENCES

AUTOSAR 2019. "AUTomotive Open System ARchitecture". http://www.autosar.org, accessed 12th April.

Baruah, S. 2012. "Semantics-preserving Implementation of Multirate Mixed-criticality Synchronous Programs". In *Proceedings of the 20th International Conference on Real-Time and Network Systems, RTNS'12*, edited by L. Cucu-Grosjean, N. Navet, C. Rochange, and J. H. Anderson, 11–19. New York: Association of Computing Machinery.

Becker, M., D. Dasari, S. Mubeen, M. Behnam, and T. Nolte. 2016. "MECHAniSer - A Timing Analysis and Synthesis Tool for Multi-rate Effect Chains with Job-level Dependencies". In *Proceedings of the 7th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*. July 5th, Toulouse, France.

Bouissou, O., and A. Chapoutot. 2012. "An Operational Semantics for Simulink's Simulation Engine". In *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems, LCTES '12*, edited by R. Wilhelm, H. Falk, and W. Yi, 129–138. New York: Association of Computing Machinery.

Canedo, A., and M. A. A. Faruque. 2012. "Towards Parallel Execution of IEC 61131 Industrial Cyber-physical Systems Applications". In *2012 Design, Automation & Test in Europe Conference & Exhibition, DATE 2012*, edited by W. Rosenstiel and L. Thiele, 554–557. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Canedo, A., T. Yoshizawa, and H. Komatsu. 2010. "Automatic Parallelization of Simulink Applications". In *Proceedings of the 8th IEEE/ACM International Symposium on Code Generation and Optimization, CGO'10*, edited by A. Moshovos, J. G. Steffan, K. M. Hazelwood, and D. R. Kaeli, 151–159. New York: Association of Computing Machinery.

Caspi, P., N. Scaife, C. Sofronis, and S. Tripakis. 2008. "Semantics-preserving Multitask Implementation of Synchronous Programs". *Association of Computing Machinery Transactions on Embedded Computing Systems* 7(2):15:1–15:40.

Cha, M., K. H. Kim, C. J. Lee, D. Ha, and B. S. Kim. 2011. "Deriving High-performance Real-time Multicore Systems Based on Simulink Applications". In *Proceedings of the 9th IEEE International Conference on Dependable, Autonomic and Secure Computing*. December 12th-14th, Sydney, NSW, Australia, 267-274.

International Electrotechnical Commission 2013. "Programmable Controllers - Part 3: Programming Languages. IEC 61131-3".

Denckla, B., P. J. Mosterman, and H. Vangheluwe. 2005. "Towards An Executable Denotational Semantics For Causal Block Diagrams". In *Proceedings of the 5th OOPSLA Workshop on Domain-specific Modeling*. October 17th, San Diego, USA.

Görür, B. K., and A. N. Çalli. 2017. "Semi-automatic Parallelization of Simulations with Model Transformation Techniques". In *Mod4Sim '17: Proceedings of the Symposium on Model-driven Approaches for Simulation Engineering*, edited by A. D'Ambrogio, U. Durak, and D. Çetinkaya, 2:1–2:12. San Diego: Society for Computer Simulation International.

Halbwachs, N. 1993. *Synchronous Programming of Reactive Systems*. New York: Kluwer.

Henzinger, T., B. Horowitz, and C. Kirsch. 2003. "Giotto: A Time-triggered Language for Embedded Programming". *Proceedings of the IEEE* 91(1):84–99.

Kumura, T., Y. Nakamura, N. Ishiura, Y. Takeuchi, and M. Imai. 2012. "Model Based Parallelization from the Simulink Models and Their Sequential C Code". In *Proceedings of the 17th Workshop on Synthesis And System Integration of Mixed Information Technologies, SASIMI'12*. March 8th-9th, Beppu, Oita, Japan, 186-191.

Lee, E. A., and S. A. Seshia. 2010. *Introduction to Embedded Systems*. Berkeley: Leeseshia.org.

MathWorks 2018. *Simulink Reference, R2018a*. Natick: The MathWorks, Inc.

Naderlinger, A. 2017a. "Simulating Execution-time Variations in MATLAB/Simulink". In *Proceedings of the 2017 Winter Simulation Conference*, edited by W. K. V. Chan, A. D'Ambrogio, G. Zacharewicz, N. Mustafee, G. Wainer, and E. Page, 1491–1502. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Naderlinger, A. 2017b. "Simulating Preemptive Scheduling with Timing-aware Blocks in Simulink". In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017*, edited by D. Atienza and G. D. Natale, 758–763. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Ozard, J., and H. Desira. 2000. "Simulink Model Implementation on Multi-processors under Windows-NT". In *Symposium on Military, Government and Aerospace Simulation*, 197–204. San Diego: Society for Computer Simulation International.

Rosen, B. K., M. N. Wegman, and F. K. Zadeck. 1988. "Global Value Numbers and Redundant Computations". In *Proceedings of the 15th Symposium on Principles of Programming Languages*, 12–27. New York: Association of Computing Machinery.

Saidi, S. E., N. Pernet, and Y. Sorel. 2019. "A Method for Parallel Scheduling of Multi-rate Co-simulation on Multi-core Platforms". *Oil & Gas Science and Technology* 74(49).

Tuncali, C. E., G. Fainekos, and Y. H. Lee. 2015. "Automatic Parallelization of Simulink Models for Multi-core Architectures". In *Proceedings of the 12th IEEE International Conference on Embedded Software and Systems*. August 24th-26th, New York, NY, USA, 964-971.

## 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. His e-mail address is andreas.naderlinger@cs.uni-salzburg.at.