5 Modeling and Simulation of TDL Applications

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

University of Salzburg, Austria
{stefan.resmerita,patricia.derler,wolfgang.pree,
andreas.naderlinger}@cs.uni-salzburg.at

Abstract. Most of the existing modeling tools and frameworks for embedded applications use levels of abstraction where execution and communication times of computational tasks are not captured. Thus, properties such as time and value determinism can be lost when refining the model closer to a target platform. The Logical Execution Time (LET) paradigm has been proposed to deal with this issue, by enabling specification of platform-independent execution times of periodic time-triggered computational tasks at higher levels of abstraction.

This chapter deals with modeling and simulation of embedded applications where LET requirements are specified by using the Timing Definition Language (TDL). TDL provides a programming model for time- and event-triggered components suitable for large distributed systems. We present specific TDL extensions that increase the expressiveness of the language, accommodating the needs of control applications such as minimum sensor-actuator delays. We describe simulation of TDL programs in dataflow models (using Simulink) and discrete event (DE) models (using Ptolemy II). We show how the Ptolemy II based simulation can be used to validate preservation of timing and value behaviors when mapping a DE model of an application with concurrent components into a sequential implementation platform with fixed priority preemptive scheduling.

5.1 Introduction

In complex embedded systems, execution and communication times related to computational tasks of an application can have a substantial influence on the application behavior that is unaccounted for in high level models. Consequently, the implementation of a model on a certain execution platform may violate requirements that are proved to be satisfied in the model. Explicitly considering execution times at higher levels of abstractions has been proposed as a way to achieve satisfaction of real-time properties [1]. One promising direction in this respect is the Logical Execution Time (LET) [2], which forms the foundation of several real-time programming languages [2] [3] [4]. Among these, the Timing Definition Language [4] is under active development, with commercially available support tools.

TDL is inspired from the Giotto programming model [2], which was targeted for control applications. Giotto proposed trading end-to-end latency (which

H. Giese et al. (Eds.): MBEERTS, LNCS 6100, pp. 107–128, 2010.
© Springer-Verlag Berlin Heidelberg 2010

must be minimal in control systems) for determinism and robustness, which have become more and more important due to increased complexity of both applications and platforms. TDL has been extended to address both requirements, in the commonly encountered case where a control task can be split into a *fast step*, used to compute controller outputs (i.e. actuator values) and a slow step, used to determine new state information. Another TDL extension, called *slot selection*, allows designation of LET values that are smaller than the invocation period for a task, providing a separation of concerns between choosing the controller sampling period, which is the job of the control engineer, and minimizing computation latency, which is the job of the software engineer. By slot selection, the designer specifies the beginning and end of a task's LET within the task's invocation period. This also enables specification of executions with fixed time offsets.

An important principle in embedded systems design is the separation between the functionality of an application and the platform where the application is implemented. This principle is adopted by modern design methodologies such as Model Driven Architecture (MDA) [5] and Platform Based Design (PBD) [6]. An MDA application consists of a platform-independent model (PIM), specifying the functionality, one or more platform-specific models(PSMs), and sets of interfaces describing the coupling between PIM and each of the PSMs. PBD proposes an iterative model-based development process where at each iteration a functionality model is mapped to a platform model. The mapped functionality becomes the functionality model for the next iteration, where it is mapped to a (usually refined) platform model. This is repeated until the final implementation is obtained for all components.

One of the main issues in the above approaches is the mapping between the functionality model and the platform model, which should be done such that the behaviors of the resultant model are in the intersection of the behavioral sets of the individual models. A commonly encountered situation is mapping a concurrent functional model into a sequential implementation platform. For a real-time application, this may lead to violation of real-time properties such as maximum sensor-actuator delays. Even if a suitable scheduling guarantees latency bounds, it may not guarantee the same value behavior as in the functionality model (a simple example of this situation will be further discussed in this paper). Using LET ensures preservation of timing and value behaviors over model refinement, by requiring that the platform model has the means to carry out the LET semantics, which refers not only to task execution (resolved by scheduling), but most importantly to data transfer (resolved by buffering). In the final implementation, LET specifications are executed by a dedicated runtime system, provided that the software components can be suitably scheduled for execution. Scheduling can be done statically for the time-triggered, periodic task executions.

Static schedulability analysis for LET-based software becomes hard to achieve, or overly conservative, in embedded control systems containing also concurrent computations triggered by environment conditions (dynamic events), where

event-triggered tasks share the same execution platform as the time-triggered part, and may preempt or delay the execution of time-triggered components. Thus, a simulation platform is needed for early verification and validation of such heterogeneous systems. Existent simulation frameworks for LET-based models operate at a functional (platform independent) level, where the most influential LET benefits cannot be shown. For example, a DE model of a time-triggered application with LET-based constraints has the same behavior as a model where the LET constraints are replaced by delays. However, when mapping the functional model into a platform model, the mapped delay-based model may exhibit new behaviors, while the behaviors of the mapped LET-based model will always be included in the behavioral set of the functional LET-based model.

In the embedded systems industry, simulation is widely used for testing and validation of complex systems. It is also used for effectively demonstrating the impact of new technologies. It is therefore important to be able to simulate a model with LET specifications in order to demonstrate the benefits of the LET approach. In this paper, we consider a platform abstraction consisting of execution times and fixed priority preemptive scheduling. Thus, the mapping means assigning to each task in the functional model an execution time and a priority. We present a Ptolemy II framework which allows simulating the behavior of a LET-based application mapped to the platform. TDL is employed to specify the timing structure based on LET. We use the simulator to run an example which shows how TDL can ensure preservation of behavior over model refinement.

This paper is structured as follows. Section 5.2 describes the Timing Definition Language, including the control-specific extensions. In Section 5.3 we present the two main simulation frameworks for TDL. The relations with existing work are shown in Section 5.4, which is followed by concluding remarks in Section 5.5.

5.2 The Timing Definition Language

We describe in this section the main constructs of the core TDL, followed by extensions of the language that specifically address control applications. The ensuing presentation of TDL components is necessarily brief. The complete TDL specification can be found in [4].

5.2.1 TDL Description

The Timing Definition Language allows the specification of timing properties of hard real-time applications by employing the LET concept and the principle of separation between timing and functionality introduced in Giotto [2]. While TDL is conceptually based on Giotto, it provides extended features, a more convenient syntax, and an improved set of programming tools.

The Logical Execution Time associated with a computational unit, or task, represents a fixed logical duration between the time instant when the task becomes ready for execution and the instant when the execution finishes. A task's LET is specified at the model level, 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). This is illustrated in Figure 5.1. 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.

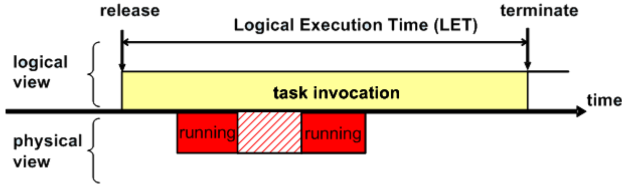


Fig. 5.1. Logical Execution Time

TDL is targeted for applications consisting of periodic software tasks designed to control a physical environment. Thus, some tasks receive information from the environment via sensors and some tasks act on the environment via actuators. A task has input ports, output ports, and state ports. State ports are employed for maintaining state information between different executions of the same task. The main structure of a task declaration in TDL is given in Figure 5.2.

```
task <task_name> {
    input <type> <list_of_input_ports>;
    ...                            //other input port declarations
    output <type> <list_of_output_ports>;
    ...                            //other output port declarations
    state <type> <list_of_state_ports>;
    ...                            //other state port declarations
    uses <external_function_call>;
}
```

Fig. 5.2. Structure of TDL task declaration

Any of the lists of ports can be empty, while exactly one external function name (possibly with arguments) must be specified after the "uses" keyword. This represents the implementation of the task functionality.

Tasks that are executed concurrently are grouped in modes. In TDL, a mode is a set of periodically executed activities: task invocations, actuator updates, and mode switches. A mode activity has a specified execution rate and may be carried out conditionally. A mode declaration is schematically shown in Figure 5.3. The frequency attribute specifies the rate of execution of the corresponding

```
mode <mode_name> [period=<time_duration>]{
 task
   [freq=<exec_rate>] <task_name>(<argument_list>);
   ...                                  //other task invocations
 actuator
   [freq=<exec_rate>] <act_name>:=<task_name>.<output_port>;
   ...                                  //other actuator updates
 mode
   [freq=<exec_rate>] if <condition> <name_of_target_mode>;
   ...                                  //other mode switches
}
```

Fig. 5.3. Structure of TDL mode declaration

activity within one mode period. Thus, the LET of a task is expressed as the mode period divided by the frequency of task invocation. Note that the time steps of all activities in a mode period can be statically determined. Mode activities are carried out by a runtime system which performs the following operations at every time step:

(1) Update output ports of tasks whose LETs end at the current time step. At time 0, the ports are initialized rather than updated.
(2) Update actuators.
(3) Test for mode switches. If a mode switch is enabled, switch to the target mode.
(4) Update input ports of the tasks whose LETs start at the current time step.
(5) Trigger the execution of the tasks whose LETs start at the current time step.

TDL provides a top level structuring unit called a *module*, which is a logically coherent group of sensors, actuators and modes. The module concept serves multiple purposes: (1) a module provides a name space and an export/import mechanism and thereby supports decomposition of large systems, (2) modules provide parallel composition of real-time applications, (3) modules serve as units of loading, i.e. a runtime system may support dynamic loading and unloading of modules, and (4) modules are the natural choice as unit of distribution because dataflow within a module (cohesion) will most probably be much larger than dataflow across module boundaries (adhesion).

A schematic example of a TDL program is shown in Figure 5.4. Notice that a module contains declarations of sensor and actuator variables, tasks and modes. In the above example, module Sender contains a sensor variable $s1$, and an actuator variable $a1$. The value of $s1$ is updated by executing the (platform-specific) driver $getS1$ and the value of $a1$ is send to the physical actuator by using the platform specific driver $setA1$. Each module has exactly one start mode, indicated by preceding the mode declaration with the reserved word "start". The declaration of the output port of task inc specifies also an initial value (10). The task is invoked in mode $main$ of the $Sender$ module, where its input port is connected to the sensor $s1$. In the same mode, actuator $a1$ is updated with the

value of the task's output port. The second module called *Receiver* imports the *Sender* module in order to connect the input of the local task *clientTask* with the output of the external task *inc*. These TDL components and their connectivity are depicted in Figure 5.5.

Let us illustrate the operations carried out by the TDL runtime system for the task *inc* during one mode period. At time 0, 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 released for execution. At time 5 (the end of the LET), the task's output port is updated, then actuator `a1` is 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 mode *main* remains active and the above operations are repeated in the next mode period.

TDL enables so-called *transparent distribution* of hard real-time applications, which can be described with respect to two points of view. Firstly, at run-time a TDL application behaves exactly the same, no matter if all modules

```
module Sender {
  sensor int s1 uses getS1;
  actuator int a1 uses setA1;
  public task inc {
    input int i;
    output int o := 10;
    uses incImpl(i,o);
  }
  start mode main [period=5ms] {
    task [freq=1] inc(s1);              //LET = 5ms (=period/freq)
    actuator [freq=1] a1 := inc.o;
    mode [freq=1] if exitMain(s1) then freeze;
  }
  mode freeze [period=1000ms] {}
}
module Receiver {
  import Sender;
  . . .
  task clientTask {
    input int i1;
    . . .
  }
  start mode main [period=10ms] {
    task [freq=1] clientTask(Sender.inc.o);         //LET = 10ms
    . . .
  }
  . . .
}
```

Fig. 5.4. Example of TDL code

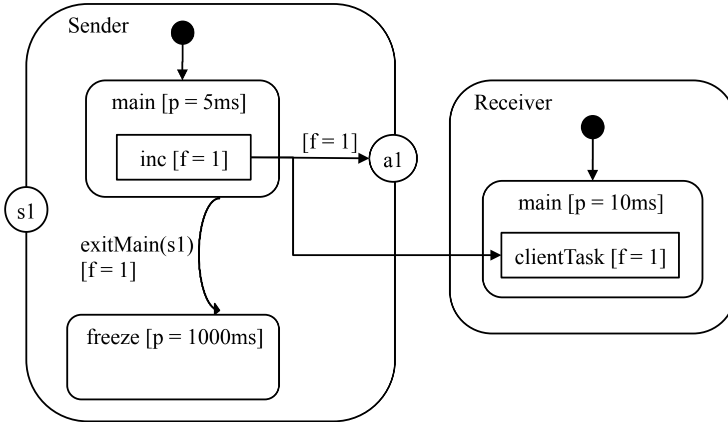


Fig. 5.5. TDL constructs defined by the code in Figure 5.4

(i.e. components) are executed on a single node or if they are distributed across multiple nodes. The logical timing is always preserved, only the physical timing, which is not observable from the outside, may be changed. Secondly, for the developer of a TDL module, it does not matter where the module itself and any imported modules are executed. The TDL tool chain and runtime system frees the developer from the burden of explicitly specifying the communication requirements of modules. It should be noted that in both aspects transparency applies not only to the functional but also to the temporal behavior of an application. The advantage of transparent distribution for a developer is that the TDL modules can be specified without having the execution on a potentially distributed platform in mind. The functional and temporal behavior of the system is independent of the mapping of modules to computation nodes, which is defined separately.

A compiler transforms TDL programs into virtual instructions called *E-Code* [7]. E-Code describes the application's reactivity, i.e. time instants to release or terminate tasks or to interact with the environment. A virtual machine, the *E-Machine* [7], interprets the instructions at runtime and ensures the correct timing behavior. According to the E-Code, the E-Machine timely hands tasks to a dispatcher and executes drivers. A driver performs communication activities, such as reading sensor values, providing input values for tasks at their release time or copy output values at their termination time.

A commercially available tool suite deals with modeling and deployment of TDL components [8]. TDL components can be written directly in textual form (TDL source code) or designed graphically by using the TDL:VisualCreator tool. The TDL:Compiler targets the TDL:E-Machine. The TDL:E-Machine exists for several different platforms, including OSEK, INtime, RTLinux, etc. The TDL:VisualDistributor can be used to assign TDL modules to a single specified computational node or a distributed system of nodes. Also, the TDL:Scheduler is employed to generate the necessary node and communication schedules. The

tools also check for the schedulability of the system, based on provided worst case execution times for the tasks, under the assumption that the periodically time-triggered TDL tasks are the only significant computations competing for the platform resources.

5.2.2 TDL Extensions for Control Applications

Reducing Latency for Control Applications

The main application field for the time-triggered programming model introduced by Giotto is implementation of control systems. A control application reads environment data through sensors, and exercises control over the physical environment through actuators. In sampled data control systems, the controller is executed periodically, polling sensors and determining control actions in every period. Usually, control actions depend on the latest sensor values and on the current state of the controller, which is also updated at every period. The time delay between reading sensors and updating actuators in the same period should be as small as possible. Thus, the controller functions are organized in two steps: *update outputs* and *update state*, with the first step to be executed as soon as possible after sensor reading. To enable advance calculation of control outputs, in TDL a task's functionality code can be split in a *fast step* (corresponding to *update outputs*) and a *regular step* (corresponding to *update state*), where the fast step is executed in logical zero time at the release time of the TDL task, and the regular step is executed within the task's LET. To this end, the task declaration is modified to allow specification of *two* external function calls, the fast one being indicated by a dedicated driver annotation called "[release]", which means that the fast function has to be executed immediately when the task is released for execution (i.e. at the beginning of the task's LET). A two-step task can now be declared according to the structure shown in Figure 5.6. Syntactically, the only addition to the single-step task declaration (shown in Figure 5.2) is another *uses* line containing the *release* annotation, which is reserved for the fast step declaration. If an output port appears in the argument lists of both functions, then it acts as output of the fast function (i.e. it must be updated by the fast function) and as input to the slow function. An example is presented in Figure 5.7.

```
task <task_name> {
    input <type> <list_of_input_ports>;
    ...                            //other input port declarations
    output <type> <list_of_output_ports>;
    ...                            //other output port declarations
    state <type> <list_of_state_ports>;
    ...                            //other state port declarations
    uses [release] <fast_function_name>(<arg_list>);
    uses <slow_function_name>(<arg_list>);
}
```

Fig. 5.6. Structure of TDL declaration for a two-step task

```
task digiCon {
  input int i1,i2;
  output int o:=0;
  state double s:=0;
  uses [release] controllerOutput(i1,i2,s,o);
                                  //o must be calculated here
  uses controllerUpdate(i1,i2,s,o);
                                  //o is an input argument here
}
```

Fig. 5.7. Declaration example of a two-step task

The explicit declaration of a task's fast and slow steps is accompanied by the introduction of a specific mode activity, called *task sequence*, to indicate actuator updates that must take place upon execution of the task's fast step. A task sequence combines a task invocation and subsequent actuator updates. These are performed at the release time of the invoked task, if the task contains a fast step that provides the required output ports. Output ports updated in the fast step are available immediately for actuator updates if the two-step task is included in a task sequence. Figure 5.8 presents the layout of a mode declaration including task sequences. An example where the task in Figure 5.7 appears in a task sequence is shown in Figure 5.9. The effect of this code is that at every 10ms, sensors `s2` and `s3` are read, the function *controllerOutput* is executed and the actuator `act1` is updated. Since these operations are considered as taking logical zero time, their execution times must be much smaller than the execution times of regular TDL tasks. Then the function *controllerUpdate* is executed, which may take up to 10ms. Task `t0` is a regular TDL task with a LET of 50ms. Thus, at every 50ms tick, sensor `s1` is read and task `t0` is released for execution. The output of `t0` is provided to actuator `act2` at the end of the 50ms period.

```
mode <mode_name> [period = <time_duration>]{
 task
   [freq=<exec_rate>] <task_name>(<arg_list>);
   [freq=<exec_rate>] {<task_name>(<arg_list>);
                        <act_name>:=<task_name>.<output_port>;}
   ...                              //other task invocations
 actuator
   [freq=<exec_rate>] <act_name>:=<task_name>.<output_port>;
   ...                              //other actuator updates
 mode
   [freq=<exec_rate>] if <condition> <name_of_target_mode>;
   ...                              //other mode switches
}
```

Fig. 5.8. Structure of TDL mode declaration with task sequence

```
start mode main [period=100ms] {
  task [freq=2] t0(s1);
  task [freq=10] {digiCon(s2, s3); act_1:=digiCon.o;} //sequence
  actuator [freq=2] act_2 := t0.o;
}
```

Fig. 5.9. Example of task sequence

Task sequences entail a specific operational semantics. The operational steps performed by the runtime system are now as follows:

(1) Update output ports of tasks whose LETs end at the current time step. At time 0, the ports are initialized rather than updated. Exception: output ports of two-step tasks that are arguments of both functions (fast and slow) are not updated.
(2) Update actuators.
(3) Execute fast tasks. For every task sequence that occurs at the current step, update the inputs of the task, then execute the fast function, then update output ports and connected actuators as specified in the sequence.
(4) Test for mode switches. If a mode switch is enabled, switch to the target mode.
(5) Update input ports of the tasks whose LETs start at the current time step, except for those inputs already updated at step 3.
(6) Trigger the execution of the regular tasks whose LETs start at the current time step. Also, for every task sequence that occurs at the current step, trigger the execution of the slow function.

Increasing Control of Time-Triggered Activities

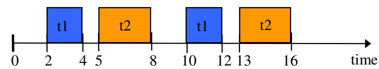
In TDL, the user can specify the endpoints of a task's LET within the task's invocation period. Thus, as opposed to Giotto, a task's LET may be different (i.e. smaller) than the task's period. TDL can express time-triggered executions such as the one in Figure 5.10b, which shows two tasks with the same invocation period of 8ms and a fixed offset of 3ms. TDL employs Giotto's syntax to specify a task's invocation period, by using a mode period p and a frequency f of task invocation within p. Thus, if the LET of a task equals its period of invocation, then the task's LET is p/f. TDL uses the additional feature of *slot selection* to allow the LET of any individual task invocation to be defined more explicitly as an interval that starts and ends at integer multiples of p/f. Thus, a task's LET corresponds to a *slot group*. The slots are numbered from 1 to p/f. TDL offers a compact syntax for specifying a task's slot groups within a mode period, as follows. A repeating pattern of slot groups is specified by using the character "*" after the pattern. A slot group can be optional, which means that the corresponding task execution may be skipped at runtime, if this helps in finding a feasible schedule. Some examples are:

`slots=1*` : all slots are mandatory and LET=p/f; this is the default.
`slots=~1|2*` : LET=p/f, the first slot is optional and the remaining slots are mandatory.
`slots=1-3*` : mandatory slot groups with LET=3*p/f each.

Figure 5.10a shows the specification of the execution pattern depicted in Figure 5.10b.

```
start mode main [period=8ms] {
    task [freq=4,slots=2] t1();
    task [freq=8,slots=6-8] t2();
}
```

(a) TDL code with slot selection



(b) Execution pattern with offsets

Fig. 5.10. Slot selection example

5.3 Simulation of TDL Models

Simulating TDL models means executing the operations described above on an executable model in a simulation platform rather than a physical execution platform. TDL is currently supported in two modeling and simulation frameworks: Simulink and Ptolemy II.

5.3.1 TDL Simulation in Simulink

The MATLAB extension Simulink from The MathWorks [9] is a widely used environment for modeling, simulating and analyzing dynamic and embedded systems. Simulink is based on the data flow programming paradigm and provides an interactive graphical interface. Together with automatic code generators such as the Real-Time Workshop (Embedded Coder), it has become the de-facto standard, particularly in the automotive domain.

Overview

Modeling TDL components manually with standard Simulink blocks is not feasible [10]. Typically, control systems involve multiple modes [11]. Depending on the current mode, the application executes individual tasks with different timing constraints or even changes the set of executed tasks. At the latest when mode switching logic and multiple execution rates come into play, it is all but impossible to understand or maintain the model. Instead, we use an automatic model generation approach to ensure TDL semantics in Simulink. Therefore, the TDL tool chain was extended and integrated in MATLAB/Simulink to model and simulate TDL applications and to support the code generation for particular, potentially distributed, hardware platforms.

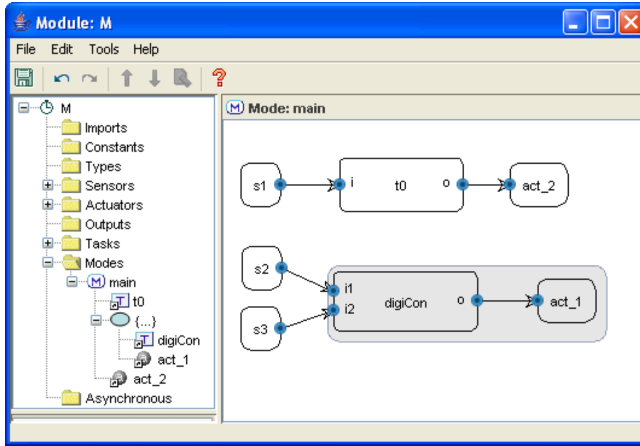


Fig. 5.11. The TDL:VisualCreator tool in Simulink

Modeling TDL in Simulink

The plant and the task respectively guard functionality is modeled with regular Simulink blocks, whereas the timing behavior, i.e. the TDL description, is specified by means of the TDL:VisualCreator tool. This graphical modeling tool is a syntax driven editor that is integrated via the *TDL Module Block* as part of the *TDL Simulink library*. Figure 5.11 shows the TDL:VisualCreator and a module `M` that corresponds to the mode declaration in Figure 5.7.

The activities in mode `main` are shown on the right, where the task sequence is indicated with the gray container that groups task `digiCon` and actuator `act_1`. Individual TDL constructs are created and managed using the tree view on the left. For each sensor (`s1`, `s2`, `s3`) and actuator (`act_1`, `act_2`) a corresponding Simulink `Inport` respectively `Outport` is automatically created for the module block. For each task (`t0`, `digiCon`), the tool generates a Simulink subsystem that may then be implemented by the control engineer. Again, Inport and Outport blocks are used to represent the task ports.

Simulating TDL in Simulink

For the simulation, we apply a model transformation with an E-Machine implementation for Simulink at its core. Drivers are automatically generated as *function-call subsystems* and are connected via Simulink signals. We implemented an E-Machine using the S-Function mechanism provided by Simulink to timely trigger their execution and thus to ensure TDL semantics. Therefore, the TDL:Compiler generates E-Code from the TDL description which is then interpreted by the E-Machine during the simulation.

Figure 5.12 shows the generated Simulink model for module `M`. The gray blocks for tasks (a) and sensors respectively actuators (b) were already generated by the TDL:VisualCreator during the modeling process. They now get linked with the rest of the newly generated model using Simulink's `Goto` and `From` blocks.

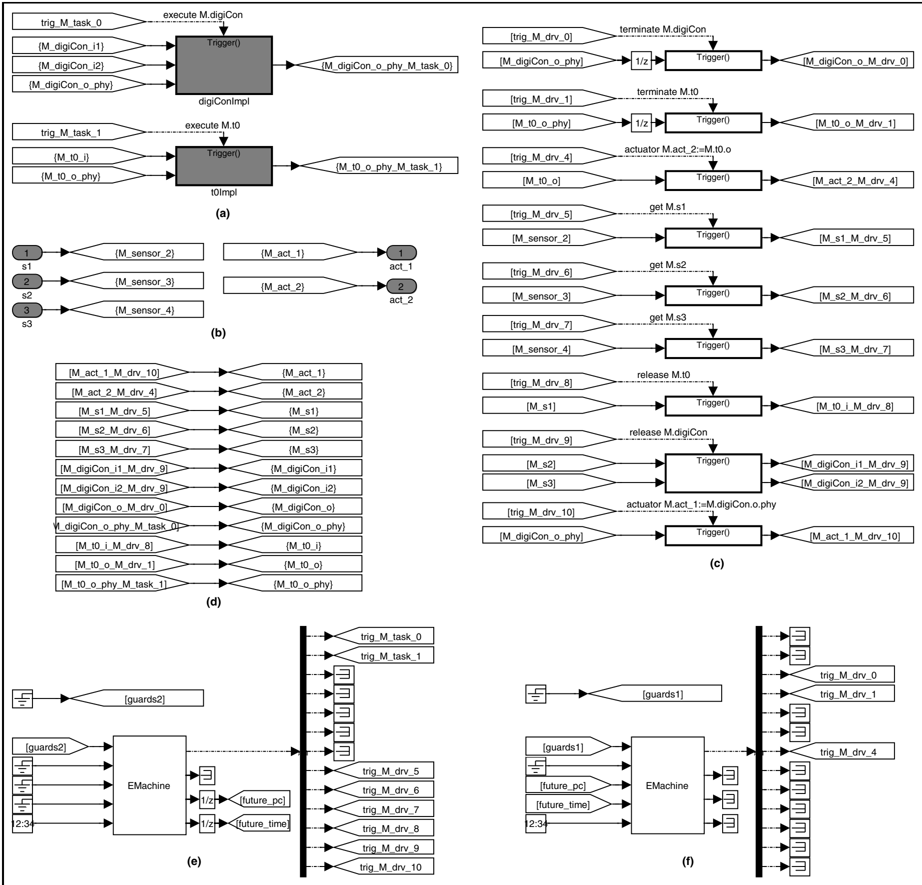


Fig. 5.12. An automatically generated TDL simulation model in Simulink

Section (c) contains the drivers (e.g. for reading sensor values or writing to actuators). The input ports of a driver are directly connected to the output ports, which corresponds with assignments in an imperative programming paradigm as soon as the system is triggered. Section (d) merges signals from drivers of different modes that write to the same port (static single assignment). As module M has only one mode, signals are simply forwarded in this example. Section (e) and (f) together implement a 2-step E-Machine architecture [12], which triggers the execution of drivers, tasks, and guards. To avoid restrictions on the set of supported blocks (e.g. for the plant) caused by Simulink's block execution strategy, we split duties of the E-Machine among two collaborating S-Functions. This allows Simulink to execute the plant or other blocks after actuators are updated and before sensors are read. Delay blocks between the release and the termination driver of a task and between the two E-Machines do not affect the timing behavior. They are, however, required to enable Simulink to resolve algebraic (feedback) loops that typically arise when simulating plants without delay

or when combining LET-based with conventional controllers that are modeled as atomic (nonvirtual) subsystems [12].

Code generation

Once the simulation exhibits satisfactory behavior, one can go about generating code. Therefore, the TDL:VisualDistributor tool, which is also integrated in Simulink, may be used to define a hardware topology and to map the TDL modules to their target nodes. This also requires to specify worst-case execution times and hardware devices for sensors respectively actuators. A flexible plugin-based code generation framework generates the required C code and, in case of a distributed system, the required communication schedule. The TDL tool chain employs MathWork's Real-Time Workshop Embedded Coder [9] to generate C code for the control task functionality. For supporting the *fast step* extension, we make use of the possibility to split a Simulink task function implementation into an Output (fast step) and an Update (slow step) function. The generated code can then be compiled and linked with the platform specific E-Machine.

The main advantage of the E-Machine implementation for Simulink is that both the simulation environment and the target platform execute the same E-Code. This is a strong indicator (albeit no proof) that the simulation and the execution of TDL modules exhibit exactly the same behavior.

5.3.2 Using Ptolemy II

Ptolemy II is the software infrastructure of the Ptolemy project at the University of California at Berkeley [13]. 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.

Overview of Ptolemy II

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. An actor is represented by a Java class that implements the actor interface. The nature of communication between actors is defined by the enclosing domain, which is itself represented by a special actor, called the domain director. A model may define an external interface that enables it to be regarded as an actor with input and output ports. Figure 5.13 shows a sample Ptolemy model. The green block represents the local director which enforces the model of computation used in the model. The model also contains actors with input ports and output ports. Actors communicate if they are connected.

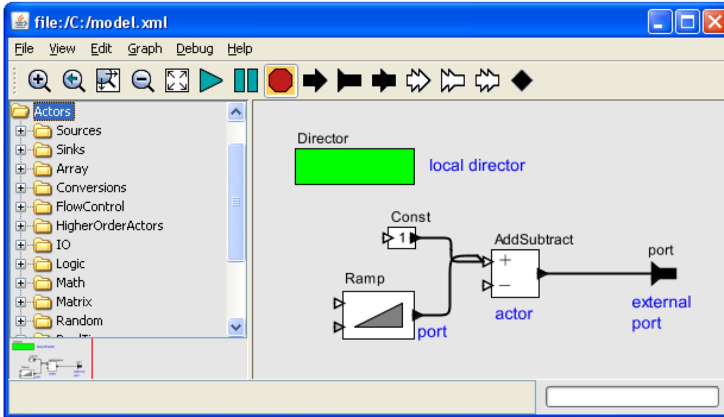


Fig. 5.13. Example of a Ptolemy model

A model can have external input and output ports and can be embedded as a composite actor in another model where it appears as an actor with local input and output ports.

Simulating a model means executing actors as defined by the top level model director. During the simulation, an actor experiences a number of iterations, where an iteration generally consists of three successive actions: *prefire*, *fire* and *postfire*. Each action is represented by a method in the actor interface. The main functionality of the actor is encoded in the *fire* method. In *prefire*, possible preconditions for execution are tested. Thus, the actor can indicate to the enclosing director that it does not wish to be fired. By convention, if the *prefire* method returns false, then the director will not call the *fire* method in the current iteration. An actor reads inputs and produces outputs in the *fire* method, which may be called multiple times in the same iteration. In *postfire*, the actor updates its persistent state and indicates to the director if the execution is complete. If *postfire* returns false, the director should perform no further iteration on the actor in the current simulation.

The TDL Domain in Ptolemy II

The implementation of TDL's modal structure is based on the modal model variant of the Finite State Machine (FSM) domain in Ptolemy, and the implementation of the LET-based semantics employs essentially a DE approach. Like modal models, TDL modules consist of modes with different behaviors, where only one mode can be active at a time. Mode switches in modal models have the same semantics as mode switches in TDL and TDL activities are conceptually regarded as discrete events that are processed in increasing time stamp order.

The TDL domain consists mainly of three specialized actors: *TDLModule*, *TDLMode*, and *TDLTask*. The *TDLModule* actor (with the associated *TDL-ModuleDirector*) restricts the basic modal model according to the TDL modal semantics. In a modal model actor, mode transitions are checked every time the

actor is fired. TDL restricts the times when mode switches can be made (mode switches are not allowed during a task's LET). A similar restriction applies to port update operations. A TDL module can have guards also on task invocations and port updates, not only on mode transitions, as in the modal model. TDL requires a deterministic choice of simultaneously enabled transitions, which is not provided by the FSM domain. In this respect, we employ a convention similar to the one used in Stateflow(R), where the outgoing transitions of the active mode are tested based on the graphical layout, in clockwise order starting from the upper left corner of the graphical representation of the mode.

We consider applications with time-triggered and event-triggered components modeled in the DE domain. The functional application model is mapped to a platform model by assigning to each task a priority and a worst case execution time. The mapped model is then simulated with the help of a specialized domain controller, which is a modified DE controller. This uses an event queue and works by processing the events in the queue in increasing timestamp order. While TDL operations can be statically scheduled (they are periodic and have the highest priority), the actual moments of task executions are represented by dynamic events, as are the executions of the other event-triggered tasks.

The main difference between the implementation of the TDL-Simulink integration and the TDL domain in Ptolemy II refers to the fact that, while the former employs a Simulink implementation of the TDL:E-Machine, the latter uses no virtual machine. TDL specifications are expressed as properties of Ptolemy actors and the TDL domain uses these properties to generate an appropriate schedule of events. TDL actions are naturally represented by discrete events, and we leverage the event handling mechanism of the DE domain to achieve a correct execution of the model. In particular, this implies that any future change in the TDL semantics can be much more easily handled in the TDL Ptolemy domain, where one has to change only the event scheduling part. In contrast, in the Simulink case, changes may need to be done in the TDL compiler, in the e-code instruction set and in the TDL:E-Machine implementation. An additional advantage of the TDL-Ptolemy integration is related to the fact that mapping of a functional model to a platform model can be done much easier in Ptolemy II than in Simulink. This is due to the versatility of Ptolemy II and the availability of different models of computation. Thus, a mapped model can be obtained from the functional model by a combination of two actions: (1) Adding properties to functional actors, and (2) Choosing or defining a suitable model of computation. This enables one to simulate the (runtime) TDL operations at the platform level.

Example

In the sequel, we show how the TDL domain in Ptolemy II can be employed to demonstrate the benefits of using TDL. In the following example, a simple application with timing constraints is developed from a high-level discrete event model to an implementation on a given platform. We outline a case where, if timing constraints are expressed without TDL, the behavior of the final implementation is different than the behavior of the original model. By using TDL,

the behavior of the original model remains unchanged and it is preserved in the final implementation.

Figure 5.14 shows an application modeled in Ptolemy II as a discrete event system, with one time-triggered and two event-triggered tasks. The actor `TTTask` is triggered by the clock signal with a period of 8 time units and it produces output with a delay of 4 units of time after being triggered. Consider a simulation of the model with two events from `sensor1` at times 5 and 9, and one event from `sensor2` at time 7. The execution of the task actors is shown in Figure 5.15. Notice that the time-triggered actor `TTTask` reads (at time 8) the value computed by the event-triggered actor `ETTask1` at time 5.

This application is to be deployed on a computational platform with a fixed priority preemptive scheduling policy. Thus, code is generated from the task actors and priorities are assigned to the computational tasks. Assume that the priority of `ETTask1` is higher than the priorities of both `ETTask2` and `TTTask`, which are equal. Consider an execution of the application on the platform with the same input as in the simulation of the functional model, where the execution

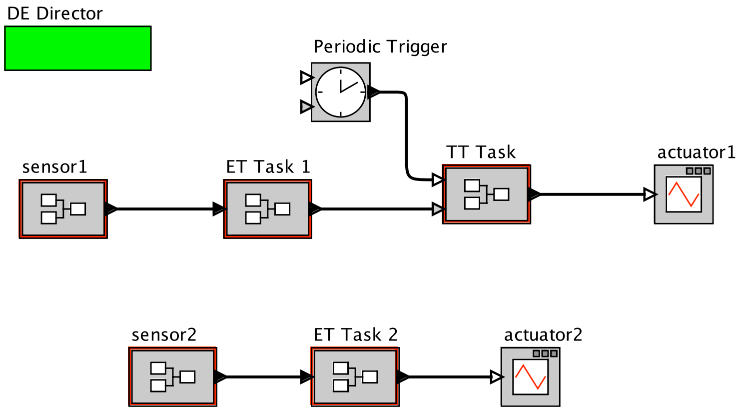


Fig. 5.14. A discrete event model

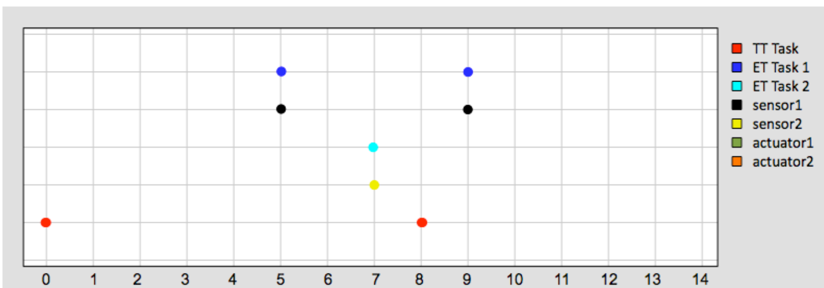


Fig. 5.15. An execution of the above model

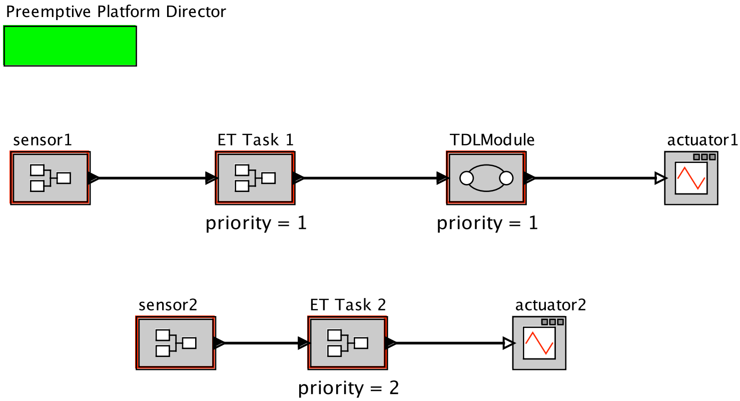


Fig. 5.16. A TDL model

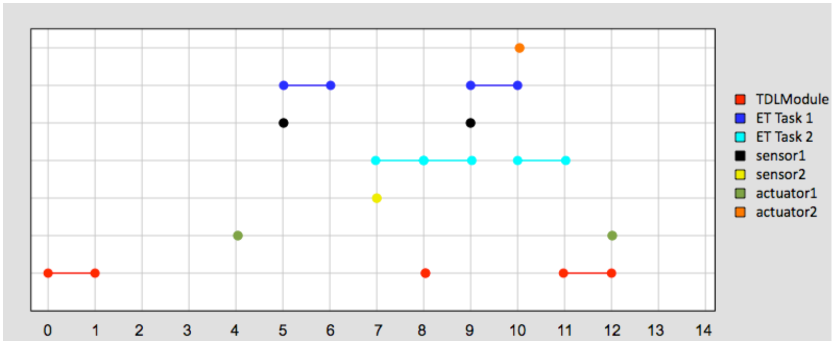


Fig. 5.17. Simulation of the TDL model

times of `ETTask1`, `ETTask2` and `TTTask` are respectively 1ms, 3ms and 1ms. In this case, `TTTask` cannot be executed at time 8, when `ETTask2` is still in execution. Also, `ETTask1` preempts `ETTask2` at time 9, further delaying the starting of execution of `TTTask` until time 11. Notice that the order of execution of `TTTask` and `ETTask1` is changed in the implementation versus the original model. In particular, this implies that `TTTask` may have a different input value, hence the output behavior of the system may be changed.

Consider now a TDL model of the above application where the delay in the original time-triggered task is replaced by a logical execution time equal to 4. Let us map the TDL model into a platform model (see Figure 5.16). A specialized director (a variant of the DE director) is employed to simulate the mapped model. Figure 5.17 shows the execution of tasks under the input described above. Notice that the TDL module actor samples its input at time 8, then uses this value as input for the TDL task corresponding to the original `TTTask`. Thus, the

mapped TDL model has the same output behavior as the TDL functional model (which has the same behavior as the functional DE model).

5.4 Related Work

TDL belongs to the family of time-triggered modeling languages and tools with roots in Giotto, such as xGiotto [14], HTL [3], and FTOS [15]. TDL stands out in this landscape due to its focus on control applications. It is, for example, the only language with a fast step feature that matches the "update outputs" part of a controller, which accommodates the need for short response times. In contrast, the Giotto software model maximizes the delay between sensor read and actuator update (placing them one LET apart), while minimizing the delay between actuator update and the next sensor read for the same task (placing them in sequence at the same time step). One important aspect in which TDL differs from Giotto is the treatment of mode switches. While Giotto allows mode switches during the LET of a task, this is not supported in TDL because it would imply a significantly more complex communication schedule generator algorithm for distributed TDL modules. Also, Giotto ensures determinism of mode switching by restricting the number of mode switch conditions that may evaluate to true to at most one. In TDL, mode switch guards are evaluated in the textual order from top to bottom and a mode switch is performed for the first condition that evaluates to true.

Among the above mentioned languages, only HTL allows flexible placement of the LET in as task's invocation period. There is also a Simulink integration of HTL [16]. In contrast to our approach, the simulation results do not match the HTL description exactly. For breaking algebraic loops, additional delay blocks are introduced which influence the observable timing. Additionally, the HTL integration in Simulink trades off accuracy for performance since it requires the sample rate of some blocks to be at least one decimal order of magnitude higher than actually required by the HTL description.

The TDL domain in Ptolemy II is related to the experimental Giotto domain in Ptolemy II[17]. The main differences between the TDL domain and the Giotto domain are as follows:

– In addition to functional models, TDL operations can be simulated also in mapped models, which contain platform specific attributes.
– The TDL domain leverages the existing DE domain while the Giotto domain is designed based on basic Ptolemy II software components.
– The implementation of the TDL domain reflects the distinction between the fundamental concepts (LET, modes) and the way these concepts are used (the operational semantics). The implementation is two-layered: the basic layer deals with scheduling LET-based tasks grouped in modes, and the operational layer corresponds to a specific time-triggered programming model. The latter extends the basic layer by specifying additional operations, as well as the order of data transfer and mode-change operations according to the programming model semantics. In principle, this enables achieving

domain controllers for other time-triggered programming models (including Giotto) by extending the basic layer.

Achieving determinism of time-triggered software is the main goal of several commercially available tools such as TTTech [18], DaVinci [19] and dSPACE [20]. A detailed comparison between TDL and each of these tools is provided in [21].

5.5 Conclusions

Timing requirements of real-time applications can be effectively achieved by using the LET approach through an established set of methodologies and tools such as the ones provided by TDL. The ability to deal with control applications was further increased by adding two extensions: (1) The *fast step*, which allows actuator update immediately upon sensor reading, and (2) The *slot selection* for flexible LET placement, which allows specification of offsets between tasks in the system.

Simulation is a powerful tool, widely used in the embedded systems industry to validate properties of complex systems. This chapter presented TDL-specific extensions of two major simulation platforms: Simulink and Ptolemy II. The TDL-Simulink integration significantly increased the accessibility of the LET-based programming model to control application developers and system integrators. The TDL tools available in Simulink make it possible to easily go through the development stages of modeling, simulation/testing, code generation and deployment to (possibly distributed) execution platforms.

The TDL domain in Ptolemy II enables, among other things, visualization of an important LET benefit: preservation of time and value determinism from high level models to lower level, platform specific, implementations. The main motivation behind its development was the observation that the influence of using LET on a system's behavior can be captured by simulation of a mapped model, even when only few platform-specific properties are considered. This could not be easily achieved by using Simulink. Ptolemy II enables experimentation and investigation of heterogeneous models of computations, where LET-based systems using Giotto and TDL can be mixed with more general, event-based systems. This can help in exploring the concept of "open" TDL models, where event-based computations can be accommodated while still guaranteeing schedulability of the system.

Acknowledgements

We thank the anonymous reviewers whose comments have been helpful in improving the presentation of this chapter.

References

[1] Stankovic, J.A.: Misconceptions about real-time computing: a serious problem for next-generation systems. Computer 21(10) (1988)

[2] 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) (February 2003)

[3] Ghosal, A., Henzinger, T.A., Iercan, D., Kirsch, C.M., Sangiovanni-Vincentelli, A.: A hierarchical coordination language for interacting real-time tasks. In: Proceedings of the 6th ACM International Conference on Embedded software, Seoul, Korea. ACM, New York (October 2006)

[4] Templ, J.: TDL - Timing Definition Language 1.5 Specification. Technical report, preeTEC GmbH (2008), http://www.preetec.com

[5] Object Management Group: Model driven architecture. Technical report (2008), http://www.gigascale.org/pubs/141.html, http://www.gigascale.org/pubs/141.html

[6] Sangiovanni-Vincentelli A.: Defining platform-based design. EEDesign of EE-Times (February 2002)

[7] Henzinger, T.A., Kirsch, C.M.: The embedded machine: predictable, portable real-time code. In: PLDI 2002: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, pp. 315–326. ACM, New York (2002)

[8] preeTEC: The TDL tool chain. Technical report, GmbH (2008), http://www.preetec.com

[9] The MathWorks (2008), http://www.mathworks.com

[10] Stieglbauer, G., Pree, W.: Visual and Interactive Development of Hard Real Time Code. In: Automotive Software Workshop San Diego, ASWSD (January 2004)

[11] Henzinger, T.A., Horowitz, B., Kirsch, C.M.: Giotto: A time-triggered language for embedded programming. Proceedings of the IEEE 91, 84–99 (2003)

[12] 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)

[13] Brooks C., Lee E.A., Liu X., Neuendorffer S., Zhao Y., Zheng H. (eds.): Heterogeneous concurrent modeling and design in java (volume 1: Introduction to ptolemy ii). EECS Department, University of California, Berkeley UCB/EECS-2007-7 (January 2007)

[14] Ghosal, A., Henzinger, T.A., Kirsch, C.M., Sanvido, M.A.: Event-driven programming with logical execution times. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 357–371. Springer, Heidelberg (2004)

[15] Buckl, C., Regensburger, M., Knoll, A., Schrott, G.: Models for automatic generation of safety-critical real-time systems. In: Proceedings of the Second International Conference on Availability, Reliability and Security (ARES), pp. 580–587 (2007)

[16] Iercan, D., Circiu, E.: 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)

[17] Brooks C., Lee E.A., Liu X., Neuendorffer S., Zhao Y., Zheng H. (eds.): Heterogeneous concurrent modeling and design in java (volume 3: Ptolemy ii domains). EECS Department, University of California, Berkeley UCB/EECS-2007-9 (January 2007)
[18] TTTech Computertechnik AG: TTP tools (2009), http://www.tttech.com/products/ttp/design-development-software
[19] Vector Informatik GmbH: DaVinci Network Designer 2.0 (2009), http://www.vector.com/vi_davinci_networkdesigner_en.html
[20] dSPACE GmbH: Real-time interface (RTI and RTI-MP) implementation guide (2009), http://www.dspace.de
[21] Farcas C., Holzmann M., Pletzer H.: The TDL advantage. Technical report, Stieglbauer G. (2004), http://cs.uni-salzburg.at/pubs/reports/T002.pdf