Multiple Real-Time Semantics on top of Synchronous Block Diagrams

Andreas Naderlinger
C. Doppler Laboratory Embedded Software Systems
University of Salzburg, Austria
andreas.naderlinger@cs.uni-salzburg.at

Keywords: Model-based design, embedded real-time systems, synchronous, Simulink, preemption

Abstract

Synchronous block diagrams form an established fundament for the model-based development of embedded real-time systems. Their synchronous reactive (SR), also called zero execution time, semantics offers indisputable advantages in designing, testing and verifying control algorithms but poses problems in the translation of multi-rate models into code. In this paper, we contrast the semantics of three different real-time programming paradigms and discuss a mechanism to represent them in models with SR semantics. This representation is based on MATLAB/Simulink blocks that are not characterized by the typical zero time behavior but whose execution may last for and optionally consume a finite amount of simulation time. Each such block represents a task in the sense of a real-time operating system. All tasks within a model may be scheduled with a static-priority approach. This allows us to observe simulations that are closer to the real timing behavior of control applications and also to consider preemption effects already in the simulation.

1. INTRODUCTION

Model-based design has become an established development approach in the field of embedded real-time systems and is a promising candidate to tame the ever increasing complexity. The model-based methodology moves away from manual low-level coding and is centered around an abstract mathematical model. Formal models are well suited in proving certain system properties and for performing analysis and verification. Code generators allow to automatically synthesize the implementation code. This reduces both development time and errors and is thus considered best practice for software development.

1.1. Synchronous Reactive Models

The development of real-time control systems involves multiple disciplines and has to consider control engineering, software, and hardware aspects. In many cases, it starts with a time-invariant controller model that is high-level and assumes concurrency under unlimited resources. In particular, *synchronous reactive (SR)* [1] models are well understood and heavily used in designing hardware logic and control applications, for example, in the automotive and aeronautic domain. In SR models, a reaction (e.g. the execution of a task) to an event must be completed before the occurrence of another event. Accordingly, task execution times may assumed to be zero. A major difficulty is that the ideal SR semantics does not hold in the implementation and that the original behavior is not preserved at run-time, although the generated code may match the modeled functionality perfectly. Typically, simulation does not properly consider the timing of the software that will implement the controller functionality [10]. While tasks are assumed to always complete in zero time in the simulation, they are known to require a finite execution time at run-time on a hardware platform. Additionally, at run-time, multiple tasks compete for the limited resource CPU, however many model-based development environments are not suited for gathering scheduling, system load or other non-functional requirements. With regard to a reasonable CPU utilization, code generators synthesise multi-rate models into multi-task implementations [20]. A real-time operating system (RTOS) employs, for example, a preemptive scheduling policy to execute the tasks at the specified rate. Even without any data dependencies between tasks, the execution of one task may have effects on the results provided by another one. So, the actual behavior can only be tested on the hardware target and since the generated code is typically fine-tuned manually, the correspondence to the model is lost. As a consequence, sooner or later, models and simulation results are often abandoned, when the hardware platform and communication topology starts dominating the development process [17].

1.1.1. Incremental Refinement vs. Transparency

Ideally, a model is valuable throughout the different development phases and simulation results are incrementally refinable. This requires models to be valid at different levels of abstractions, from a high-level, time invariant, and *platform independent* to a concrete and possibly *platform specific model* (PIM and PSM in the Model-Driven Architecture [13]). The final model eventually 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.

A different research direction strives to move away from the platform-centric approaches and to apply a different programming paradigm that eliminates the need for platform-specific models to a certain extent. Timing constraints are expressed by high-level programming languages and guaranteed by a compiler that checks the constraints for a given platform [9].

This article is organized as follows. Section 2 summarizes three well-known paradigms in real-time programming. Section 3 describes the main contribution, a mechanism to represent them in models with SR semantics. We use MATLAB/Simulink [19] as a prominent software product that enables the development and simulation of SR models using block diagrams. A case study of the presented approach is described in section 4. We discuss related work in section 5 and conclude the article in section 6.

2. RT PROGRAMMING PARADIGMS

According to [11], we can differentiate between three programming paradigms for concurrent real-time (RT) software, only one of which is covered by Simulink directly. In the following we shall describe all of them and present subsequently how the remaining two can also be considered in Simulink.

2.1. Zero Execution Time (ZET) Model

The *zero execution time (ZET)* (aka *synchronous*) model [5] abstracts from the physical execution time of tasks. The key assumption of this model is that tasks execute in zero time, i.e., they provide their results immediately. A major advantage of ZET-based languages is their amenability to verification making them attractive to safety-critical applications. Practical implementations of the ZET model must ensure that a task reads its input values at the occurrence of the triggering event and provides the output before the occurrence of the next event. Although not based on a formal semantics, Simulink follows the synchronous approach (see section 3.1).

2.2. Logical Execution Time (LET) Model

In the LET (aka *timed*) model, the execution of a task is associated with a logical time span, called the *logical execution time (LET)* [7]. Inputs are read only at the beginning and outputs are written only at the end of the LET, i.e. at predefined time instants called *release* and *termination time* respectively. These I/O operations are assumed to execute in ZET, i.e., they are synchronous operations. During the LET of a task there is no communication or synchronization point. The LET is independent from the time it takes the task to execute on a particular platform. The actual execution of the task is subject to some scheduler and may start after its release time or even get preempted. However, the task must be completed no later than the end of the LET. The worst case execution time (WCET) of the task must be known and the following relation must hold: execution time $\leq$ WCET $\leq$ LET. Tasks that are completed before the termination instant buffer their results. Applications following the LET paradigm exhibit equivalent behavior on different kinds of computational nodes and communication systems [4].

2.3. Bounded Execution Time (BET) Model

The *bounded execution time (BET)* (aka *scheduled*) model is the classical and probably most widely used model for real-time programming. It relies on classical scheduling theories as described, for example, in [12]. For a BET program to execute correctly, it must be ensured that each task completes before its deadline, i.e., within its execution bounds. The necessary runtime support for BET programming is provided by the underlying operating system, e.g., by means of adequate scheduling mechanisms. Tasks in the BET model provide their output when they finish execution, thus the response time is equal to the execution time of tasks. The execution time of a task depends, for example, on processor performance, scheduling mechanism, system load, and memory caches. Consequently, the programs based on the BET model may suffer from variations in their input/output behavior (jitter) and are not compositional with respect to their observable timing behavior.

3. SIMULATION OF TIMING BEHAVIOR

In this section we present a mechanism to simulate tasks conforming to different programming paradigms. We support all three paradigms discussed above within Simulink. Simulink is close to SR semantics and naturally supports the ZET paradigm only. The two other paradigms differ in the sense that computation takes time. Simulink provides explicit delay blocks to postpone the propagation of signals. This is a widely used mechanism to mimic the execution time of a control algorithm on an actual hardware platform in the simulation. We argue that a simulation would benefit from a more fine-grained consideration of execution time to become more realistic. This becomes even more true if complex control algorithms are executed as a single block, for example, when whole subsystems are translated into S-Function blocks that execute their corresponding C code (see the case study in section 4). To properly support the BET paradigm, time must not only pass after a block has been executed. Time must pass during the execution of a block. Currently, to the best of our knowledge, Simulink itself provides no mechanism to support this. We will start with a short summary of how Simulink executes blocks and then describe a mechanism to support also tasks based on BET and LET as Simulink blocks.

3.1. Block Execution in Simulink

In simple terms, at each simulation step, the Simulink engine first computes the set of all outputs of all blocks in the model as a function of the current state (and optionally of the input), second it updates the state of all blocks. This is done by executing the blocks in a fixed order that was derived during the initialization. Executing a block means to invoke a set of C functions. Most importantly, the engine executes the output function of all blocks in a first and then all the update functions in a second iteration. In the output function, the block calculates and potentially updates its output signals. In the update function, the block calculates its new state. After the model's state has been updated, the engine advances the simulation time. For deriving a fixed-point solution, Simulink requires each block in the model to be executed only once at every time step. Thus, the whole code corresponding to a particular block is executed at one simulation time instance and without consuming any simulation time.

3.2. Simulation-time Consuming Blocks

In the following we present the concept of a Simulink block that is not necessarily characterized by the typical zero execution time behavior [14]. Instead, the execution of the block may last for a finite amount of simulation time and span several simulation steps. During the execution, the block may perform I/O operations by reading input signals and updating output signals. The block is based on Simulink's S-Function interface and integrates well with standard Simulink blocks. Each such block in a model represents one task in the sense of an RTOS and is described by a set of properties, such as *period* and *offset*, as well as a *task function* implemented in C. This block serves as the basis to implement tasks that may follow the various programming paradigms. Depending on the paradigm in use, different additional properties, such as the *LET* or the *priority* of the task are required to be specified.

3.2.1. The Task Model

In our model, a task is a sequential program without any internal synchronization point (comparable to OSEK's basic task concept [16]) and can consequently not be blocked by waiting for an external event. The task function consists of a finite number of contiguous segments. Each segment has an associated execution time. Segment boundaries represent barriers that allow synchronization with the simulation environment. The execution of the segment is performed instantaneously, i.e. the operations are performed in zero execution time. However, the next segment is not executed before the specified execution time of the previous segment has elapsed. Execution times of the individual segments are assumed to be known, e.g. using WCET analysis tools. The execution time information is part of the C code. It may have been introduced by means of adequate code instrumentation tools.

The timely progress to the next segments of a task function is based on a synchronization mechanism that decides when to proceed with the execution of a particular task and when to return the control to the simulation engine to execute other parts of the model such as the plant or to put the simulation clock forward.

3.3. A Synchronization Mechanism

This subsection exemplifies the usage of a synchronization mechanism in order to enable simulation of all three paradigms ZET, BET, and LET. Figure 1 shows a Simulink model containing a ramp block as signal source, an instance of our simulation-time consuming block (X-Task) representing a task, and a scope. The task may follow either the ZET, the BET, or the LET paradigm. Individual sample implementations are listed below. For demonstration purposes, they all read two input values and write these values on a single two-dimensional output signal without performing any meaningful computation. Their different behaviour is discussed subsequently.

The task functionality is to be implemented in a single function called `taskFunction` and all three implementation share the following code frame. `InputRealPtrsType` is a type definition provided by Simulink for accessing input signals. `Task` is part of our synchronization API and used, for example, to hold information about the task period, priority, etc.

```
#define IRPT InputRealPtrsType

void taskFunction(SimStruct *S) {
  Task *t = ssGetUserData(S);
  real_T a, b; //input
  real_T *c;   //output

  // (i) ZET, (ii) BET,
  // or (iii) LET implementation from below
}
```

3.3.1. ZET (i)

The ZET code conforms to an ordinary S-Function implementation. The *syncInput* and *syncOutput* functions are aliases for `ssGetInputPortSignalPtrs` and `ssGetOutputPortSignal` from the S-Function API re-

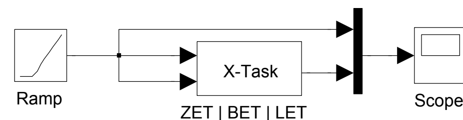


Figure 1: A task with either ZET, BET, or LET semantics in Simulink. The different results of the Scope block are shown in Figure 2.

spectively and access the input/output signals. The first *syncInput* call reads the first (and only) element ([0]) of the input signal of port 0 (which is signal *a*). The second call reads signal *b*. The call *syncOutput* returns a pointer to the output signal *c*. Subsequently, the two-dimensional signal *c* is assigned to *a* and *b*. No simulation time passes during the execution of the code.

```
a = *((IRPT) syncInput(0, t))[0];
//... computation
b = *((IRPT) syncInput(1, t))[0];
//... computation
c = (real_T *) syncOutput(0, t);
c[0] = a;
c[1] = b;
```

3.3.2. BET (ii)

The BET code is very similar. The additional parameter in *syncInput* and *syncOutput*, however, indicates the modified semantics. This parameter refers to the time it takes to execute the last code segment, i.e. the simulation time that should have passed since the last synchronization call. The task reads its first input immediately when the task is released (i.e. after 0.0 seconds). We then assume a computational operation that requires $2\mu s$. Then the second input is read (consequently 0.000002 seconds after the last API call). After another operation that requires $1\mu s$, the two output values are written. In the case of BET, *syncInput* and *syncOutput* also perform the synchronization with the simulation environment, such that simulation time may pass or other blocks may execute. The total execution time of this task is $3\mu s$.

```
a = *((IRPT) syncInput(0, t, 0.0))[0];
//... computation
b = *((IRPT) syncInput(1, t, 0.000002))[0];
//... computation
c = (real_T *) syncOutput(0, t, 0.000001);
c[0] = a;
c[1] = b;
```

Optionally, the specified execution time does not only delay the execution of the subsequent code segment, but also prevents other blocks from executing for this particular time span. This simulates the behavior on an actual hardware platform, where the execution of a task consumes CPU time. Scheduling of multiple BET tasks is discussed in section 3.4 in more detail.

3.3.3. LET (iii)

The LET paradigm requires a different coding style as I/O operations are only allowed at the release and termination time. The task function is split into two pieces. The first part is executed immediately when the task is released. All inputs are read and the actual functionality is performed. The second part is executed after the LET has expired, independently of the actual execution time of the task. Then all outputs are written. The I/O operations *syncInput* and *syncOutput* are again only aliases for the appropriate Simulink functions. The *syncRelease* and *syncTerminate* call mark the release and termination time of the task, respectively. It delays the propagation of the output value until the end of the task's LET (as specified as a task parameter) without consuming CPU time for the duration of the LET.

Obviously this is the same behavior as executing a ZET task (or an ordinary S-Function) whose output signals go through a unit delay block with a sample time equal to the task's LET. However, simulating tasks that have different LETs in different modes is not feasible with this delay block approach [18]. Additionally, the presented LET approach helps to verify that the execution time of a particular program path is really smaller than the specified LET of the task.

```
syncRelease(t);
a = *((IRPT) syncInput(0, t))[0];
b = *((IRPT) syncInput(1, t))[0];
//... computation
syncTerminate(t);
c = (real_T *) syncOutput(0, t);
c[0] = a;
c[1] = b;
```

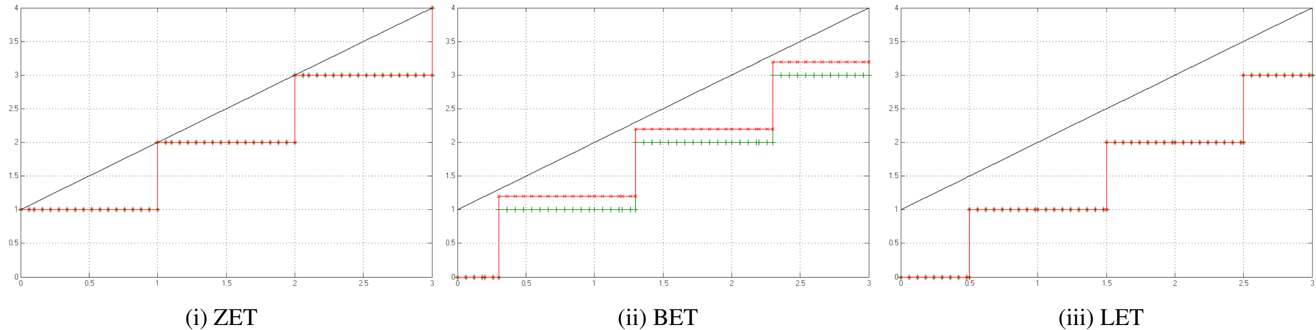
3.3.4. A Comparison between ZET, BET, and LET

Figure 2 shows the different behavior of the three paradigms. Note that in order to obtain a reasonable axes-labeling of the Simulink scope, the timing values (x-axis) have been scaled, such that $10\mu s$ represents $1s$. In all three cases, the period of the task is $10\mu s$ and the offset is 0. The LET in (iii) is specified to be $5\mu s$. The ramp parameters are as follows: *slope = 1, start time = 0, initial output = 1.*

The black line corresponds to the ramp signal, the green line corresponds to output $c[0]$ (input a), and the red line corresponds to output $c[1]$ (input b). For the ZET and LET implementation, both input values are read at the same time, i.e. when the task is released. Thus the two output signals overlap in both cases ($c[0] = c[1]$). However, the ZET implementation provides the outputs instantaneously (at release time), whereas the LET implementation delays the output by the logical execution time of the task ($5\mu s$). The BET implementation only reads the first input (a) at release time. The second one (b) is read with a delay of $2\mu s$ (thus the ramp counter has already increased further). When the task provides its outputs $1\mu s$ afterwards, the different values become obvious.

3.4. Scheduling of BET Tasks

As described above, when executed on a hardware platform, the tasks compete for the CPU and the scheduling schema applied may considerably influence the behavior. However, whether or not these scheduling effects are observable depends on the programming paradigm. The LET



(i) ZET (ii) BET (iii) LET

Figure 2: Output of the scope block from Figure 1 with the respective paradigm.

paradigm abstracts from the physical execution and thus from a particular scheduling. A LET program behaves exactly the same on any platform where the program has been shown to be *time safe* [8]. This time safety check also includes a schedulability analysis.

A BET program, in contrast, is prone to scheduling effects such as preemption, variations in task execution times, or overhead of the real-time operating system. These effects may cause nondeterminism and data integrity problems. Even without any data dependencies between tasks, the execution of one task may have effects on the results provided by another one (see the example in section 3.4.3).

3.4.1. Static-priority Rate Monotonic Scheduling

In our simulation, each BET task block represents one task in the sense of an RTOS. It may autonomously read its inputs and write its outputs. Additionally, interaction between tasks may be based on shared memory communication via global variables. All BET tasks share the CPU and are subject to rate monotonic scheduling [12].

In addition to period and offset (phase), each BET task has a priority assigned statically. The priority of the individual tasks of a model serves as the basis for our scheduling mechanism and is also used to impose the correct sequence in Simulink's block update order. This order is derived once in the start-up phase of the simulation and remains constant during the whole simulation. The scheduling and timely execution of the individual segments of a task function is based on our synchronization mechanism mentioned above that decides which task to execute and when to continue with the simulation.

Currently, we only support *static-priority scheduling* [2]. We plan to further investigate, whether dynamic-priority scheduling mechanisms are also possible without distorting the exact timing. Compared to other simulation environments, such as Ptolemy [3] that fosters different execution strategies (models of computation), Simulink does not give much leeway in this respect and ensuring correct semantics is nontrivial. Our current implementation assumes that only one active instance for each task exists at any time, i.e. the worst-case reaction time of each task is assumed to be smaller than its period. Furthermore, the current implementation is limited to the simulation of single-processor systems and task priorities must be unique.

3.4.2. Full Preemption vs. Cooperative Tasks

The presented approach does not provide full preemption support. The scheduling rather relies on cooperative tasks. Tasks cannot be interrupted arbitrarily, but only at defined points (see `sync...` calls above). As long as all I/O operations are synchronized, the observable behavior is the same. It must be noted, however, that I/O operations are assumed to be atomic. Consequently, potential data integrity problems are ignored by now.

3.4.3. Scheduling Example

In the following example we contrast the traditional approach of using S-Functions in combination with standard Unit-Delay blocks with our proposed mechanism to consider task execution times.

We consider a *low priority* task τ_0 with phase $\phi_0 = 0$ and period $p_0 = 3$ and a *high priority* task τ_1 with $\phi_1 = 3.5$ and $p_1 = 5$. Their execution times c_0 and c_1 are both assumed to be 1. At the end of its execution time, τ_0 provides an output. The different behaviors are shown in Figure 3 and discussed below:

(i) When implemented as ordinary S-Function, the blocks execute in zero time and the output is provided instantaneously. The execution time of the tasks are ignored.

(ii) Although the output of τ_0 can be delayed by a constant time using a Unit-Delay block, the execution of the tasks does not consume simulation time. Consequently, the other tasks are not scheduled correctly and potential preemption is ignored. In addition, execution times may in general be data dependent and result from the control flow.

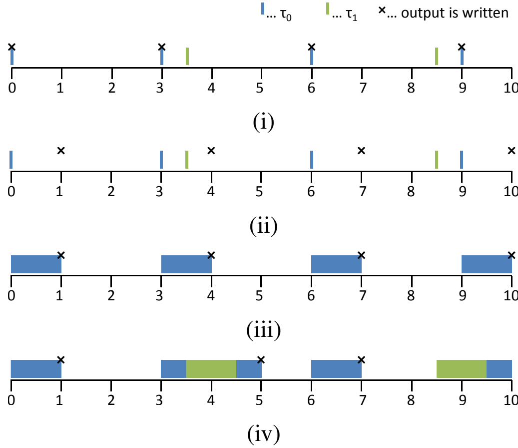
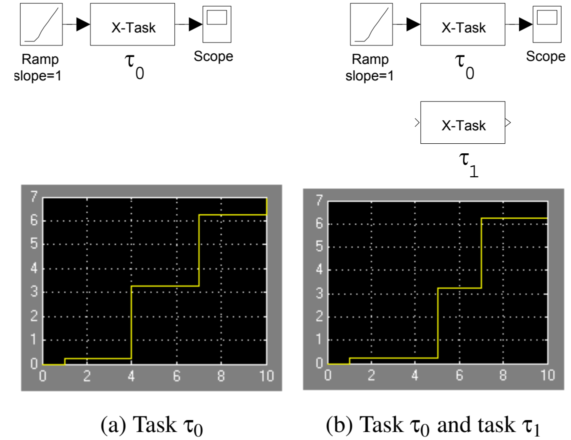


Figure 3: Scheduling example.



(a) Task τ_0 (b) Task τ_0 and task τ_1

Figure 4: The effect of task preemption in Simulink.

(iii) When implemented with BET semantics, τ_0 executes for $c_0 = 1$ and thus provides its output with the expected delay. For the scheduler, the execution task consumes simulation time. This becomes obvious in (iv).

(iv) When both tasks are present, τ_1 preempts τ_0 at 3.5 and the output of τ_0 is delayed by c_1 until 5. The 4th execution of τ_0 does not start until τ_1 completes at 9.5. This case best approximates the behavior expected when executing the two tasks on a single-processor system.

Figure 4a shows τ_0 as the only BET task in a Simulink model. The scope block shows the behavior described in (iii). Figure 4b shows the behavior (iv) when τ_1 is added.

4. CASE STUDY: AN AUTOMATIC TRANSMISSION CONTROLLER

In this section we apply the idea of a time consuming Simulink block to one of the demo models that come with Simulink.

The model is named *sldemo_autotrans.mdl* and is used to demonstrate Simulink in combination with Stateflow for modeling an automotive drivetrain. Originally, the gear selection for the transmission is modeled as a Stateflow diagram. We used the Real-Time-Workshop Embedded Coder to generate C code for the shift logic and replaced the Stateflow block by a BET task block executing the generated code. Figure 5 shows the Simulink model with the shift logic implemented as a BET task block named *Task_ShiftLogic*. The period of the task is equal to the original sample time (0.04 seconds) and the offset is 0.

We instrumented the controller code with API calls of our synchronization mechanism that also encode the time it takes the code to execute. For lack of real execution times of the individual code segments on a particular hardware platform, we applied the following strategy to arrive at a coarse estimate. The time to execute a particular basic block is the number of assembly instructions in the block multiplied by a constant factor. For now, we assume that all assembly instructions take the same amount of time, which is somewhat justified as simple RISC machines like the ARM7 finish an instruction and start another in almost each clock cycle. For demonstration purposes, the factor was chosen such that the maximum observed execution time of the controller consumes about 0.02*s*, which is 50% of the period.

Figure 6 shows a close up view of the EngineRPM signal to compare the original behavior and the behavior when considering execution time. As a consequence of the different states in the controller and the resulting differences in the taken execution paths, the execution time in each period varies. We observed task executions with the number of assembly instructions ranging from about 30 to more than 500, an effect that would be totally ignored in ordinary simulations. In total, we inserted almost 100 synchronization calls. In case other BET tasks are present, at every synchronization point higher priority tasks would preempt the shift logic controller.

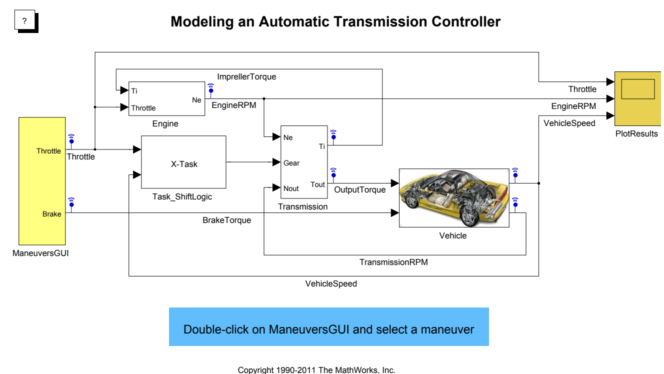


Figure 5: The modified automatic transmission controller model.

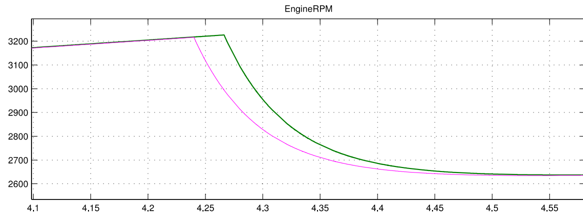


Figure 6: A close up view of the RPM signal with the original (red) Stateflow block and the BET task block (green).

5. RELATED WORK

The different programming paradigms discussed in this paper are in detail described in [11]. The Simulink integration of a LET-based software description language is presented in [15]. Numerous papers propose mechanisms to reduce the mismatch between the simulation and execution of SR models. We will confine ourselves to a few approaches related to Simulink. Several wait-free mechanisms have been proposed to ensure data consistency. The built-in Rate Transition (RT) block, for example, is able to guarantee data consistency for tasks with harmonic rates under rate monotonic scheduling [19]. More general buffering protocols are described in [20].

While these approaches retain the data flow character of Simulink, there is also a more software centric approach: TrueTime [6] is a toolbox for Simulink that enables 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. Currently, however, it requires the implementation code to be in a special format with multiple entry points, making it difficult to use for existing systems. In contrast to our approach, all TrueTime tasks are executed in the context of the kernel block, which consequently subsumes the whole control software. In our case, tasks are represented as individual blocks with their own input and output signals and they can appear at different positions in Simulink's sorted block order.

6. CONCLUSION

In this paper, we sketched different paradigms for real-time programming and contrasted their semantics to the synchronous reactive like semantic of the widely used MATLAB/Simulink environment. We presented a mechanism to emulate the individual approaches by introducing the concept of a Simulink block that is not characterized by the typical zero-execution time behavior but whose execution may last for and optionally consume a finite amount of simulation time. This reduces differences between the simulated control algorithm and its real code behavior. A rate monotonic scheduler implementation allowed us to simulate task preemption effects in multi-rate Simulink models.

REFERENCES

[1] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, Robert, and D. Simone. The synchronous languages 12 years later. In *Proc. of The IEEE*, 2003.

[2] G. C. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer Academic Publishers, Boston, Massachusetts, 2002.

[3] J. Eker, J. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. R. Sachs, and Y. Xiong. Taming heterogeneity - the Ptolemy approach. *Proc. of the IEEE, Special Issue on Modeling and Design of Embedded Software*, 91(1):127–144, January 2003.

[4] E. Farcas, C. Farcas, W. Pree, and J. Templ. Transparent distribution of real-time components based on logical execution time. *SIGPLAN Not.*, 40(7):31–39, 2005.

[5] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer, 1993.

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

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

[8] T. Henzinger, C. Kirsch, R. Majumdar, and S. Matic. Time safety checking for embedded programs. In *Proc. International Workshop on Embedded Software (EMSOFT)*, volume 2491 of *LNCS*, pages 76–92. Springer, 2002.

[9] T. Henzinger and J. Sifakis. The embedded systems design challenge. In *Proceedings of the 14th International Symposium on Formal Methods (FM), Lecture Notes in Computer Science*. Springer, August 2006.

[10] D. Iercan and E. Circiu. Modeling in Simulink temporal behavior of a real-time control application specified in HTL. *Journal of Control Engineering and Applied Informatics (CEAI)*, 10(4):55–62, 2008.

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

[12] J. Liu. *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[13] S. Mellor, K. Scott, A. Uhl, and D. Weise. Model-driven architecture. In J.-M. Bruel and Z. Bellahsene, editors, *Advances in Object-Oriented Information Systems*, volume 2426 of *Lecture Notes in Computer Science*, pages 233–239. Springer Berlin, Heidelberg, 2002.

[14] A. Naderlinger. Execution-time aware Simulink blocks. Poster, SpringSim'12, 2012.

[15] A. Naderlinger, J. Templ, and W. Pree. Simulating real-time software components based on logical execution time. In *SCSC '09: Proceedings of the 2009 Summer Computer Simulation Conference*, 2009.

[16] OSEK/VDX. Operating system specification 2.2.3, February 2005.

[17] T. W. Pearce. Simulation-driven architecture in the engineering of real-time embedded systems. In *Proc. of RTSS-WIP*, 2003.

[18] G. Stieglbauer. *Model-based Development of Embedded Control Software with TDL and Simulink*. PhD thesis, University of Salzburg, 2007.

[19] The MathWorks. Simulink, User's Guide, R2011b, 2011.

[20] G. Wang, M. D. Natale, P. J. Mosterman, and A. Sangiovanni-Vincentelli. Automatic code generation for synchronous reactive communication. In *Proc. of ICESS*, Washington, DC, USA, 2009. IEEE CS.