Modeling and simulation of timing behavior with the Timing Definition Language (TDL)

Josef Templ¹, Andreas Naderlinger¹,², Patricia Derler³, Peter Hintenaus¹, Wolfgang Pree¹,², Stefan Resmerita¹,²

¹C. Doppler Laboratory Embedded Software Systems, Univ. Salzburg, Austria
²Chrona.com, Salzburg, Austria and Torrance, California
³Electrical Engineering and Computer Sciences (EECS) Department, University of California, Berkeley

Abstract

Most of the existing modeling methods and tools for embedded application development use levels of abstraction where execution and communication times of computational tasks are not adequately captured. Thus, properties such as time and value determinism, which are crucial for safety-critical real-time systems, cannot be guaranteed when refining the model to move it closer to a target platform. In many cases, the link between the abstract model and its implementation is broken during the refinement process. An additional problem is that the resulting software components are often not portable and composable. The Logical Execution Time (LET) abstraction has been proposed to deal with these issues and to offer a correct-by-construction development that formally guarantees time and value determinism and that delivers portable embedded software components. LET allows the explicit, platform-independent specification of execution times.

This chapter deals with the modeling and simulation of safety-critical embedded applications where timing requirements are specified in the Timing Definition Language (TDL) which supports the LET abstraction. TDL provides a programming model for both the time-triggered paradigm and the event-triggered paradigm. TDL components can be automatically deployed to large-scale distributed systems. We present a selection of TDL constructs and sketch the
seamless integration of TDL with two simulation environments, the MATLAB®/Simulink® [31] products from MathWorks® and the open-source Ptolemy framework from the University of California, Berkeley.
**Introduction**

Traditional development of software for embedded systems is highly platform specific. Exploiting a specific platform enables reducing cost of hardware to a minimum whereas high development costs of software are considered acceptable in case of large quantities of devices being sold. Nowadays, with ever more powerful processors in the low cost range, we observe even more of a shift of functionality from hardware to software and a general tendency towards more ambitious requirements. Modern cars or airplanes, for example, contain dozens of so-called electronic control units interconnected by multiple buses and are driven by several million lines of code. In order to cope with the increased complexity of the embedded software, a platform independent “high-level” programming style becomes mandatory, as testing alone can never catch all errors. In particular, in case of safety-critical real-time software, this applies not only to functional aspects but to the temporal behavior of the software as well. Dealing with time, however, is not covered at all by any of the existing high-level imperative languages. Simulation environments that offer delay blocks allow at best the approximation of the simulated behavior to the behavior on the execution platform.

One reason is that 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 [36]. 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 [12]. One promising direction in this respect is the Logical Execution Time (LET) [3].

This chapter presents the explicit specification of the timing behavior using the LET-based
Timing Definition Language (TDL). TDL is under active, commercially supported development [14]. As simulation is widely used in industry for testing and validation of complex systems (e.g., [29]), it is important to be able to simulate TDL-based systems. Thus, we describe how TDL has been integrated with two distinctly different simulation environments, that is, MATLAB®/Simulink® [4] from MathWorks® and Ptolemy [5], an open source environment developed at the University of California at Berkeley. Where MATLAB/Simulink is treated as yet another execution platform, Ptolemy is more closely aligned with TDL principles and so offers simulation capabilities that allow a more straightforward integration of TDL. We chose these two simulation environments to demonstrate two quite different approaches for simulating TDL-based systems.

The Timing Definition Language (TDL)

The Timing Definition Language (TDL) [1] at its core follows the time-triggered programming model [3, 26]. In a time-triggered system, all activities are triggered only by the ticks of a single global clock. In order to increase the range of applicability, TDL also supports a limited form of event-triggered programming, which allows, for example, responding to hardware interrupts.

TDL Properties

TDL programs that only rely on the time-triggered features exhibit the following properties by construction:

*Time and value determinism*

Value determinism means that a program provides the same outputs if it is provided with the same inputs. Time determinism means that a program provides the outputs at the same times if it is provided with the inputs at the same times, where all times are relative to the program start.
TDL aims for both time and value determinism. Thus, a TDL program provides the same outputs at the same times if it is provided with the same inputs at the same times. In other words, the chronologically ordered sequence of outputs (time plus values) of a TDL program, which is also referred to as the observable behavior of a program, is deterministic and platform independent.

Portability

TDL programs represent a platform independent description of the timing behavior of an application. Everything that is platform specific, for example accessing sensors or actuators, is defined outside the TDL program. TDL programs behave exactly the same independent of the underlying CPU, network bandwidth, or operating system. Even when simulating a TDL application, for example under MATLAB/Simulink, the application exhibits the same behavior.

Transparent distribution

Since TDL abstracts from the execution platform, a TDL application shows the same observable behavior in case of a distributed system as on a single-node system. Thus, the fact that a distributed system is used as an execution platform is transparent. It is the task of the TDL compiler to generate a suitable network communication schedule for maintaining the observable behavior of the application [7].

Time safety

The TDL compiler provides a time safety check, which guarantees that a program behaves as expected for a particular target platform given that the worst case execution times for the tasks to be executed are known for that platform. In case of a distributed platform, the compiler also guarantees that the network communication preserves the expected observable behavior of the application.
Compositionality

A TDL program consists of a set of so-called modules. All modules are executed in parallel and the data flow between modules is handled by the TDL runtime system. Adding another module to the application does not change the observable behavior of the previously existing modules.

TDL Language Constructs

In the sequel we introduce the individual TDL language constructs informally. For more details including a formal grammar please refer to the TDL Language Specification [1].

Modules

At the outermost level, a TDL application consists of a set of modules. Two modules can either be independent, that is, they share no data, or cooperating. Cooperating modules exchange data via ports. Statically, a module provides a namespace. Dynamically, modules are executed in parallel—possibly on different nodes in a distributed system. All modules share a common clock which, in the case of a distributed execution platform, has to be distributed to the individual nodes of the platform. A module may encapsulate a finite state machine, where the states are denoted modes. In a mode the temporal aspects of all activities are defined.

Ports

Data flow within a single TDL module, between multiple modules, and between a TDL module and the physical environment is exclusively based on ports. A port is a typed variable that is accessed (read or written) at specific time instances only. Sensor and actuator ports (sensors and actuators for short) are the only means for a TDL module to communicate with the environment. A sensor declaration defines a typed read-only variable to represent a particular value in the physical environment and provides input to the TDL application. An actuator declaration is an initialized and typed write-only variable that influences a particular value in the physical
environment and provides output from the TDL application to the environment. The access to the hardware is performed by user provided setter and getter functions that are external to TDL.

**Tasks**

A task is the computational unit in TDL. It defines a namespace for input, output, and state ports. Each task is associated with a task function, that is, a stateless piece of code without any synchronization points. A single invocation of a task at runtime creates a *task activation*. A task activation lasts for a strictly positive amount of time that starts at the release time (the time when the task activation is released) and ends at the termination time (the time when the task activation is terminated). The time between these two instants is called the *Logical Execution Time (LET)* of the task activation (see Figure 1). At the release time, the input ports of the corresponding task are updated with the values read from the output ports of other tasks and sensors that have been passed as parameters by the task invocation. The actual execution of the task function can be scheduled at will, as long as its execution starts after the release time and finishes before the termination time. The task activation locally buffers the output of the task function. At termination time, the output ports of the task are updated with the values stored in the local buffers of the task activation. State ports hold data that must persist among multiple activations of a task.

![Figure 1 Logical Execution Time (LET)](image)

**Modes**

Modules that encapsulate a state machine have a dedicated start mode each and can switch
between modes independently of others. A mode \( m \) specifies a mode period \( P_m \) (in µs) and a set of activities. As long as a module remains in mode \( m \) the activities associated with \( m \) are repeated with period \( P_m \). A mode activity either is a task invocation, an actuator update, or a mode switch. Mode activities may be guarded. A \textit{guard} is a function that returns either true or false. A guarded mode activity is executed only if its guard evaluates to true.

When defining a mode activity within a mode, a frequency \( f \) for this activity is specified. This frequency divides the mode period \( P_m \) into \textit{slots} of duration \( P_m/f \) each. These slots define the times during the mode period at which the guard of the activity is evaluated and the activity is executed. Actuator updates and mode switches are executed at the end of a slot. Task invocations result in the release of a task activation at the beginning and the termination of the particular activation at the end. Thus, the LET of an activation spans the entire slot, and, by default, the task invocation is executed in each slot.

For control systems, the fixed relationship between the rate of task invocations and the LET of each task activation poses problems as the delay between reading the sensors and updating the actuators consumes phase reserve in the control loop (e.g., [2]). Besides increasing the frequency of a task invocation and thus the sampling rate, which may be prohibitive in terms of CPU load, TDL offers two mechanisms for dealing with this situation, slot selection and task splitting.

\textit{Slot selection} is the explicit selection of the slots in which a mode activity should be executed. For task invocations, slot selection maintains the basic pattern of freezing the input ports of a task at the release of each of the tasks activations, and update of the output ports (and actuators) at termination. It allows a separation between specification of the LET of a task activation and the repetition rate of the associated task. Slots that are not selected go unused.

\textit{Task Splitting} means to split the single task function into two functions, one called fast step and
one called slow step. TDL assumes that the fast step does not consume any time. At release time, first the input ports of the task are updated and then the fast step is executed. As a modification of the basic actuator update pattern, there is the possibility to update an actuator with a value calculated in the fast step immediately after it has finished execution. The slow step is executed afterwards, during the LET of the activation. At the termination time, the output ports of the task are updated and further actuator updates may be performed. In a control system application, for example, the actual controller may be moved into the fast step. Thus, the delay between reading the output of the plant and updating the actuator which delivers the input to the plant is minimized. A state estimator (e.g., [2]), representing a higher computational load, will then be moved into the slow step.

Asynchronous (= event-triggered) activities

In addition to time-triggered (alias synchronous) activities, it is often necessary to execute event-triggered (alias asynchronous) activities as well [8]. TDL supports asynchronous task invocations and actuator updates. Such an asynchronous activity is either triggered by an update of an output port, by the occurrence of a hardware interrupt, or by the tick of a timer that may potentially introduce its own time base.

By integrating asynchronous activities into TDL, the TDL runtime system is able to provide the synchronization of the data flow between synchronous and asynchronous activities. It has been shown in [9] that a lock-free synchronization approach with a negligible impact on the timing of the time-triggered activities is possible with the semantics outlined below.

Events may be associated with a priority and are registered in a priority queue when they arrive. Processing the events is delayed and supposed to be performed sequentially by a single background thread that runs whenever there are no time-triggered activities to perform. Input
ports are read as part of the asynchronous execution, not at the time of registering an event. Output ports are updated immediately after an asynchronous task invocation has finished. If an activity is triggered again before it has started to execute it will not be executed a second time but remains registered once. In case of a distributed system, the communication of asynchronous output values to remote nodes is supposed to rely on asynchronous network operations. Since any network operation introduces a delay, the transparent distribution property (see above) does not hold in case of asynchronous activities [10].

**Example TDL Modules**

The following example of two TDL modules exemplifies the textual syntax of TDL (see also Figure 2). As an alternative to the textual representation, the TDL toolchain also provides a syntax-driven editor that supports a visual and interactive modeling of TDL modules (see Figure 6).

```
module Sender {
    sensor int s1 uses getS1;
    actuator int a1 uses setA1;

    public task inc {
        input int i;
        output int o := 10;
        uses incImpl(o);
    }

    start mode main [period=5ms] {
        task
            [freq=1] inc(s1);//LET = 5ms/1 = 5ms
        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
        ...
    }
```

Module *Sender* contains a sensor variable *s1* and an actuator variable *a1*. The value of *s1* is updated by executing the (platform-specific) function *getS1* and the value of *a1* is sent to the physical actuator by using the platform-specific function *setA1*. The declaration of task *inc* contains an input port *i* and an output port *o* with an initial value of 10. This task is invoked in the mode *main*, where it reads input from the sensor *s1*. In the same mode, actuator *a1* is updated with the value of the task's output port. The timing behavior of the mode activities is specified by means of individual frequencies within their common mode period. For example, with a frequency of 1, the activation of task *inc* is defined to have a LET of 5 ms. The second module called *Receiver* imports the *Sender* module in order to connect the output of the task *inc* with the input of the task *clientTask*.

*The TDL toolchain*

TDL introduces appropriate abstractions to separate timing from functionality and platform-independent from platform-specific aspects. In order to obtain executable software, the textual TDL description must be compiled and combined with external functions that implement the required functionality. Figure 3 outlines the TDL toolchain. It shows as a central component the TDL compiler that compiles a textual TDL program to platform-independent embedded code.
(so-called E-code). The TDL compiler also offers a plug-in-architecture for generating target platform specific output. For example, on an automotive platform with OSEK as operating system the platform-specific output could include so-called OIL files [30]. The E-code together with platform-specific output and the functionality code corresponding to task function implementations is used by the TDL run-time system, the so-called E-machine [11], to execute TDL applications.

![Figure 3 The TDL toolchain](image)

**E-code**

The TDL compiler generates E-code for each mode of a module. The E-code covers a single mode period and is repeated by means of a jump instruction to the beginning of the mode. For every logical time instant at which E-code must be executed, one E-code block is generated. An E-code block is a list of E-code instructions. It specifies for one logical time instant the actions that must be taken by the E-machine in order to comply with the timing specifications and LET semantics. The following generic sequence of actions comprises one E-code block for a logical time instant $t$:

1. Update the output ports of all tasks that are defined to be terminated at $t$.
2. Update all actuators that are defined to be updated at $t$.
3. Switch mode if a mode switch is defined at $t$.
4. Update the input ports of all tasks that are defined to be released at $t$. 

5. Release all task activations that are defined to be released at $t$.

6. Sleep until the next logical time instant that must execute E-code.

Sensors are read whenever their value is required. However, at one particular logical time a sensor is read at most once.

In the following, we illustrate these actions for the module *Sender* from the previously described example. At time 0, actions in the start mode are processed. Output ports are initialized and connected actuators are updated. Then, the sensor $s1$ is read and an activation for task *inc* is released. There are no further actions to be processed at time 0. At time 5, which is the end of the LET of the first activation of task *inc*, the task’s output port is updated. Following this, the actuator $a1$ is updated. Next, the mode switch condition in the guard function *exitMain* is evaluated. This causes sensor $s1$ to be read and the value is provided as input to *exitMain*. If the guard evaluates to true, a mode switch to the empty mode *freeze* is performed and no further actions are processed. Otherwise the module remains in the mode and the next activation for task *inc* is released. Figure 4 shows the periodic execution pattern of task *inc* in mode *main* of module *Sender* and of the task *clientTask* of module *Receiver*.

![Figure 4 The periodic execution of the tasks *inc* and *clientTask*](image)

In the following we describe and compare the integration of TDL with the two simulation environments MATLAB/Simulink and Ptolemy. As these two environments are quite different, this requires two entirely different integration strategies. This applies to both modeling and simulation. While Ptolemy is open source, targeted to support a variety of different models of
computation, and highly adaptable even for fundamental elements, MATLAB/Simulink is proprietary and more restrictive in its adaptability.

Seamless TDL integration with MATLAB®/Simulink®—a developer's perspective

Simulink builds on MATLAB (both products by MathWorks) [4] and has become the de-facto standard for the modeling and simulation of real-time systems in various domains such as automotive, avionics, and aerospace. First attempts to integrate TDL with MATLAB/Simulink started as early as the initial development of TDL in 2003 [15]. In Simulink, systems are modeled in a visual and interactive environment using (mostly time-based) block diagrams. Code generators may then automatically translate the block diagrams into software, for example, into C code.

According to the results described in [16], manually implementing LET semantics in Simulink is strongly discouraged. Even simple models of single-mode systems are cluttered with additional blocks to ensure that the timing behavior in the simulation conforms to LET semantics. It turned out, that it is practically infeasible to model LET based applications with multi-modal behavior by hand, even when using the Simulink extension Stateflow® [32]. In the following we describe an approach that is based on an explicit timing specification with TDL. For the simulation, the TDL specification is automatically translated into a Simulink model with an E-machine implementation at its core. In this sense MATLAB/Simulink represents yet another execution platform for TDL modules.

We will start with a developer’s perspective of the TDL integration with MATLAB/Simulink that covers the modeling, the simulation, and finally the platform mapping and the code generation.

Extension of the TDL toolchain for MATLAB®/Simulink®
Figure 5 outlines the TDL toolchain when used together with MathWorks tools. Real-Time Workshop® Embedded Coder™ [33] (RTW-EC in Figure 5) can be used to generate the C source code for the TDL task implementations. The so-called TDL:VisualCreator tool allows the visual and interactive modeling of TDL applications. For mapping TDL modules to specific platforms, that is, for organizing the various build steps we provide the so-called TDL:VisualDistributor tool. It allows a developer to:

- Define a hardware topology. This can be a single node or a cluster consisting of potentially heterogeneous nodes that are connected, for example, via a time-triggered Ethernet.
- Assign the individual TDL modules to their target nodes.

A built-in code and schedule generation framework generates platform-specific code, a communication schedule in case of a distributed platform, make files, and any other output required for a particular platform [14].

![Figure 5 The TDL toolchain in MATLAB®/Simulink®](image)

**Modeling**

The modeling typically comprises two principal components, the controller and the plant. The plant is modeled as usual in MATLAB/Simulink. The controller is modeled with one or multiple TDL module block(s) (see Figure 6) available in the TDL library.
Each TDL module block represents one TDL module. Instead of using the textual representation of TDL, the module is edited using the TDL:VisualCreator tool that opens when double-clicking a TDL module block. The TDL module is 2-way synchronized with Simulink. This means, that a change in the TDL:VisualCreator, such as adding a sensor, is immediately reflected in the Simulink representation of the particular TDL module and vice versa. A sensor of the TDL module is represented as `Inport`, and an actuator is represented as `Outport` of the TDL module block. A task is represented as a `(Function-Call) Subsystem` that resides within the TDL module block. Within this subsystem, the functionality of the task may be modeled with appropriate
library blocks (excluding those that comprise continuous-time behavior) with inherited sample
time. Figure 6 shows the previously described TDL example within a MATLAB/Simulink model
and the two tools TDL:VisualCreator and TDL:VisualDistributor. The TDL:VisualCreator tool
lists the individual elements such as the sensor $s_1$ and the task $inc$ of module Sender in a tree
representation. The activities of mode $main$, for example, are shown in the right-hand half of the
frame. Their timing is specified via properties in the table below the tree representation.

**Simulation**

The overall system can be simulated once the application developer has finished the modeling
phase, that is, timing behavior has been specified using the TDL:VisualCreator while controller
functionality and plant behavior have been modeled with Simulink blocks. From the developer's
point of view, there is no observable difference to starting a simulation if there were no TDL
blocks present. This is achieved by an internal model translation (as sketched in the toolchain) to
ensure that the simulation corresponds with the TDL specification. In fact, during the simulation
the compiled TDL program is executed within an E-machine encapsulated in a Simulink S-
function. Details of how this is accomplished are described below.

As the TDL code and schedule generators ensure that the timing behavior of TDL modules is
equivalent when executed on a single node and when distributed among multiple nodes, the
simulation can assume execution is on a single node and it is not necessary to account for any
communication behavior.

**Platform Mapping and Code Generation**

In order to generate code for the application, the target platform must be specified. This is
performed with the TDL:VisualDistributor tool that is integrated with Simulink as a *TDL
Distribution block* (see the yellow box in the top-right corner of the Simulink model in Figure 6).
The platform involves the specification of the (potentially heterogeneous) node platforms, their interconnections within the cluster, and the communication protocol. For this purpose, the developer may choose from a set of available node (such as a dSpace MicroAutoBox for prototyping) and cluster (such as FlexRay) plug-ins. After every TDL module of the Simulink model has been assigned to a node, the developer may start the generation of platform specific code, a communication schedule in case of distributed platforms, make files, and any other required output. If desired, the TDL:VisualDistributor tool can also trigger the Real-Time-Workshop Embedded Coder (RTW-EC) to generate C code for all the tasks of the TDL modules.

**Seamless TDL integration with MATLAB®/Simulink®—implementation perspective**

On an embedded hardware platform, the E-machine represents the core piece for a LET based execution. E-machine implementations exist for several different platforms [13]. For the TDL integration with MATLAB/Simulink, we implemented an E-machine that is based on a Simulink S-Function [16]. An S-Function is a Simulink block that references a user defined functionality implemented in a programming language such as C and compiled by the MATLAB Executable (MEX) compiler. In this way, the built-in Simulink blockset can be extended. S-Functions are composed of *callback methods* that the Simulink engine executes at particular points during the simulation.

As the S-Function implements the E-code interpreter, it must be invoked whenever the simulation time matches the logical time of a TDL activity, as defined in the E-code. According to the E-code instructions, the S-Function triggers the execution of so-called *Function-Call Subsystems*. Each task and each guard is represented as a Function-Call Subsystem that is provided by the developer. Additional Function-Call Subsystems are generated automatically as part of the model translation when the simulation is started. They implement the port assignment operations, for
example, to update an actuator port with the value of a task output port.

Figure 7 exemplifies this E-machine approach for a simplified application. The placement of the individual blocks conforms to the data flow, which is basically from left to right along the arrows from a source to a sink. The source value is read by a sensor which provides the value to a guard and a task. The actuator block uses the output port of a task to write to a sink. The E-machine triggers the individual blocks according to the E-code resulting in the indicated order (1-6). The input port of such a generated subsystem is directly connected to the output port, which corresponds to an assignment in the imperative programming paradigm as soon as the system is triggered. This ensures the correct LET behavior of a task activation, for example when triggering its release and termination at the correct time instants. Both fixed and variable sample time approaches for the E-machine are possible [17]. The suggested value for a fixed sample time is the GCD (greatest common divisor) of all activity periods.

![Figure 7 The basic principle of an E-machine as a Simulink® S-Function and Function-Call Subsystems](image)

Resolving Data Dependencies

The S-Function implementation of the E-machine for a simulation environment is analogous to E-machine implementations for hardware platforms. Compared to other simulation approaches, such a Simulink E-machine results in an efficient simulation model [16]. However, because of data dependency problems that can occur in simulation environments, the practical applicability of this basic mechanism turned out to be limited. We identified the following application...
scenarios that in general cannot be handled by this integration concept:

- Cyclic import relationships between LET-based controllers (TDL modules)
- Control loops involving plants without delay
- Control loops with mixed LET-based and conventionally modeled controllers

These cases are discussed in detail in [17]. They are all related to cyclic data flow dependencies and the ability of the simulation environment to find a valid strategy for executing each individual block. Like many other simulation environments, Simulink does not support cycles without a delay except for special cases [35]. Delays are introduced by explicit delay blocks, or by other blocks whose output is not directly controlled by the input (though possibly dependent on the block state). Those blocks are said to have indirect (or non-direct) feedthrough. When Function-Call Subsystems are involved, Simulink reports a data dependency violation, which is similar to an algebraic loop error [28], when attempting to simulate a model with a direct data dependency cycle. From the control engineer's point of view, this appears to be counterintuitive, since the LET of a task is always greater than zero and thus should introduce the required delay. The problem is that the simulation environment is not aware of this LET characteristic.

In [18], we propose an E-machine implementation that consists of two interacting S-Functions. Without violating the TDL specification, i.e. without changing the timing behavior of the simulation, this approach introduces additional delay blocks in order to resolve the cyclic dependencies. This 2-step E-machine architecture is capable of simulating the scenarios described above and also supports TDL applications with mixed time- and event-triggered (asynchronous) activities [17]. In case of simulating event-triggered activities, the simulation of events and the corresponding reaction cannot be guaranteed to match the behavior on a specific target platform. This is because asynchronously activated tasks do not have a LET, and also
because the simulation is not aware of any scheduling strategy, distribution topology, or CPU speed of the target platform.

**TDL integration with Ptolemy II**

Ptolemy II is the software infrastructure of the Ptolemy project [5], which studies modeling, simulation, and design of concurrent, real-time, embedded systems. It is an open source tool written in Java that 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 that can communicate with other actors through ports. 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.

**The TDL domain**

TDL is implemented as an experimental domain in Ptolemy. 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 port updates. To ensure the LET of a task activation, input ports 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 one of all simultaneously enabled transitions, which is not provided by the FSM domain. We resolve this by employing a convention similar to the one supported by Stateflow [4], 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. 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 is amenable to DE actors. A TDL task is implemented as 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.

![Figure 8 The TDL module Sender in Ptolemy II](image)
The module \textit{Sender} from the example above modeled in Ptolemy is shown in Figure 8. The model shown in the top-left box of the figure contains a TDL Module and two actors to provide sensor values and display actuator values. The TDL Module contains two modes \textit{main} and \textit{freeze} (see Figure 8, the bottom-left box). Both modes have their period indicated by an associated 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 the task invocation, which determines the LET, is defined as a parameter.

Because of the iterative execution strategy of Ptolemy, problems with irresolvable cyclic dependencies as described for the Simulink integration do not arise. As the LET is always greater than zero, there is an actor without a direct dependency between inputs and outputs. Consequently, the DE director is able to resolve the loop and to process the TDL actions represented as events in the correct order and potentially interleaved with the plant.

\textbf{A Comparison between the Simulink\textsuperscript{®} and the Ptolemy II integration}

In the Simulink integration, the developer models the functionality (task implementations and the plant) as usual with Simulink blocks. Timing, mode switching logic, and the overall application data flow are expressed in TDL with the TDL:VisualCreator tool. The TDL toolchain automatically creates a simulation model that contains data flow and timing information. On the other hand, the open and extensible architecture of Ptolemy facilitates the expression of all TDL semantics directly in the model. In Ptolemy, the timing specifications of TDL are expressed as properties of the respective actors (i.e. within the simulation model) whereas in Simulink we abstract from both the simulation and the execution platform.

Both approaches extend the existing simulation framework with new blocks (actors) that ensure the TDL semantics during the simulation. In Simulink, the timing requirements are encoded in a
static E-code representation and enforced by an E-machine, resulting in a low computational overhead. In Ptolemy, TDL actions, such as releasing a task activation or performing a mode switch, are generated dynamically and are represented as discrete events. They are enqueued in the event queue of the DE director, which then schedules the appropriate TDL actions at their corresponding time instants. This independence of E-code may make experiments with future TDL extensions more straightforward, because testing new features with Simulink potentially requires changing the compiler, the E-code instruction set, and the E-machine.

Related Work

There is a Simulink integration of another LET-based language called HTL [19]. The HTL compiler is capable of compiling an HTL description into a Simulink model, which is then equipped with functionality for the control laws. The Simulink integration supports the hierarchical structure of HTL descriptions, but restricts communication with the environment (plant model) to a single module, which limits the support for simulating distributed applications. Although HTL, similar to TDL, is based on an E-code variant, the HTL Simulink integration does not follow an E-machine approach but uses standard built-in MATLAB/Simulink blocks. This results in a simulation that does not exhibit the same behavior as the execution of the generated code, since the timing is distorted in several situations (e.g., by Unit-Delay blocks).

Simulink is closely related to synchronous languages. The simulation engine of Simulink executes subsystems implementing controller functionality with 0 duration in time. However, the semantics of Simulink are not defined formally [20], whereas synchronous languages are based on strict formal definitions and aim at formal verifiability. In several approaches, synchronous languages are combined with Simulink:

Argos, a synchronous language with a rigorously defined graphical notation, is prototypically
embedded in Simulink as a less powerful but also less complicated alternative to Stateflow [21].
The implementation assumes that outputs calculated in response to an event are provided immediately. An extension [22] accounts for computational delays and corresponds to the implementation of the synchronous languages Argos and Esterel, where the response must be provided before a next event occurs. In both approaches, a synchronous program is embedded in Simulink.

The work in [23] focuses on the conversion of a Simulink model into a synchronous program. Simulink systems consisting of a predefined subset of discrete-time blocks are translated automatically into Lustre, such that the original Simulink behavior is preserved. This enables the usage of tools for formal validation, simulation, synthesis, etc. Subsequent work [24] also includes Stateflow blocks. The authors of [25] apply these results in the context of distributed systems. They present a layered end-to-end approach by translating Simulink models to the Lustre-based modeling environment SCADE in a first step. In a second step, the Lustre program is annotated in order to define timing assumptions and requirements and to specify the mapping to individual nodes. Finally, the application is mapped to a Time Triggered Architecture (TTA) cluster [26].

The MathWorks product SimEvents® [4] extends Simulink with a mechanism for discrete-event simulation. This allows for simulating models that comprise continuous-time, discrete-time, and discrete-event components. Similar to the TDL integration with Ptolemy, an event-based scheduling can ensure the correct LET semantics of tasks in Simulink. However, based on experience gained in previous work, we doubt that a manual modeling of LET semantics scales to complex applications with multi-modal behavior. A future research project could attempt to combine SimEvents with the TDL-based model translation described above. Our Simulink
integration could especially benefit from the event-based approach to simulate asynchronous activities of TDL. It is also conceivable that the E-machine is not sample-based but is itself triggered asynchronously by events.

TrueTime [34] is a Simulink toolbox for simulating networked control systems. It facilitates the co-simulation of control tasks, network transmissions, and continuous plant dynamics. TrueTime addresses the fact that traditional control design in MATLAB/Simulink often disregards temporal behavior and introduces a real-time kernel to be simulated in parallel with the plant. The kernel block is implemented as an S-Function that simulates a flexible real-time kernel, provides support for A/D and D/A converters, for network connections and interrupt channels. It is event-driven and supports both periodic and aperiodic tasks. Different scheduling mechanisms, such as rate-monotonic or earliest-deadline-first, may be used. Control tasks are implemented in MATLAB function, C, or Simulink blocks. For simulating distributed applications, network blocks support different network types. TrueTime allows a developer to experiment with various scheduling mechanisms, and to investigate the true timing behavior of control applications in Simulink, taking into account latencies, execution times, jitter, and network effects. It requires execution times, distribution, scheduling mechanism to be known in advance in order to approximate the behavior on the real execution platform. It is a platform-centered approach, whereas simulating TDL modules abstracts from the platform and the network topology.

The TDL domain in Ptolemy II is related to the experimental Giotto domain in Ptolemy II [27]. The Giotto domain is designed based on basic Ptolemy II software components, whereas the TDL domain leverages the existing DE domain. The implementation of the TDL domain reflects the distinction between the fundamental concepts (LET, modes) and the manner in which these concepts are used (the operational semantics). The implementation is two-layered: the basic layer
pertains to 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 forms the basis of domain controllers for any other time-triggered programming models (including Giotto) by extending the basic layer.

Conclusion

The Logical Execution Time (LET) allows the explicit specification of timing behavior independent of a specific platform, that is, LET abstracts from the physical execution time and thus from the execution platform and the communication topology. Thus, LET-based systems exhibit behavior equivalent or close to the platform without the necessity to specify platform details. Another significant advantage compared to the state-of-the-art is that the resulting embedded real-time systems generated from a LET-based language such as TDL are deterministic, composable, and portable. In this chapter we presented the core concepts of TDL and its integration in two quite different modeling and simulation environments: MATLAB/Simulink and Ptolemy II. In both cases the integration concept and its implementation ensure that the simulation of time-triggered tasks exhibits the same behavior as the execution of the generated code on any potentially distributed hardware platform.

References


ming language design and implementation (PLDI), pp. 315-326, ACM, 2002.


35. Ben Denckla. Many cyclic block diagrams do not need parallel semantics. SIGPLAN Not. 41, 8, pp. 16-20, August 2006.