Modeling AUTOSAR Implementations in Simulink

Jian Chen[1], Manar H. Alalfi[2], Thomas R. Dean[1], and Ramesh S[3]

[1] Department of Electrical and Computer Engineering, Queen's University, Kingston, Canada
{jian.chen, tom.dean}@queensu.ca
[2] Department of Computer Science, Ryerson University, Canada
manar.alalfi@scs.ryerson.ca
[3] General Motors R&D, Warren, MI, USA
ramesh.s@gm.com

Abstract. AUTOSAR (AUTomotive Open System ARchitecture) is an open industry standard for the automotive sector. It defines the automotive three-layered software architecture. One layer is application layer, where functional behaviours are encapsulated in Software Components (SW-Cs). Inside SW-Cs, a set of runnable entities represent the internal behaviours and are realized as a set of tasks. To address AUTOSAR's lack of support for modelling behaviours of runnables, other modelling languages such as Simulink are employed. Simulink simulations assume tasks are completed in zero execution time, while real executions require a finite execution time. This time mismatch can result in failures of analyzing an unexpected runtime behaviour during the simulation phase. This paper extends the Simulink environment to accommodate the timing relations of tasks during simulation. We present a Simulink block that can schedule tasks with a non-zero simulation time. This enables more realistic analysis during the model development stage.

Keywords: AUTOSAR, Simulink, Simulation, Scheduling

1 Introduction

Modern automotive systems are software intensive and the complexity of these systems is rapidly growing. There are many critical functions of modern vehicles that rely on software. An example is embedded controllers that coordinate with each other to perform advanced control functions such as autonomous driving, active safety, an infotainment. All of these functions are related to software which indicates cars contain sone of the largest pieces of software[4]. A modern high-end car features around 100 million lines of code. To address the challenge of automotive systems, a worldwide development partnership AUTOSAR was formed [1]. AUTOSAR standardizes the entire automotive electronic software architecture and development methodology [16].

[4] http://spectrum.ieee.org/transportation/systems/this-car-runs-on-code

While there are many tools that support the AUTOSAR process, MATLAB/Simulink (ML/SL) is a popular option for developing automotive software that meets the AUTOSAR standard, providing a tool chain that supports the AUTOSAR development process. We can directly use Simulink blocks to develop AUTOSAR software components. Embedded Coder[5] provides the mapping of the Simulink models to AUTOSAR components and generates the AUTOSAR compliant production code.

Simulation is a process of representing the actions of a real-world system. Through simulation, engineers can evaluate the system design and diagnose problems in the early phase of the design process. However, the Simulink simulation algorithm dose not take every factor of the real world into account such as the time of real-world computation. In other words, software tasks are completed in zero execution time in the simulation stage. In the real world, software tasks take non-zero execution time that varies according the hardware platform. Hence, simulation cannot reflect a real execution at run-time on a specific target hardware platform. Therefore, developing a more realistic model of an AUTOSAR based software application in Simulink is needed.

AUTOSAR supports a modified version of the priority ceiling scheduling[7]. In this approach, when a lower priority process uses a shared resource, and a higher priority process needs access to the shared resource, the priority of the lower priority process is raised to a higher priority so that it may finish using the resource. This is not always the desired behaviour. Ideally, the contention over the shared resource should be minimized during the modelling phase. However, this requires accurate simulation of the timing of each of the software components.

Automotive ECU (Electronic Control Unit) software consists of multiple threads which are often encapsulated in time-triggered tasks and executed on a Real-Time Operating System (RTOS). The use of model-based development in creating the ECU software is limited in that a thread in an ECU is derived from multiple Simulink/StateFlow models that are independently developed, validated and code-generated. The concept of a thread and timing are largely absent at the time of development and validation of the models. Thus there is a large discrepancy between the run-time semantics and the models giving rise to additional work at run-time. This can be avoided if the run-time abstractions of time-triggered tasks can be captured early at the modelling level. This would enable carrying out detailed concurrency and timing analysis early in the cycle thereby reducing the overall time and efforts involved in the development and validation cycle of ECU development. In this research, we propose an approach that can reflect the real system behaviours during the simulation phase and in the future, identify race conditions at the model level.

[5]https://www.mathworks.com/products/embedded-coder.html

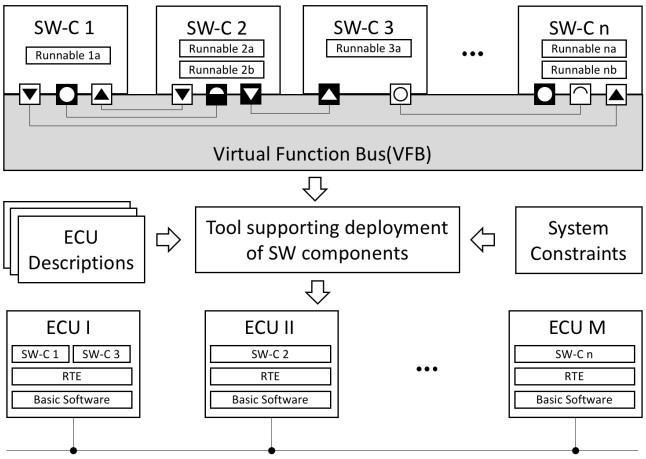

Fig. 1. AUTOSAR components, interfaces and runnables. (Adapted from [1])

1.1 AUTOSAR

AUTOSAR aims to meet the needs of future cars, provides an open industry standard among suppliers and manufacturers. The best way to achieve this goal is to minimize the coupling of software modules through abstraction. Hence, AUTOSAR defines three main layers: the application layer, the runtime environment (RTE), and the basic software (BSW) [17].

The functions in application layer are implemented by SW-Cs, which encapsulate part or all of the automotive electronic functions as shown in Figure 1. The component communications are via a new concept VFB (Virtual Functional Bus), which is an abstraction of all communication mechanisms of AUTOSAR. Using VFBs, engineers abstract the communication details of software components. Inside the SW-Cs, the internal behaviours are represented by a set of runnables. A runnable is the smallest piece of code that can be independently scheduled either by a timer or an event. Finally, runnables are implemented as a set of tasks on a target platform. Runnables from different components may be mapped into the same task and must be mapped in such a way that ordering relations and causal dependencies are preserved.

1.2 AUTOSAR Support in ML/SL

In fact, ML/SL has supported AUTOSAR compliant code generation since version R2006a. ML/SL and Embedded Coder provide a powerful platform for AUTOSAR software development from behaviour modeling to production code generation. First, each single AUTOSAR concept can be represented by an ML/SL block. Existing ML/SL blocks can be applied to AUTOSAR development and no additional AUTOSAR-specific blocks are required. Table 1 shows examples

of key mappings between AUTOSAR concepts and Simulink concepts [15]. Second, ML/SL provides a *Simulink-AUTOSAR Mapping Explorer* for configuring the mapping of Simulink inports, outports, entry-point functions, data transfers, and lookup tables to AUTOSAR elements. Last, Embedded Coder software supports AUTOSAR-compliant C code generation and AUTOSAR XML(ARXML) description files exporting from an ML/SL model.

Table 1. Examples of ML/SL and AUTOSAR Concepts Mapping

ML/SL	AUTOSAR
Subsystem	Atomic Software Component
Function call subsystem	Runnable
Function calls	RTEEvents

1.3 Simulink

ML/SL system models are blocks connected to each other by signals between input and output ports. ML/SL simulation engine determines the execution order of blocks before simulation in a sorted order, called the block invocation order. The block invocation order can be determined by the data dependencies among the blocks. ML/SL uses two kinds of block direct feedthrough and non-direct feedthrough to ensure the simulation can follow the correct data dependencies. A block for which the output ports is directly determined by its input ports is a direct-feedthrough block, while a block for which inputs only affect its state is a non-direct feedthrough block. ML/SL use the following two basic rules to form the sorted order [11]: A block must be executed before any of the blocks whose direct-feedthrough ports it drives; Blocks without direct feedthrough inputs can execute in arbitrary order as long as they precede any block whose direct-feedthrough inputs they drive. All blocks are scheduled in a sorted order and executed in a sequential execution order. The simulink engine maintains a virtual clock to execute each ordered block at each virtual time. Hence, a Simulink block is usually exhibited as a zero execution time behaviour.

Simulink Coder[6] not only supports code generation for ML/SL models, it offers a framework to execute the generated code in a real-time environment. The framework assures the generated code follow the standard of simulation engine and the implementation should preserve the semantics of models. Simulink Coder has two code generation options for periodic tasks: single task and multi-task. Single task implementations can preserve the semantics during the simulation because the generated code is invoked by a simple scheduler in a single thread without preemptions. For multi-task implementations, the generated code is invoked by a rate monotonic (RM) [8] scheduler in a multithreaded RTOS environment,

[6]https://www.mathworks.com/products/simulink-coder.html

where each task is assigned a priority and preemptions occur between tasks. As a consequence of preemption and scheduling, the implementation semantic can conflict with the model semantic in a multi-rate system. Hence, the Simulink simulation does not always reflect the actual model behaviours in implementation. In this work, we develop a scheduler that can schedule the executions of ML/SL blocks with priorities and preemptions during the simulation.

1.4 Scheduler

ML/SL uses a scheduler mechanism to schedule the execution of Simulink subsystems in a specific order [12]. The scheduler is implemented by Stateflow charts and it implicitly controls the order of execution in a Simulink model. There are three kinds of schedulers that can be implemented using Stateflow including Ladder logic scheduler, Loop scheduler, and Temporal logic scheduler. In this work, we developed a new scheduler to replace the ML/SL scheduler to enable a more realistic simulation.

2 Related Work

Logical Execution Time (LET) [6] was introduced as part of the time-triggered programming language Giotto. It abstracts from the physical execution of a real-time program to eliminate I/O execution time so that a LET model execution is independent from its actual execution. LET uses ports to define a logical task execution, input ports take values at the start of a task and the output ports release the values at the end of the task execution. LET has an assumption that actual task execution should be able to be finished during the logical execution. Derler *et al.* [3] demonstrated that real-time software based on LET paradigm has the ability to exhibit the equivalent behaviour on a specific platform during the simulation phase in ML/SL. However, Naderlinger *et al.* [14] points out that data dependency problems may occur when simulating LET-based software.

In order to keep data consistency and preserving semantics, Ferrari *et al.* [4] discuss the proof of absence of interference, disabling of preemption, communication buffers and semaphores as possibilities on a single-core resource in the context of AUTOSAR. Zeng *et al.* [18] present similar mechanisms for the preservation of communication semantics for a multi-core platform.

TrueTime [5] simulator is an ML/SL based network simulation toolbox and it is good for co-simulation of scheduling algorithms, control algorithms, and network protocols. TrueTime is designed as a research tool that requires a learning curve for system engineers to use this tool. Additionally, tasks cannot be expressed directly using production code and requires a special format for function code.

Cremona *et al.* [2] propose a framework TRES, which is used for a co-simulation of the software model and the hardware execution platform. It adds the schedulers and tasks to Simulink models to model the scheduling delays.

Recently, Naderlinger [13] introduces timing-aware blocks into ML/SL, which consumes a finite amount of simulation time so that simulation behaviour of ML/SL models is equivalent to real-time execution behaviour.

Our work differs in the sense that we aim to bring the impact of real-time execution to the semantics of model simulation in the context of AUTOSAR. Hence, our approach natively support AUTOSAR development in ML/SL and the model scheduler can be integrated into code generation.

3 Model Scheduler

In order to reflect the real-time execution of an AUTOSAR Simulink model on a actual hardware during the simulation process, we propose a customized scheduler, Model Scheduler, which schedules the order of execution of each subsystem at a specific time so that Simulink simulation is able to capture the real behaviour of AUTOSAR applications.

Our model scheduler replaces the Stateflow temporal logic scheduler in the ML/SL model and schedules a set of given tasks with non-zero execution time so that the model can have a real-time behaviour during simulation. The model scheduler is implemented as an S-Function block that can easily substitute for the Stateflow scheduler in an ML/SL model. The model scheduler takes tasks and runnables information as input parameters and outputs scheduled subsystem function call triggers. Inside the model scheduler, we implemented a preemptive scheduling algorithm written in C based on Fixed Priority Scheduling (FPS) [9] algorithm, which computes the scheduling and the model scheduler outputs a subsystem function call trigger when a task is scheduled. A function call trigger is a control signal, which triggers the connected subsystem to execute when a control signal has function-call event.

While one of the standard scheduling algorithms in OSEK/AUTOSAR is priority ceiling scheduling, we would like to minimize the changes in priority of tasks due to shared resources. Thus we use FPS so that we can identify race conditions that occur in the model. In FPS, each task has a fixed priority preassigned by users, and they are stored in a ready queue in an order determined by their priorities. The highest priority task are selected from the ready queue to execute. The oldest task will be selected if there are more than one of same priority tasks exist. In a preemptive system, if a higher priority task is scheduled during the execution of a lower priority one, then the higher priority task is executed immediately and the lower priority task is moved to the ready queue. RM scheduling algorithm is one of the widely used FPS algorithm and it is used in Simulink Coder for code generation. In RM scheduling, the priority of a task is associated with its period, a task has a smaller period then it has a higher priority.

The S-Function provides a mechanism to extend the capabilities of ML/SL by customizing blocks and S-Functions can be accessed from a block diagram so-called *S-Function block*. Customized algorithms can be added to Simulink models via S-Function blocks written either in MATLAB or C. An S-Function block has

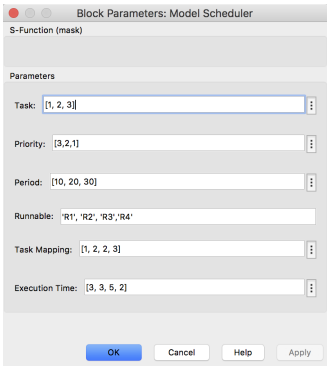

Fig. 2. Model Scheduler parameters

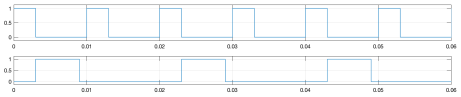

Fig. 3. Task active chart

a parameter field that users pass specify parameters to the corresponding S-Function block. S-Functions communicate with the ML/SL engine through a set of callback functions so-called S-Function API. Thus, S-Functions make it possible to control ML/SL simulation process by using a customized algorithm.

4 Tool Implementation: Model Scheduler

Model Scheduler is implemented as a custom configurable Simulink library block that includes an FPS algorithm written in C and invokes the connected subsystem in the ML/SL interactive development environment.

An S-Function contains a set of callback functions, which the simulation engine executes at different stages during simulation. We use output function *(mdlOutputs)* that computes the output values based on the input parameters. Before running a simulation, we need to provide the necessary parameters in the block parameters dialogue shown in Figure 2. The parameters include *Task*, *Priority*, *Period*, *Runnable*, *Task Mapping*, *Execution time*. The user-entered parameters are implemented by a block mask, which provides the parameter dialogue box.

A simple example shown in Figure 4 illustrates the usage of our model scheduler. In this example, model scheduler schedules three runnables and they are mapped to two tasks. Each subsystem represents an AUTOSAR runnable. In general, the execution order of specific Simulink subsystem is determined by a Stateflow scheduler. In our case, the Stateflow scheduler is replaced by our tool Model Scheduler and each Simulink subsystem is scheduled to be executed at a specified time with a finite execution time. Model Scheduler yields three outputs. The first output port is the trigger signals that periodically output the specified time for each subsystem. The trigger signals are connected to a demux block which splits the multiple trigger signals to a single signal to trigger each subsystem. The second output port is a runnable activation chart that illustrates each runnable schedule. It shows the start and finish time of each runnable including

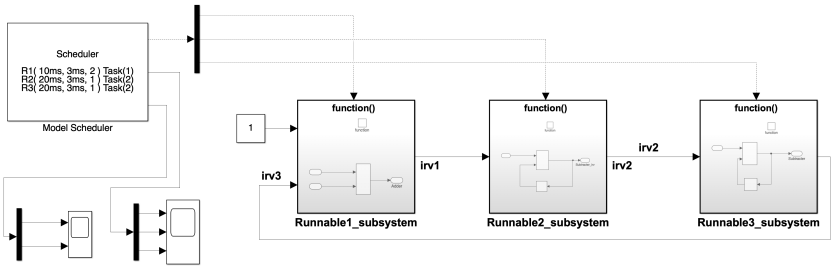

Fig. 4. A simple example of using Model Scheduler to schedule AUTOSAR SW-Cs.

the runnable execution time. The third output port is a task activation chart that illustrates each task schedule.

Model Scheduler is based on an FPS algorithm and implemented as a level 2 S-function written in C. Our algorithm takes as arguments the six input parameters mentioned above. The parameters can be grouped into two levels, one describes the properties of a task such as *Priority* and *Period* the other one describes the properties of a runnables such as *Task Mapping* and *Execution Time*. Model Scheduler reserves this two-level information and computes the current active runnable signal. In this work, we assume execution time of each runnable is already known. The execution time could either be measured by running the code on a test platform, or by analysing the behaviour of generated code (or Simulink model) by off-the-shelf tools. Normally the FPS algorithm computes a scheduling table, then a runtime dispatching algorithm invokes each task according to the precomputed table.

Simulating a model has three phases: model compilation, link phase and simulation loop phase [10]. The Simulink engine each time goes through the loop is called as one simulation step. In each simulation step, model scheduler is executed and computes the running task and runnable of current sampling time and yields a signal to the output port when the current sample time is a beginning of a task period. The output signal triggers the connected subsystem, which is a runnable of the current execution task. The model scheduler determines the current active runnable along with the associated task information at each single simulation step. If a task or a runnable is expected to run at this simulation step, then model scheduler invokes a macro to trigger the subsystem connected to model scheduler.

Let us see an example of how model scheduler performs the schedule computation. The simple example (Figure 4) has the following settings shown in Table 2. During the simulation loop phase, task T_1 and T_2 are all scheduled at the first simulation step and model scheduler maintains a scheduling table to store the scheduled tasks. T_1 is the only executed task at the first simulation step due to its higher priority and the execution of T_1 takes 3ms as only runnable R_1 is mapped to T_1. In the first simulation step, model scheduler output a function-call signal to trigger T_1 that is connected to the first output port of

Table 2. The simple example settings

Task	Period(ms)	Execution Time(ms)	Priority	Runnable
T_1	10	3	2	R_1
T_2	20	3	1	R_2
T_2	20	3	1	R_3

demux *Runnable1_subsystem*. There is no output signals at the simulation step two because it is still during the execution of T_1. Until simulation step three, T_1 completes its execution and it is time to trigger T_2. R_2 and R_3 are mapped to T_2 so they have the same priority and period. R_2 is executed at this simulation step because its connection order is before R_3. R_3 is executed right after the completion of R_2. After execution of R_3, that is an idle time so there is no trigger signal being output. Figure 3 illustrates the task execution process during each simulation step. T_1 is active during the first three simulation steps and T_2 is active at the following six simulation steps in the first $10ms$ period.

5 Case Study

In this section, we use AUTOSAR compliant ML/SL models scheduled by our model scheduler to show the scheduler can capture the actual behaviour on a hardware platform during a Simulink simulation.

5.1 AUTOSAR Model Scheduling

First, we demonstrate a simple example that shows how Model in the Loop (MIL) analysis benefits from our model scheduler. Figure 5 shows a simple example using a model scheduler to schedule four runnables mapped to three tasks. The details of settings are shown in Table 3.

Table 3. Using model scheduler parameters setting

Task	Period(ms)	Execution Time(ms)	Priority	Runnable
T_1	10	3	3	R_1
T_2	20	3	2	R_2
T_2	20	5	2	R_3
T_3	30	2	1	R_4

In this example, we have four runnables which are mapped to three tasks: R_1 is mapped to T_1; R_2 and R_3 are mapped to T_2; R_4 is mapped to T_3. The model scheduler takes parameters of tasks as input to calculate three outputs. The first output is the runnable triggers which are connected to four subsystems accordingly. The other two outputs are time execution diagrams of tasks and runnables.

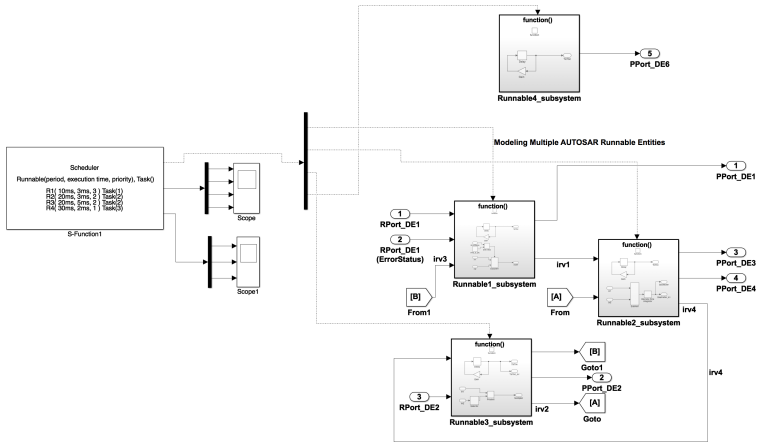

Fig. 5. Using Model Scheduler to schedule AUTOSAR SW-Cs.

If we simulate this example using a standard scheduler, the execution order of this example is $T_1T_2T_3$ or $R_1R_2R_3R_4$ respectively and they are completed within the first $10ms$ period based on tasks settings. In reality the above order is not possible, since each task requires a certain amount of execution time on a hardware platform. Our scheduler tries to be realistic by taking execution time into account.

Figure 6 shows runnables execution diagram of our scheduler. There are four output signals and each output signal represents each runnable. The first signal shows the execution of R_1 in T_1. It has the highest priority so it is trigged at the beginning and takes $3ms$ execution time. After the execution of T_1, the next highest priority is T_2 with 2 runnables. R_2 is triggered at time of $3ms$ and takes another $3ms$ execution time. R_3 is supposed to be triggered right after the completion of R_2. However, R_3 is triggered right after the completion of the second R_1 instance because the period of T_1 is $10ms$ and the execution time for both R_1 and R_2 are $3ms$. There is only $4ms$ left before R_1 is triggered at next period and it is less than the execution time of R_3 of $5ms$. Because a runnable is the smallest atomic component within a SW-C, there is no preemption between runnables. R_3 cannot be preempted by R_1. Thus, R_3 is scheduled to be executed after the completion of second R_1 instance. The execution order of our scheduler is $T_1T_2T_1T_2T_3$ or $R_1R_2R_1R_3R_4$. Figure 7 shows the tasks execution diagram. There are three output signals represent tasks execution. During the first period of T_2, T_1 is triggered twice and T_2 is preempted by T_1.

5.2 Scheduler Example

In this section, we use an example model, which is scheduled by two different schedulers the Stateflow Scheduler and Model Scheduler, to show our scheduler is able to simulate actual behaviours during the simulation phase. The State-

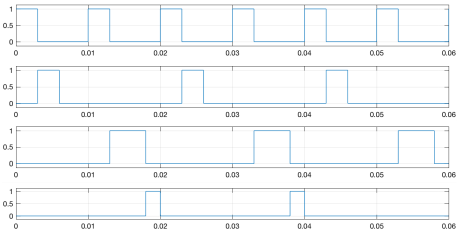

Fig. 6. Runnables execution time diagram

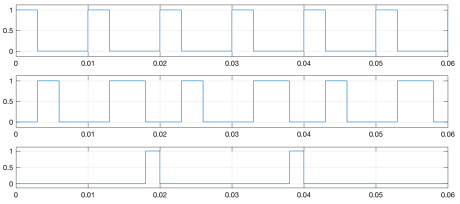

Fig. 7. Tasks execution time diagram

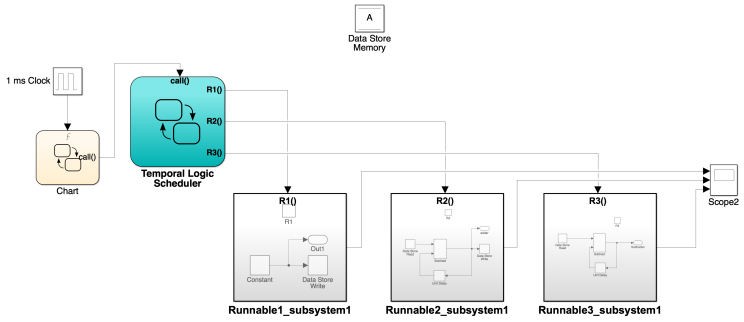

Fig. 8. An Example using Stateflow Scheduler

flow scheduler takes zero execution time during the Simulink simulation. On the contrary the model scheduler triggers each subsystem which takes a specified execution time during simulation. By comparing simulation results between these two schedulers, the potential unexpected behaviours of Simulink models are exposed.

Figure 8 shows the normal example which uses a Stateflow temporal logic scheduler to trigger each task. The parameter settings are the same as Table 2 except the execution time of R_3 is $5ms$. There are three runnables (R_1, R_2, R_3) mapping to two tasks (T_1, T_2) in this example. R_1 writes a constant value to a global variable A. R_2 reads A first then writes the summation of A and its delay value to A. R_3 reads A then subtracts its delay value from A, and outputs the result. Figure 9 shows the simulation output of this normal example. The three signals are the outputs of R_1, R_2, R_3 from top to bottom. From this simulation result, the output of R_3 is an increasing number. In the normal simulation, the execution order are T_1T_2 or $R_1R_2R_3$.

We replaced the Stateflow scheduler with our model scheduler and run simulation again, we can get a different result shown in Figure 10. In the second simulation, the output of R_3 is a pattern of zero, constant value, which is different from the previous example. The execution order of this example are $T_1T_2T_1T_2$ or $R_1R_2R_1R_3$. In the previous example, R_3 always reads A which is written by

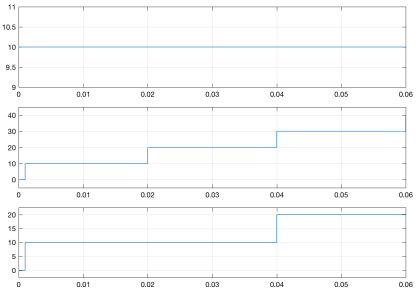

Fig. 9. Output of Stateflow scheduler

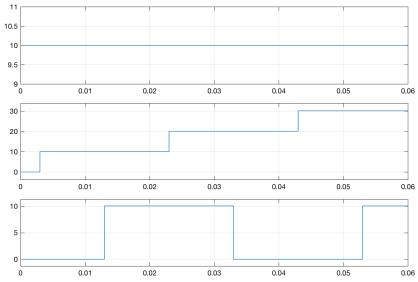

Fig. 10. Output of Model Scheduler

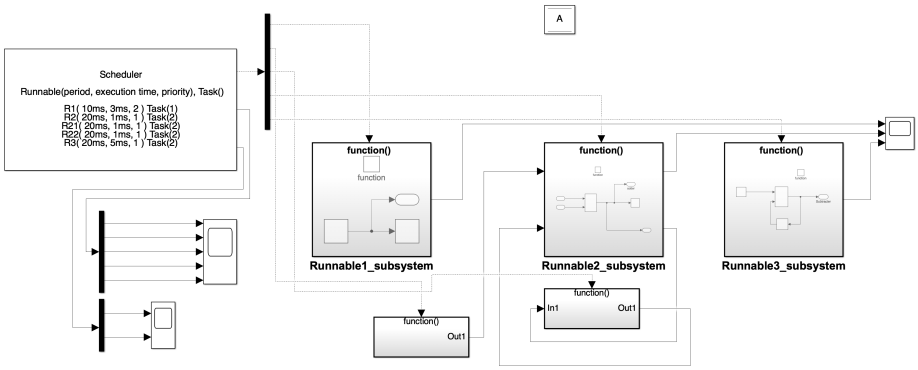

Fig. 11. Splitting a single runnable into three subsystems

R_2. Using our scheduler, R_3 reads the global variable A from the output of the second R_1 instance because T_2 is preempted by T_1 during the execution of T_2.

So far, we have demonstrated model scheduler is capable of simulating task interference while runnables are atomic execution. However, runnables run within the context of a task and tasks can be preempted. Hence, runnables can be preempted by runnables in other tasks. In order to show model scheduler is able to simulate the preemption at runnable level, we manually split a single runnable into several function-call subsystems so that our model scheduler treat these subsystems as "runnnables" to simulate runnables preemptions. This splitting example is depicted in Figure 11. When we set the parameters accordingly, we can simulate the preemptions at the runnable level.

6 Limitations

One of the shortcomings of model scheduler is that our model scheduler does not yet support runnable preemption automatically. In reality, runnables can be preempted by another task during its execution. If we can simulate this scenario

during simulation phase automatically, it can increase the confidence of design and reduce human effort. Since the time for each block is an estimation, it may not accurately represent the real time of the system. Multiple simulations with different time parameters may be needed to cover the possible behaviours of the system.

7 Future Work

We plan to automatically transform Simulink models to subdivide a runnable into subsystems automatically. Additionally, our current model scheduler only support periodic events. Both periodic and aperiodic tasks exist in real-time system and aperiodic events are necessary in automotive software. One possible area of expansion is to support aperiodic events in our model scheduler. Further, we could add more real-time scheduling algorithms such as Earliest-deadline-first (EDF) scheduling to model scheduler so that engineers can verify the design under different scheduling algorithms to meet the requirements of diverse target platforms

We also plan to use the modified models to identify interference between tasks. Currently, our model scheduler requires execution time as parameters to perform simulation so that we can find potential issues during the simulation phase. In the future, we perform a model scheduler simulation based on the input parameters, we can model the execution times as variables inside the scheduler and change the value of execution until we find a potential interference.

8 Conclusion

Model scheduler is able to schedule Simlulink models in a more realistic way so that ML/SL simulation can reflect the real-time execution on the target platform. This was implemented in an S-Function block based on FSP algorithm written in C. Model scheduler can manage the hierarchy of tasks and runnables, moreover runnables are scheduled according to the tasks parameters. We have demonstrated a few simple examples to show the abilities of model scheduler. The approach discussed in this paper enables that ML/SL simulation takes software execution time into account without any modification to the current models. It can fill in the gap between the semantics of model simulation and its real-time execution.

References

1. AUTOSAR: Autosar development partnership. http://www.autosar.org (2018)
2. Cremona, F., Morelli, M., Di Natale, M.: TRES: A Modular Representation of Schedulers, Tasks, and Messages to Control Simulations in Simulink. Proceedings of the 30th Annual ACM Symposium on Applied Computing (2015) 1940–1947

3. Derler, P., Naderlinger, A., Pree, W., Resmerita, S., Templ, J.: Simulation of LET models in Simulink and Ptolemy. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Volume 6028 LNCS., Springer, Berlin, Heidelberg (2010) 83–92
4. Ferrari, A., Di Natale, M., Gentile, G., Reggiani, G., Gai, P.: Time and memory tradeoffs in the implementation of AUTOSAR components. In: 2009 Design, Automation & Test in Europe Conference & Exhibition, IEEE (apr 2009) 864–869
5. Henriksson, D., Cervin, A., Årzén, K.E.: TrueTime : Real-time Control System Simulation with MATLAB / Simulink. Proceedings of the Nordic MATLAB Conference (2003)
6. Henzinger, T.A., Horowitz, B., Kirsch, C.M.: Giotto: A time-triggered lan- guage for embedded programming. Emsoft **91**(1) (2001) 166–184
7. L., S., R., R., J., L.: Priority inheritance protocols: An approach to real-time synchronization. IEEE Transactions on Computers **30**(9) (Septebmer 1990) 1175–1185
8. Lehoczky, J., Sha, L., Ding, Y.: The rate monotonic scheduling algorithm: exact characterization and average case behavior. [1989] Proceedings. Real-Time Systems Symposium (1989) 0–5
9. Liu, C.L., W. Layland, J.: Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment Scheduling Algorithms for Multiprogramming. Journal of the Association for Computing Machinery **20**(1) (jan 1973) 46–61
10. MathWorks.: Developing S-Functions, r2017b. http://www.mathworks.com (2017)
11. MathWorks.: Simulink User's Guide, r2017b. http://www.mathworks.com (2017)
12. MathWorks.: Stateflow User's Guide, r2017b. http://www.mathworks.com (2017)
13. Naderlinger, A.: Simulating preemptive scheduling with timing-aware blocks in Simulink. In: Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017, IEEE (mar 2017) 758–763
14. Naderlinger, A., Templ, J., Pree, W.: Simulating real-time software components based on logical execution time. In: Proceedings of the 2009 Summer Computer Simulation Conference. SCSC '09, Vista, CA, Society for Modeling & Simulation International (2009) 148–155
15. The AUTOSAR Consortium: Applying simulink to autosar, r3.1. (2006)
16. The AUTOSAR Consortium: AUTOSAR Methodology, r4.3. (2018)
17. The AUTOSAR Consortium: The AUTOSAR Standard, r4.3. (2018)
18. Zeng, H., Di Natale, M.: Mechanisms for guaranteeing data consistency and flow preservation in AUTOSAR software on multi-core platforms. In: SIES 2011 - 6th IEEE International Symposium on Industrial Embedded Systems, Conference Proceedings, IEEE (jun 2011) 140–149