Timing Definition Language (TDL) Modeling in Ptolemy II

Stefan Resmerita, Patricia Derler and Wolfgang Pree

Department of Computer Science
University of Salzburg
Austria

Technical Report 21
June 19, 2008

Abstract

This work presents a modeling and simulation platform for embedded control systems where real-time requirements are explicitly specified in the system model by using the Timing Definition Language (TDL). TDL employs the Logical Execution Time (LET) abstraction to specify platform-independent execution times of periodic time-triggered computational tasks, with the main aim of achieving time and value determinism of composable embedded software. For a certain execution platform, this aim is attained if the software components can be suitably scheduled for execution so that the LET specifications are satisfied. In general, an embedded control system contains also concurrent computations triggered by environment conditions (dynamic events), which share the same execution platform. This makes static schedulability analysis for the LET-based part hard to achieve. Thus, a simulation platform is needed to investigate the interaction between time-triggered components and components triggered by dynamic events. We have implemented such a simulation platform by extending the Ptolemy II framework, thus leveraging models of computation that are already available in Ptolemy II, such as the modal model and the discrete event domains.

1 Introduction

The work described here represents a significant step towards achieving a modeling and simulation environment for models of embedded software consisting of concurrent components, some of which must be executed according to the time-triggered computational model defined by the Timing Definition Language (TDL), the other components being triggered by dynamic events.

TDL specifications can be naturally expressed in Discrete Event (DE) models, where communication between components is carried out by means of discrete events placed on a common timeline. Thus, the first step towards achieving the aforementioned objective is to enable simulation of DE models containing TDL specifications. To this end, we build on the Ptolemy II software system [1], developed at the University of California at Berkeley. Ptolemy II enables simulation of computational systems involving multiple models of computation, being thus well positioned to deal with embedded control systems, which generally consist of distributed collections of interacting event-triggered computational tasks that also interact with a physical environment, which can be modeled by differential equations. In particular, Ptolemy II enables the simulation of Discrete Event (DE) and continuous time (CT) components.

This report presents the TDL domain in Ptolemy II, that is, the add-on Ptolemy software components which allow the specification and simulation of DE models with TDL semantics. The TDL domain implements a restricted DE semantics according to TDL specifications. It allows leveraging other Ptolemy domains to experiment with heterogeneous models involving TDL components. The TDL domain will also be used for evaluating future TDL developments.

The report is structured as follows. Section 2 describes the basic concepts of TDL and Ptolemy II. Section 3 describes the implementation of the TDL domain in Ptolemy, and Section 4 shows examples of using the TDL-Ptolemy components. Related work is described in Section 5.

2 Background

In this section, we briefly describe the TDL language and the Ptolemy II software. For more detailed information we refer the reader to the provided references.

2.1 Timing Definition Language (TDL)

TDL [7] 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 [3]. While TDL is conceptually based on Giotto, it provides extended features, a more convenient syntax, and an improved set of programming tools.

The LET associated with a computational unit, or task, represents the 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 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.

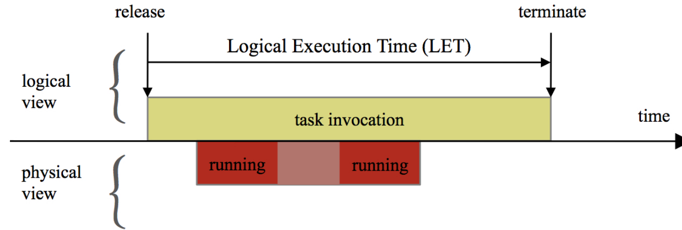

Figure 1: Logical Execution Time

TDL is targeted for applications consisting of periodic software tasks designed to control a physical environment. Thus, some tasks take information from the environment via sensors and some tasks act on the environment via actuators. Tasks that must be executed concurrently are grouped in modes. In TDL, a mode is a set of periodically executed activities such as task invocations, actuator updates, and mode switches. A mode activity has a specified execution rate and may be carried out conditionally. TDL provides a top-level structuring unit called a *module*, which is a logically coherent group of sensors, actuators and modes. Modules are useful especially in dealing with distributed applications, which can be decomposed in loosely-coupled components executing in parallel on different processors.

A sample TDL module is presented in Figure 2, which contains the module definition in TDL code together with a schematic representation of the main parts. The module contains one sensor variable, one actuator variable, and two modes. The main mode specifies a task invocation activity, an actuator update and a conditional mode switch, each of which must be executed once per mode period, which is every 5ms in this example. This implies that the task's LET is 5ms. The actuator is updated with the task output value at the end of the LET.

One aspect in which TDL differs from Giotto at the mode level 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.

2.2 Ptolemy II

Ptolemy II is the software infrastructure of the Ptolemy project at the University of California at Berkeley [1]. The project studies modeling, simulation, and design of concurrent, real-time, embedded systems. Ptolemy II is an open source tool written in Java which allows modeling and simulation of systems adhering to various models of computation (MoC). Conceptually, a MoC represents a set of rules which govern the execution and interaction of model components. The implementation of a MoC is called a *domain* in Ptolemy. Some examples of existing domains are: Discrete Event (DE), Continuous Time (CT), Finite State Machines (FSM), and Synchronous Data Flow (SDF).

Ptolemy is extensible in that it allows the implementation of new MoCs. Most MoCs in Ptolemy support actor-oriented modeling and design, where models are built from actors that can be executed and which can communicate with other actors through ports. 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 3 shows a sample Ptolemy model.

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.

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

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

    start mode main [period=5ms] {
      task
        [freq=1] inc(); // LET=5ms/1=5ms
      actuator
        [freq=1] a1 := inc.o;
      mode
       [freq=1] if exitMain(s1) then freeze;
    }

   mode freeze [period=1000ms] {}
}
```

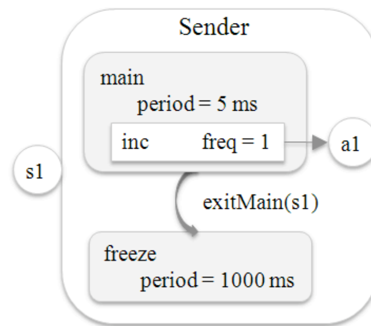

Figure 2: Sample TDL Module

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.

3 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

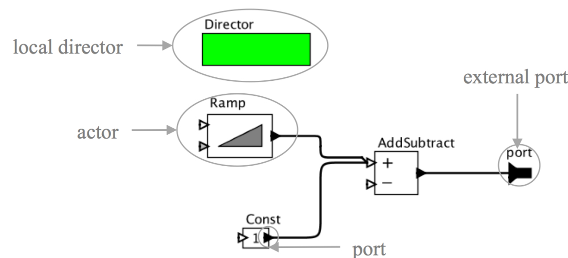

Figure 3: Example of a Ptolemy model

only one mode can be active at a time. Mode switches in modal models have the same semantics as mode switches in TDL. Moreover, the graphical representation of modal models can be reused for TDL modules. 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 *TDLModuleDirector*) 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.

3.1 Implementation

The TDL domain was defined by deriving it from existing Ptolemy components, to maximize reuse. The diagram in Figure 4 shows the TDL-specific components (white boxes) and their dependency relationships with existing Ptolemy II components (gray boxes). Most of the relationships are class inheritance.

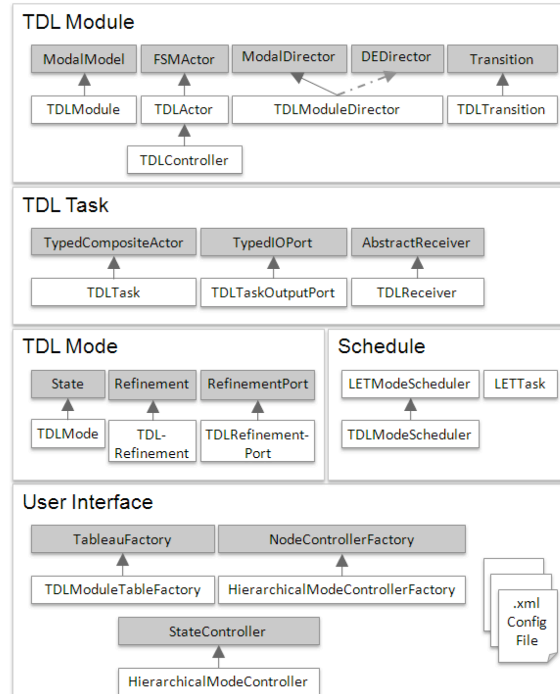

Figure 4: TDL Implementation in Ptolemy II

The *TDLModule* actor is derived from the modal model. The mode controller and the module director of a *TDLModule* are derived from the equivalent classes in the modal model. The execution semantics of TDL is implemented in the *TDLModuleDirector*. Currently, all TDL activities are periodic, and a TDL module does not change during simulation. Thus, the *TDLModuleDirector* can determine a static schedule of all activities for each mode. However, we intend to change this design to an event-driven approach, as described in Section 6. A *TDLTransition* is derived from an FSM transition because it needs a parameter to specify when the transition can be tested.

The *TDLMode* is derived from an FSM state, having a parameter for the mode period and imposing the restriction that the state can have exactly one refinement, which specifies the behavior of the mode. Refinements of TDL modes need special ports which are actuators in TDL, having associated frequencies and initial values. The schedule of a mode is determined by two components: the *LETModeScheduler*

and the *TDLModeScheduler*. The former generates a basic schedule derived from each task's LET, the invocation period and the offset for the first invocation of a task. At every LET endpoint within the mode period, three actions are planned in the following order:

1. Update task output ports. The output ports of each task with the LET ending at this time are updated with values calculated in the previous execution.
2. Update task input ports. The input ports of each task with LET starting at this time are updated with values from sensor variables or from output ports of other tasks.
3. Execute LET tasks. Each task with LET starting at this time is executed.

Figure 5 shows an example of three tasks, T1, T2 and T3, with different LETs and invocation periods. The gray bars show the LETs of the tasks. The arrow at the beginning of a gray bar describes the updating of input ports, the arrow at the end of a gray bar represents the updating of output ports. The LET schedule that is generated from this example can be found in Figure 6.

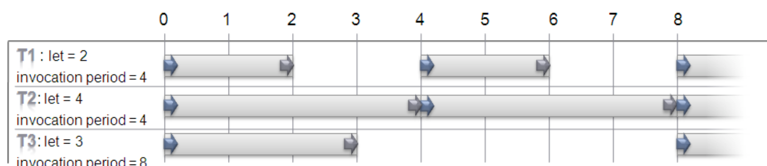

Figure 5: TDL Tasks Example

The *TDLModeScheduler* adds actions specific to the TDL semantics to the LET-based schedule. The combined operations executed at every LET endpoint are:

1. Update task output ports.
2. Update actuators connected to the ports involved in step 1.
3. Test for mode switches. If a mode switch guard evaluates to true, start from the beginning of the schedule in the target mode.
4. If no mode switch evaluates to true in the preceding step, then deal with the fast tasks. A fast task is executed in logically zero time, so it is treated differently than a regular task (with non-zero LET). First, input ports of fast tasks are read, then the fast tasks are executed, then their output ports are updated and finally connected actuators are updated at the same time step.
5. Update input ports of LET tasks.
6. Execute LET tasks.

Time	Scheduled actions	Example		
0=8	Update tasks output ports		T2_out	
	Update tasks input ports	T1_in	T2_in	T3_in
	Execute tasks	T1	T2	T3
2	Update tasks output ports	T1_out		
	Update tasks input ports			
	Execute tasks			
3	Update tasks output ports			T3_out
	Update tasks input ports			
	Execute tasks			
4	Update tasks output ports		T2_out	
	Update tasks input ports	T1_in	T2_in	
	Execute tasks	T1	T2	
6	Update tasks output ports	T1_out		
	Update tasks input ports			
	Execute tasks			

Figure 6: TDL Schedule Example

A *TDLTask* specifies the basic unit of execution, with the associated LET. In general, this is an opaque composite actor. Currently, we support only SDF actors in TDL tasks.

TDLModule actors are best used in the DE domain. However, since a *TDLModule* is a restricted type of a DE actor, it can be used in any domain where a DE actor can be used. When a TDL module actor is fired, the actor obtains the current model time from the outer director and checks if it has to perform the operations detailed above. If so, steps 1 and 2 are executed, after which the actor schedules with the outer director a re-firing at the current time instant, and returns control to the outer director. Upon being re-fired, the module actor executes steps 3 and 4 and then requests again to be re-fired at the current time instant and returns. Steps 5 and 6 are executed in the second re-firing. By relinquishing the execution control as described above, the module actor allows the outer director to propagate output values in the enclosing model, which may contain a zero-time path from an output of the actor to one of its inputs. Notice that a fast task can be used to break a loop with zero-time delay (which would be rejected by a DE domain controller).

In order to represent TDL actors graphically, minor changes of Vergil, the visual editor framework in Ptolemy, were accomplished.

4 Examples

The usage of the TDL domain in Ptolemy can be illustrated by a first straight-forward example. Figure 7 shows a DE model containing two TDL modules and other DE actors to produce sensor values and display the output. The plots displaying the output values are shown in figure 8.

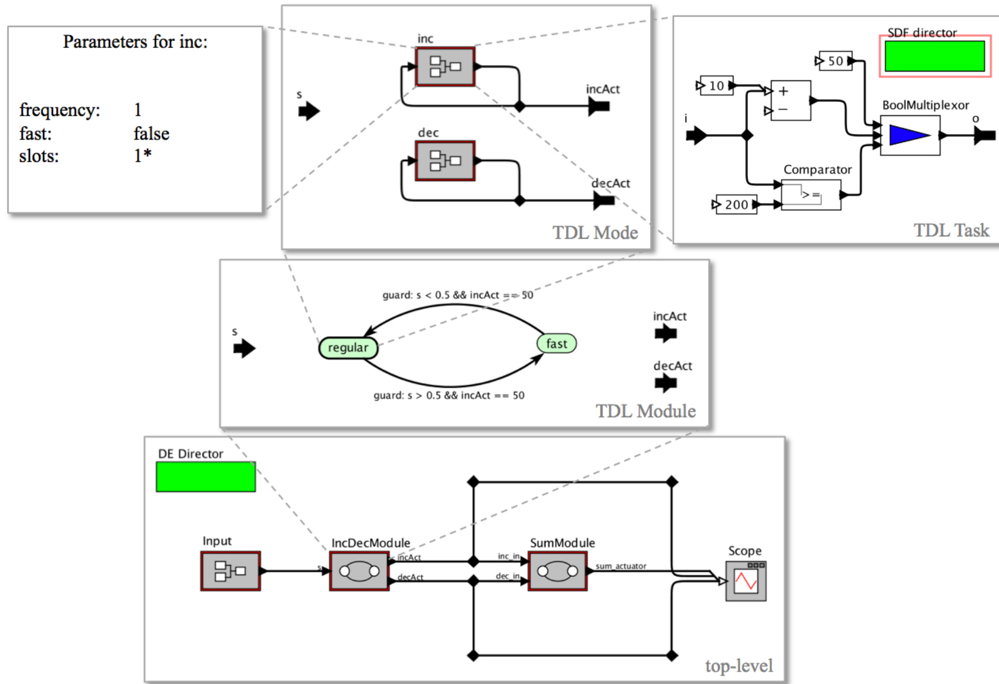

Figure 7: IncDec example: model

The first TDL module (*incDecModule*) reads sensor inputs and produces two output values. The outputs are produced by two tasks, an increment task (*incTask*) and a decrement task (*dec*). The increment task increments an initial value until this value reaches 200 and then resets the value to 50 (see 2 in figure 8). The decrement task behaves similarly but decrements the value until it reaches 50 and then resets the value to 200 (see 3 in figure 8). The initial output value of the increment task in this example is 50 and for the decrement task is 200. The module consists of two modes, one called "regular" and one called "fast" (*fastDec*). Both modes contain the increment and the decrement tasks. In the regular mode, both tasks have the same LET. In the fast mode, the LET of the decrement task is half of the value used in the regular mode. A switch between these two modes is triggered by a sensor which

produces a sinewave signal (see plot number 1 in Figure 8). The conditions for switching between the two modes are specified in the guard expressions of the transitions. The condition for the mode switch consists of two parts: the sensor value must be greater or lower (dependent on the source mode) than 0.5 and the signal produced by the increment task must be 50.

The second TDL module (*sumModule*) produces the sum of the output signals from the first module (see plot number 4 in Figure 7). When the first TDL module is in the regular mode, the sum of the increment and decrement tasks is a constant signal. When the fast mode is active, the sum is a sawtooth signal.

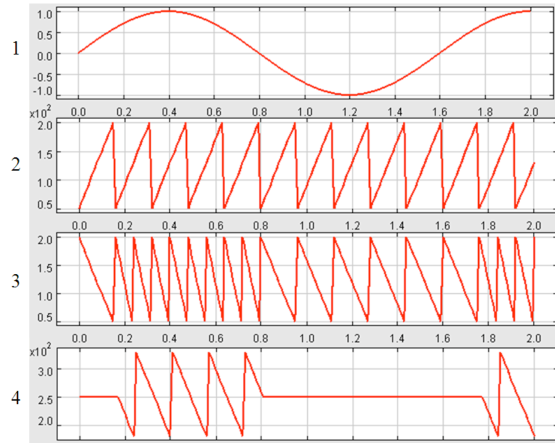

Figure 8: IncDec example: output

The example in figure 9 shows an interesting behavior of TDL modules that are connected to other actors that execute in zero logical time. According to the schedule of a TDL module, at time t, task output ports are updated and actuators are written. At the same time, input ports are read. In the example, a new value on the actuator causes a new input for the TDL module at the same time t. Thus, after the actuator is updated, the DE actor that executes in logically zero time must be executed. This is done by having the TDL module relinquish control to the top level director after updating the actuators. The director can now fire the zero-time actor. Then the module actor is re-fired (at the same time t), in order to read the newly updated input.

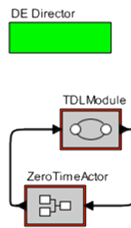

Figure 9: Example: Zero Time DE Actor

The TDL domain in Ptolemy has already been tested with more complex models, such as the one shown in Figure 10.

The Ptolemy model represents a control system for vehicle active rear steering, where rear wheels are steered automatically depending on the steering of the front wheels and on the vehicle dynamics. The model contains controller actors (VehicleDynamics and RearActuatorController) representing TDL modules and the plant (Vehicle) as a continuous time actor.

5 Related Work

The closest existing related work is the experimental Giotto domain in Ptolemy [2] . Although TDL shares with Giotto the fundamental concepts of LET and mode-based computation, there are significant

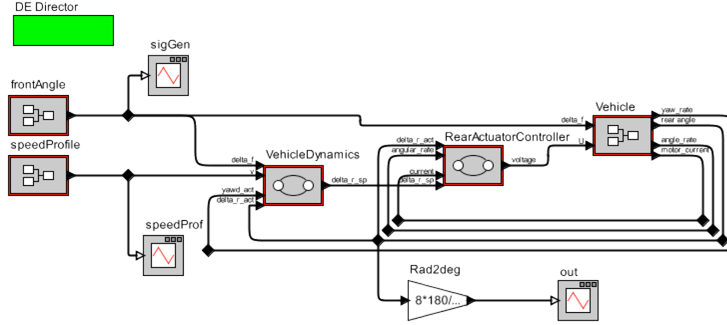

Figure 10: Example: Active Rear Steering

semantical differences between the two languages. From the viewpoint of Ptolemy integration, the main differences between the TDL domain and the Giotto domain are in the purpose, the relationship with existing modeling techniques, and the implementation. Our goal is to evolve the current TDL domain into a more comprehensive model of computation, where platform attributes (such as execution times, scheduling) are also taken into consideration. Ptolemy II already provides a model of computation aiming at this direction, implemented in the Timed Multitasking (TM) domain [4] . We envision integrating TM features into the TDL Ptolemy domain, to achieve the simulation of executions of TDL models on specific platforms. From the viewpoint of modeling techniques, we consider that LET specifications are best expressed at the level of DE models. Thus, a TDL model (that is, a model containing TDL components) is regarded as a particular case of a DE model, with additional constraints on how events destined to, or generated by, TDL components are placed on the timeline. This is reflected in our implementation of the TDL domain, which leverages the existing, and well studied, DE domain. In contrast, the Giotto domain is designed based on basic Ptolemy software components. Moreover, our implementation 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.

A commercially available tool suite deals with modeling, simulation 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. A TDL compiler is provided, which targets a real-time virtual machine, called the TDL:E-Machine. To deploy the TDL model on a platform, an implementation of TDL:E-Machine is needed for the platform. 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.

The tools described above have also been integrated in Matlab(R)/Simulink(R), in order to: (1) Use the extensive Simulink block libraries to define the functional part of the model (tasks and guard functions), (2) Employ the code generation facilities to generate C code for the functional part, in addition to the e-code (the code for the TDL:E-Machine) for the timing part, and (3) Simulate the TDL model. Figure 11 depicts the TDL tool chain.

The approach to simulating TDL components by using Ptolemy II is different than the Simulink-based approach in at least two important respects:

- The above tool suite employs a Simulink implementation of the TDL:E-Machine. In contrast, no virtual machine is used in the TDL domain for Ptolemy. TDL specifications are expressed as properties of Ptolemy actors and the TDL domain uses these properties to generate an appropriate

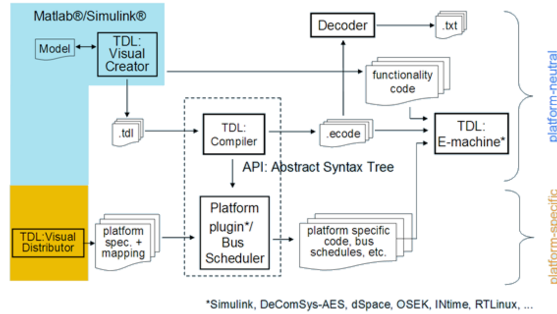

Figure 11: TDL Tool Chain

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.

- The TDL Ptolemy domain can be further specialized for various types of platforms, for which execution times and scheduling policies can be simulated. This will enable simulation of systems where computational tasks outside TDL jurisdiction (such as tasks triggered by dynamic events) compete for the same platform resources with TDL components (which are periodic time-triggered tasks). This significantly enlarges the spectrum of application areas of the simulator.

Although various research efforts deal with designing and scheduling mixed time-triggered and event-triggered systems (e.g., [5] [6]), the authors are not aware of any other simulation environments dedicated to heterogeneous models involving LET-based time-triggered components.

6 Conclusions and Further Work

The TDL Ptolemy domain is our first significant step towards simulating heterogeneous models involving (but not limited to) periodic time-triggered tasks with logical execution times. We are currently changing the TDL module actor to use the DE model of execution of mode activities, instead of the static schedules. The new TDL module actor has an event queue, where events (time stamped activities) are placed by the active mode according to the operations between two consecutive mode changes. This enables a departure from the purely periodic setup. Hence, the TDL module actor will be a mixture between a modal model and a DE actor.

We further plan to integrate features of Ptolemy's Timed Multitasking domain into the TDL domain, to achieve simulation of TDL models corresponding to different types of platforms. We intend to get closer to the platform level, where the execution of heterogeneous models becomes increasingly complex and TDL requirements are generally not guaranteed by static analysis.

References

[1] C. Brooks, E.A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, H. Zheng (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.

[2] C. Brooks, E.A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, H. Zheng (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.

[3] T. Henzinger, C. Kirsch, M. Sanvido, W. Pree. From Control Models to Real-Time Code Using Giotto. *IEEE Control Systems Magazine 23(1)*, February 2003.

[4] J. Liu and E. A. Lee, Timed Multitasking for Real-Time Embedded Software, *IEEE Control Systems Magazine 23(1)*, February 2003.

[5] Pop, T., Eles, P., and Peng, Z. 2002. Holistic scheduling and analysis of mixed time/event-triggered distributed embedded systems. In Proceedings of the Tenth international Symposium on Hardware/Software Codesign (Estes Park, Colorado, May 06 - 08, 2002). CODES '02. ACM, New York, NY, pp. 187-192.

[6] Scaife, N. and Caspi, P. 2004. Integrating Model-Based Design and Preemptive Scheduling in Mixed Time- and Event-Triggered Systems. In Proceedings of the 16th Euromicro Conference on Real-Time Systems (June 30 - July 02, 2004). ECRTS. IEEE Computer Society, Washington, DC, pp. 119-126.

[7] Templ, J. (2007) TDL - Timing Definition Language 1.4 Specification. Available at http://www.preetec.com/

[8] The TDL tool chain. Available at http://www.preetec.com/