Simulation of LET Models in Simulink and Ptolemy

Patricia Derler, Andreas Naderlinger, Wolfgang Pree, Stefan Resmerita, and Josef Templ

C. Doppler Laboratory Embedded Software Systems
University of Salzburg
firstname.lastname@cs.uni-salzburg.at

Abstract. This paper describes two different approaches of simulating embedded control software whose real-time requirements are explicitly specified by means of the Logical Execution Time (LET) abstraction introduced in the Giotto project. As simulation environments we chose the black-box MATLAB/Simulink product and the open-source project Ptolemy II. The paper first sketches the modeling of LET-based components with the Timing Definition Language (TDL). As the LET abstraction allows the platform-independent modeling of the timing behavior of embedded software, a correct simulation of TDL components is equivalent to the behavior on a specific platform. We integrated TDL with both MATLAB/Simulink and Ptolemy and highlight the differences and similarities of the particular TDL simulation.

Keywords: simulation of real-time behavior, real-time modeling, Simulink, Ptolemy, Logical Execution Time (LET), Timing Definition Language (TDL).

1 Introduction

This work compares the way simulation environments with different goals facilitate the simulation of embedded control software modeled with the Timing Definition Language (TDL) [2]. TDL harnesses the Logical Execution Time (LET) abstraction for deterministically describing the timing behavior of a set of periodic tasks. If the tasks can be scheduled for a specific, potentially distributed platform given the worst-case execution time for each task for each computing node, the observable behavior of the system will be the same on the platform as in a simulation given the same inputs.

MATLAB/Simulink is a commercially available tool suite used to simulate control systems and also to generate C code. Simulink defines a fixed model of computation (MoC) that can only be adapted to some extent by means of so-called solvers as well as via the triggering of block executions. Ptolemy is an open-source simulation environment that serves as playground for experimenting with different MoCs and their combination in heterogeneous models.

We implemented the TDL model of computation in both, Simulink and Ptolemy, and describe the different integration approaches we had to take. We first introduce TDL.

C. Choppy and O. Sokolsky (Eds.): Monterey Workshop 2008, LNCS 6028, pp. 83–92, 2010.

Sections 2 and 3 sketch the core concepts how TDL components are modeled and simulated in Simulink and Ptolemy. Section 4 compares the two approaches.

Timing Definition Language (TDL). While TDL [2] is conceptually based on the LET-abstraction introduced in the Giotto project [3], it provides extended features, a more convenient syntax, and an improved set of programming tools. The LET associated with a computational unit, called task, represents the duration between the time instant when the task becomes ready for execution and the instant when the task terminates. A task's LET is specified independently of the task's functionality. When deploying the model on a platform, the LET specification is satisfied if the total physical execution time of the task is included in the LET interval for every task invocation, and an appropriate runtime system ensures that task inputs are read at the beginning of the LET interval (the release time) and task outputs are made available at the end of the LET interval (the termination time). Figure 1 illustrates the LET abstraction for one task invocation. Between release and termination points, the output values are those established in the previous execution; default or specified initial values are used during the first execution.

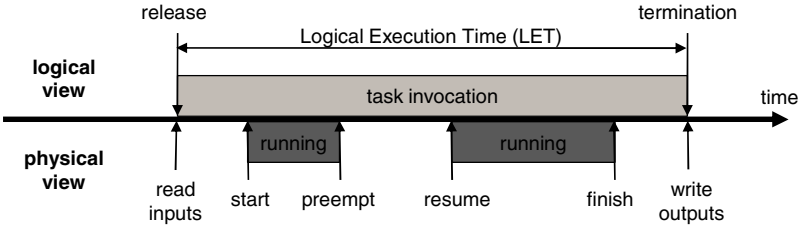

Fig. 1. Logical Execution Time (LET)

TDL is targeted at control applications consisting of periodic software tasks for controlling a physical environment. Thus, some tasks take information from the environment via sensors and some tasks act on the environment via actuators. Tasks that must be executed concurrently are grouped in modes. In TDL, a mode is a set of periodically executed activities that can be task invocations, actuator updates, and mode switches. A mode activity has a specified execution rate and may be carried out conditionally. TDL provides a top-level structuring unit called a module, which consists of sensors, actuators and modes that typically form a unit that delivers a specific functionality. A TDL module might be a complex combustion engine controller or a PID controller in a process automation system.

Figure 2 shows a schematic representation of a sample TDL module. The module contains one sensor variable *s1*, one actuator variable *a1*, and two modes called *main* and *freeze*. The mode *main* specifies a task invocation activity, an actuator update and a conditional mode switch, each of which must be executed once per mode period, which is every 5 milliseconds in this example. In other words, the task's LET is 5 ms. The actuator is updated with the task output value at the end of the LET. The mode *freeze* contains no activity at all.

```
module Sender {
 sensor int s1 uses getS1;
 actuator int a1 uses setA1;
 public task t1 {
  input int i;
  output int o := 10;
  uses t1Impl(i, o);
 }
 start mode main [period=5ms] {
  task [freq=1] t1(s1); //LET=5ms
  actuator [freq=1] a1 := t1.o;
  mode [freq=1] if exitMain(s1)
                then freeze;
 }
 mode freeze [period=1000ms] {}
}
```

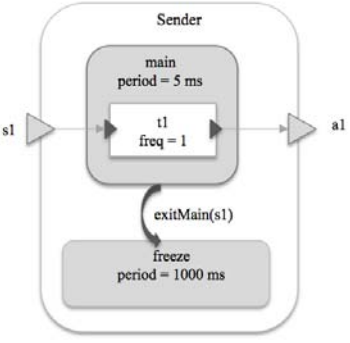

Fig. 2. TDL sample module

Executing TDL modules means executing all actions defined in the TDL code in the correct order at the specified points in time. A simulation environment for TDL modules must enforce the execution of these actions at the correct times and in this order. For a simulation time instant *t* those TDL actions are:

1. Update output ports of task invocations logically terminating at *t*.
2. Update actuators that are defined to be updated at *t*.
3. Test for mode switches that are defined at *t*. Switch mode if a switch is enabled.
4. Update input ports of tasks that are defined to be released at *t*.
5. Execute tasks that are defined to be released at *t*.

A general concept in simulating dynamic systems is that a simulation time is used to determine the actions that must be processed. TDL actions that are scheduled for a particular simulation time instant are executed without changing the time. If no actions remain, the simulation time is increased.

Consider the execution of task *t1* in the previously described example. At simulation time *0* actions in the start mode are processed. Output ports are initialized and connected actuators are updated. Sensor *s1* is read and the value is provided as input for the task, which is then executed. There are no more actions to be done at time *0*. Then the simulation time is increased to *5*, which is the end of the LET of *t1*. At simulation time *5*, output ports and actuators are updated. Next, the mode switch condition in the guard function *exitMain* is evaluated. If it evaluates to true, a mode switch to the empty mode *freeze* is performed and no further actions are processed. Otherwise the module stays in the mode and the task is executed again. The following sections describe how this general approach is implemented in Simulink and Ptolemy.

2 TDL Modeling in Simulink

The MATLAB extension Simulink from The MathWorks [7] was initially targeted for simulating control systems. It has significantly grown in popularity and due to numerous specific libraries is used for modeling systems ranging from control systems to artificial neural networks. It provides a visual, interactive environment for modeling block diagrams based on the data-flow paradigm. Simulink's model of computation is based on continuous time. This MoC is rather complex and there exists no formal definition; the implementation is hidden in the simulation engine [9, 10]. A straightforward modeling of TDL components with standard Simulink blocks is not feasible especially if they comprise several modes [6]. Simulink provides an extension mechanism by the so-called S-Function interface [8]. The subsequent section describes the Simulink integration of TDL by means of a customized S-Function and the corresponding model transformation.

2.1 LET Semantics in Simulink

The integration of TDL in Simulink requires both the modeling of TDL components, and their simulation adhering to TDL semantics. We implemented a custom Simulink block that represents a TDL module. As outlined in [6], modeling the TDL timing and data-flow relations especially for multiple modes is not feasible using standard Simulink facilities. Timing information is spread all over the model and the multitude of required signal connections makes it all but impossible to reason about or to maintain the model. It is unsolved how to obtain the exact TDL semantics in the general case of a multi-mode multi-rate system. Instead, we use a special purpose editor for describing timing aspects of the tasks, the data-flow between task ports, sensors and actuators, as well as the grouping of tasks to modes, and the mode switching logic. Tasks and guards are implemented in standard Simulink subsystems, which are referenced by the editor. In order to simulate a model, we automatically generate a simulation model consisting of standard Simulink blocks, and customized S-Function blocks.

Figure 3 shows the Simulink model for the sample TDL module from section 2. The model in Fig. 3a contains a TDL module block, a block representing the plant and a scope block to display actuator values. The content of the TDL module block in Fig. 3b shows subsystems for the task and guard function implementations and blocks for the sensor and the actuator. Fig. 3c depicts a sample implementation for the task *t1*.

To ensure that all TDL actions are executed at the correct time instants we implemented the concepts of E-Code and E-Machine in Simulink [11]. Henzinger et al. [4] introduced the E-Code concept in the realm of the Giotto project as a way of encapsulating the timing behavior and the reactivity of real-time applications. E-Code is a sequence of instructions for one period of every mode that describes the timing of all TDL actions. At run-time, these instructions are interpreted by a virtual machine, the E-Machine, which hands tasks to a scheduler or executes drivers. A driver performs communication activities, such as reading sensor values, providing input values for tasks at their release time, copy output values at their termination time, or updating actuators.

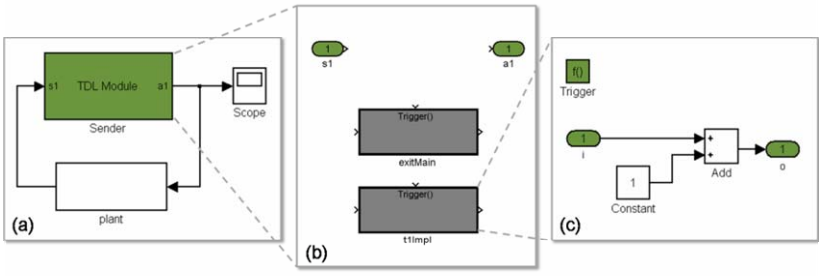

Fig. 3. (a) Simulink model with the TDL module block *Sender*, (b) contents of the Sender module block, (c) sample implementation of the t1Impl task function

For simulating the model, the TDL compiler [5] generates E-Code for all modules in the Simulink model. We implemented the E-Machine concept by using S-Functions. Drivers are generated and connected automatically. They are modeled as *Triggered Subsystems* where input ports are directly connected to output ports. This corresponds with assignments in an imperative programming paradigm as soon as the system is triggered. E-Machine blocks trigger the execution of these subsystems such that the TDL semantics are followed.

Figure 4 shows the generated simulation model for the TDL sample module *Sender*. It links with (a) the task- and guard functionality and (b) the plant model using *Goto* and *From* blocks. Section (c) contains the drivers (e.g. for reading sensor values or writing to actuators). As Simulink requires *static single assignment (SSA) form*, section (d) typically merges signals from drivers of different modes that write to the same port. As the sample module only executes activities in one single mode, signals are simply forwarded in this example. The remaining parts (e) and (f) together implement the 2-step E-Machine architecture described below.

2.2 Execution Mechanism

In typical application scenarios, TDL modules are simulated together with non-TDL controllers – typically modeled as atomic (nonvirtual) subsystems – or together with plants that don't introduce a delay. The original concept of the E-Machine [11] was adopted in order to avoid algebraic loops respectively data-dependency violations caused by Simulink's block execution strategy [12]. To solve the simulation model, Simulink derives a sorted block order based on data dependencies in the initialization phase. In this order, the simulation engine executes each block only once at a particular step. This approach is more efficient than using fixed point iteration (e.g. Ptolemy), but poses problems when trying to simulate closed control loops if no valid block order can be determined. In order to solve this and to allow Simulink to execute the plant or other blocks after actuators are updated and before sensors are read, we split duties of the E-Machine among two collaborating S-Functions (E-Machine 1 and E-Machine 2).

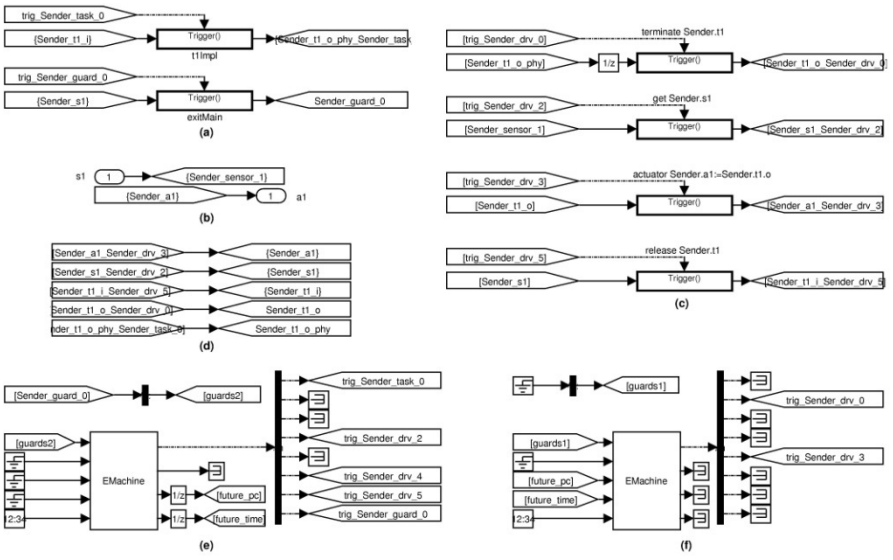

Fig. 4. The generated simulation model for the TDL module Sender in Simulink

The logical execution time of a task is controlled by the cooperating E-Machine pair. Additionally, a unit-delay block (1/z) is required for each task to indicate the fact that time passes between the task's release and terminate time. While the E-Machine ensures that the delay has effectively no impact on the timing behavior, Simulink needs it to derive a valid block update order and thus to be able to simulate the whole system.

Figure 4 show E-Machine 2 (e) and E-Machine 1 (f) with all triggers required for module *Sender*. Both E-Machines are executed at the same simulation time instants, but at different positions in the global block update order. At each simulation time, E-Machine 1 executes first to terminate tasks and update actuators, then the simulation engine executes the plant or other non-TDL blocks, and finally E-Machine 2 reads sensor values, decides on mode switches and releases tasks for the next execution. Each E-Machine executes only a subset of all drivers. Basically, E-Machine 1 executes sensor independent drivers such as task termination and actuator drivers, while E-Machine 2 executes task functions and potentially sensor dependent drivers such as mode switch or release drivers. In fact, the two E-Machines process different sections in the E-Code. We use *nop* (no operation) instructions together with an argument in order to separate the individual sections and to identify them during simulation.

E-Machine 2 signals mode switches to E-Machine 1 in order to resume with the right E-Code instructions at the next simulation step. A detailed description of the E-Machine architecture and its implementation is given in [13].

3 TDL Modeling in Ptolemy

Ptolemy II is the software infrastructure of the Ptolemy project at the University of California at Berkeley [1]. The project studies modeling, simulation, and design of

concurrent, real-time, embedded systems. Ptolemy II is an open source tool written in Java, which allows modeling and simulation of systems adhering to various models of computation (MoC). Conceptually, a MoC represents a set of rules, which govern the execution and interaction of model components. The implementation of a MoC is called a *domain* in Ptolemy. Some examples of existing domains are: Discrete Event (DE), Continuous Time (CT), Finite State Machines (FSM), and Synchronous Data Flow (SDF).

Ptolemy is extensible in that it allows the implementation of new MoCs. Most MoCs in Ptolemy support actor-oriented modeling and design, where models are built from actors that can be executed and which can communicate with other actors through ports. A Ptolemy model explaining the main Ptolemy entities is shown in Figure 5. The nature of communication between actors is defined by the enclosing domain, which is itself represented by a special actor, called the domain director. Simulating a model means executing actors as defined by the top-level model director.

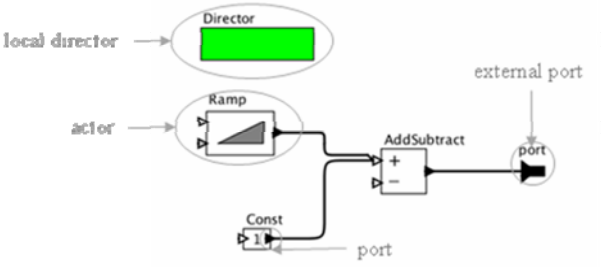

Fig. 5. Ptolemy II model

We implemented TDL as an experimental domain in Ptolemy. The implementation is based on the modal model variant of the Finite State Machine (FSM) domain in Ptolemy. Like modal models, TDL modules consist of modes with different behaviors, where only one mode can be active at a time. Transitions between states in modal models have the same behavior as mode switches in TDL.

The TDL domain consists of three specialized actors: TDLModule, TDLMode and TDLTask. The TDLModule actor (with the associated TDLModuleDirector) restricts the basic modal model behavior according to the TDL semantics. In modal models, mode switches are made whenever a mode switch guard evaluates to true whereas in TDL modules, mode switches are only allowed at predefined points in time. Similar restrictions apply to the port updates. To ensure LET semantics of the tasks, input ports of TDL tasks are only allowed to be read once at the beginning of the LET, output ports are only allowed to be written at the end of the LET and not when a task finished its computation. TDL requires a deterministic choice of simultaneously enabled transitions, which is not provided by the FSM domain. In this respect, we define an order on all transitions from a mode and take the first enabled transition in this order. TDL timing information such as the mode period is associated with TDL actors in the model.

TDL activities are conceptually regarded as discrete events that are processed in increasing time stamp order. Thus, a TDL module can be seen as a restricted DE actor. This enables the usage of TDL modules inside every domain that can deal with DE actors. The example from section 2 modeled in Ptolemy is shown in Figure 6.

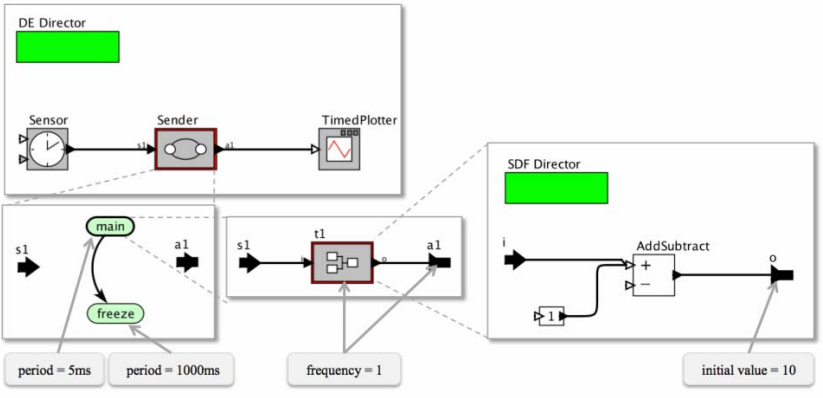

Fig. 6. A TDL module in Ptolemy

The model shown in the top left box of Figure 6 contains a TDL Module and two actors to provide sensor values and display actuator values. The TDL Module contains two modes (see Figure 6, the lower left box). Both modes have the period associated as a parameter. The main mode contains the task and the association of sensor and actuator values to input and output ports of the task. The frequency of task invocation, which determines the LET, is defined as a parameter. The task is an SDF actor, which executes in logically zero time. The top-level director is a DE director. The DE director uses a global event queue to schedule the execution of actors in the model. The TDL module places events in this queue for every time stamp where at least one TDL action is scheduled.

4 Summary and Conclusions

We presented the modeling and simulation of TDL real-time components with logical execution time (LET) in two different simulation environments, namely Simulink and Ptolemy. Due to fundamental differences in the simulation environments we had to apply two different approaches regarding both, modeling and simulation. Table 1 lists the main concepts in modeling and simulation of TDL modules and how they were implemented in Simulink and Ptolemy.

The open and extensible architecture of Ptolemy allowed us to express the complete TDL semantics in the model. In the Simulink integration, the developer only models the functionality (task implementations and the plant) with Simulink blocks. Timing, mode switching logic and the overall application data-flow are described with a special purpose editor. An elaborate model transformation automatically creates a simulation model that contains data-flow and timing information.

Both approaches extend the existing simulation framework with a new actor (block) containing tasks with real time requirements described by TDL. In Simulink, the timing requirements are enforced by the E-Machine, in Ptolemy the director of the TDL domain employs the TDL semantics.

Table 1. Comparison of TDL Modeling and Simulation in Simulink and Ptolemy

	Simulink	**Ptolemy**
Modeling		
TDL Module	A TDL module block (actor) was implemented and is available in a library.	
TDL Mode	Modes are defined in the TDL editor.	A TDL mode actor was implemented and is available in a library. A TDL mode contains input and output ports of a TDL module (=sensors, actuators) and TDL tasks.
Data-flow (Connections between sensors, task ports, and actuators)	Data-flow is defined in the TDL editor.	Graphically connect TDL module ports to task ports.
Task functionality	An embedded model (subsystem) implements the task functionality.	
	Stub subsystems are generated, which provide the input and output ports. They have to be implemented with Simulink blocks.	A TDL task actor was implemented and is available in a library. The TDL task is a composite actor containing the embedded model.
Simulation		
Triggering of all actors (blocks) in the model (TDL modules and plant)	Simulink triggers all blocks of the plant and the E-Machine S-Function blocks.	The top-level director, which has to be able to deal with DE actors, triggers all actors including TDL modules.
Timing description of TDL actions	Generate static E-Code before starting the simulation.	Generate events dynamically during the simulation.
Enforcing the timely execution of TDL actions	The E-Machine interprets E-Code instructions and triggers TDL actions.	The TDL Director creates events for all TDL actions.

Ptolemy uses events to schedule TDL activities dynamically whereas in Simulink, we compute a static list of TDL activities in the form of E-Code before starting the simulation. The main advantage of the static approach is its low computational overhead for determining the next TDL actions. Maintaining an E-Code program counter is enough, it is not necessary to create events and handle dynamically changing event queues. Event queues, on the other hand, potentially require less storage space than E-Code, because they contain only the immediate follow-up events, not all activities of a mode period.

References

[1] Brooks, C., Lee, E.A., Liu, X., Neuendorffer, S., Zhao, Y., Zheng, H. (eds.): Heterogeneous Concurrent Modeling and Design in Java, Introduction to Ptolemy II. vol. 1. EECS Department. University of California, Berkeley, UCB/EECS-2007-7 (January 2007)
[2] Templ, J.: TDL - Timing Definition Language 1.4 Specification (2007), http://www.preetec.com/
[3] Henzinger, T.A., Kirsch, C.M., Sanvido, M., Pree, W.: From Control Models to Real-Time Code Using Giotto. IEEE Control Systems Magazine 23(1) (2003)
[4] Henzinger, T.A., Kirsch, C.M.: The Embedded Machine: Predictable, portable real-time code. In: Proc. of the PLDI. ACM Press, New York (2002)
[5] preeTEC, http://preetec.com/
[6] Pree, W., Stieglbauer, G.: Visual and Interactive Development of Hard Real Time Code. In: Automotive Software Workshop San Diego (ASWSD) (January 2004)
[7] The MathWorks, http://www.mathworks.com
[8] The MathWorks. Simulink 7, Writing S-Functions (2008)
[9] Carloni, L., Benedetto, M.D.D., Passerone, R., Pinto, A., Sangiovanni-Vincentelli, A.: Modeling Techniques, Programming Languages and Design Toolsets for Hybrid Systems, Project IST-2001-38314 COLUMBUS-Design of Embedded Controllers for Safety Critical Systems, WPHS: Hybrid System Modeling, version: 0.2, Deliverable number: DHS4-5-6 (July 2004)
[10] Baleani, M., Ferrari, A., Mangeruca, L., Sangiovanni-Vincentelli, A.L., Freund, U., Schlenker, E., Wolff, H.-J.: Correct-by-Construction Transformations across Design Environments for Model-Based Embedded Software Development. In: Proc. of the Conf. on Design, Automation and Test in Europe, DATE (2005)
[11] Stieglbauer, G.: Model-based Development of Embedded Control Software with TDL and Simulink. PhD thesis, Department of Computer Sciences. University of Salzburg, Austria (2007)
[12] Mosterman, P.J., Ciolfi, J.E.: Interleaved execution to resolve cyclic depen-dencies in time-based block diagrams. In: Proc. of the 43rd IEEE Conference on Decision and Control, CDC (2004)
[13] Naderlinger, A., Templ, J., Pree, W.: Simulating Real-Time Software Components based on Logical Execution Time. In: SCSC 2009: Proceedings of the 2009 Summer Computer Simulation Conference (2009)