Visual Modeling of Real-Time Behavior

Andreas Naderlinger, Wolfgang Pree and Josef Templ

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

Abstract. This paper describes the visual representation of the Timing Definition Language (TDL), a high-level textual description language for timing aspects of embedded real-time systems. For this purpose we have designed and implemented the so-called TDL:VisualCreator tool. The paper presents the core concepts of TDL, which comprises now both the time-triggered and event-triggered paradigm, and exemplifies the textual and its corresponding visual representation. We also point out how the TDL:VisualCreator tool is integrated with the visual development and simulation environment MATLAB/Simulink.

1 Introduction

Traditional development of software for embedded systems is highly platform specific. Ever more ambitious requirements and the increased complexity of the resulting software make a more platform independent "high-level" programming style mandatory. In case of real-time software, this applies not only to functional aspects but also to the temporal behavior of the software. In particular for distributed systems, dealing with time, however, is not covered appropriately by any of the existing component models for high-level languages.

The Timing Definition Language (TDL) [1] is such a high-level software description language. It allows the explicit and platform independent timing specification of multi-mode multi-rate real-time components. TDL is based on the logical execution time (LET) abstraction introduced in the realm of Giotto [2], which means that the observable temporal behavior of a TDL task is independent of its physical execution.

TDL adheres to the AUTOSAR [3] vision as the timing behavior is defined independent of a specific execution and communication platform. It allows you to develop embedded real-time software components once and deploy them on any potentially distributed platform that offers sufficient computing and communication resources. However, TDL goes one significant step beyond AUTOSAR, as it offers a specification of the timing behavior that allows fully automated generation of efficient production code from the TDL program for a specific platform. This is in stark contrast to AUTOSAR whose specification is not adequate for fully automated code generation. Time-triggered communication schedules, e.g., have to be defined manually which is error-prone and which requires tedious testing efforts against the AUTOSAR model.

TDL is also well integrated in MathWorks' [4] MATLAB/Simulink, the de-facto standard modeling and simulation environment in the automotive domain. This allows the simulation and automatic code generation of TDL based applications. Simulink provides an interactive graphical interface.

This paper describes a graphical representation of TDL and the development of time- and event-triggered applications with our TDL:VisualCreator tool in Simulink.

2 Visual Modeling of TDL Modules in MATLAB/Simulink

TDL itself offers a textual notation for defining the timing behavior of time- and event-triggered activities and the data flow between them. TDL-based applications may be composed of several components called TDL modules. A TDL module forms a unit that consists of sensors, actuators, and modes. A mode is a set of periodically executed activities. The activities are task invocations, actuator updates, and mode switches. All activities can have their own rate of execution and all activities can be executed conditionally. A set of TDL modules corresponds to a set of independent automatons that execute their time-triggered activities in parallel and that can switch their modes independently. The tasks represent the functionality, particularly the control laws. The task implementation is not done in TDL, but in an imperative programming language, such as C, or modeled in MATLAB/Simulink. Fig. 2 shows two TDL modules as *TDL Module blocks* embedded in a Simulink model.

Fig. 1 lists the textual representation of a sample module *Sender*. This simple module comprises one sensor *s1*, two actuators *a1* and *a2*, as well as a task *inc* and two operational states *main* and *freeze* called modes. Mode *main*, which is defined as the start mode, executes three activities with a period of 10 ms. Within one such period the activity sequence consisting of task *inc* and actuator *a2* is executed 5 times. Thus the LET of the task is 2 ms (10/5). The actuator *a1* is updated with the output value of *inc* at the same rate. In contrast to *a2*, *a1* is not updated until the task's LET has expired. Additionally, once every period the sensor *s1* is evaluated in a mode switch condition (implemented in *exit*). When *exit* evaluates to *true*, the module switches to the second mode *freeze*. This textual description of TDL covers three different aspects: (i) declarations, (ii) data-flow semantics, and (iii) state transitions.

```
module Sender {
 sensor double s1 uses getS1;
 actuator int a1 uses setA1;
 actuator int a2 uses setA2;
 public task inc {
  output int o := 10;
  uses incImpl(o);
  uses [release] incRelease(o);
 }
 start mode main [period=10ms] {
  task [5] {inc(); a2 := inc.o}
  actuator [5] a1 := inc.o;
  mode [1] if exit(s1) then freeze;
 }
 mode freeze [period=1000ms] {}
}
```

Fig. 1. Textual representation of a TDL sample module

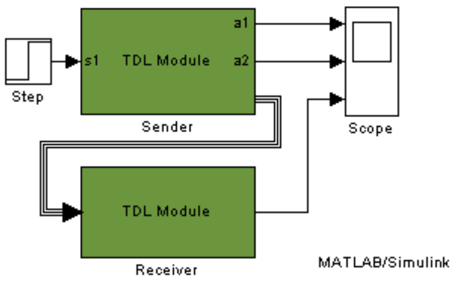
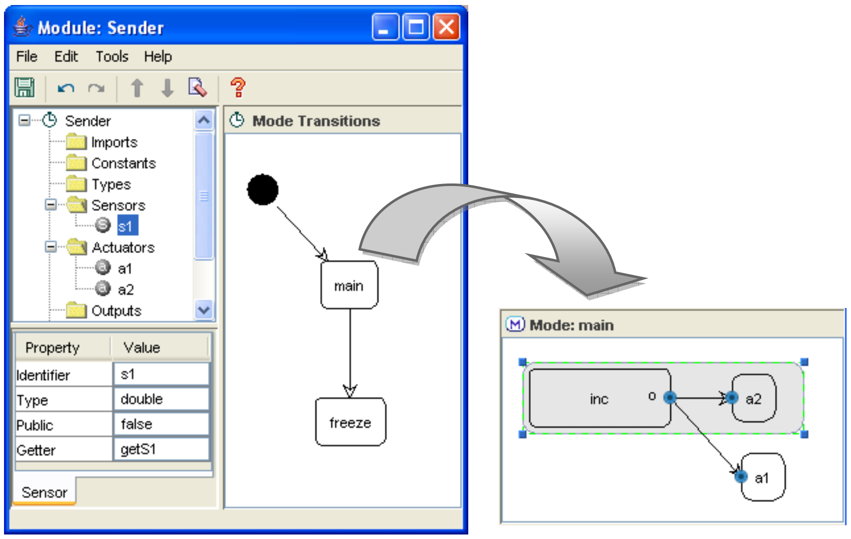


Fig. 2. TDL Modules in MATLAB/Simulink

Simulink developers are used to design their application within an interactive and graphical environment. They click their way to a control application rather than writing code. For supporting a visual and interactive modeling of TDL applications we designed and implemented the so-called TDL:VisualCreator tool [5] (see Fig. 3). It is a syntax-driven editor that offers exactly the same TDL constructs as the textual TDL version.

The tool can either be used as a stand-alone application or as a seamlessly integrated add-on for the visual development and simulation environment MATLAB/Simulink. In the latter case, the application's functionality (i.e. the implementation of external functions such as *inc* or *exit*) can be modeled with Simulink (see Fig. 4). This has the advantage that the behavior of the application can be simulated. Furthermore, the LET abstraction guarantees that the simulated behavior is equivalent to the behavior on any, potentially distributed, execution platform that provides enough resources to pass a time-safety check.



Property	Value
Identifier	s1
Type	double
Public	false
Getter	getS1

Fig. 3. The TDL:VisualCreator tool

The main window of the tool provides three panes that allow the visualization and the editing of the aspects mentioned above. The upper left part of the window shows a tree representation of the TDL module, with the module name as the root and its elements as sub-nodes of the corresponding folders. Below the tree there is table that shows a list of all properties of the currently selected item in the tree. The pane on the right side of the window provides the context specific editing capabilities for modeling data-flow or state transitions.

The ordering of the individual elements in the tree view follows the TDL language specification [1]. The first part of a TDL program (up to but excluding the mode section) is purely declarative. Therefore, the visualization capabilities are limited and are restricted to the tree based representation of TDL constructs in combination with a tabular listing of their properties.

Using the tree view, a developer may import from other TDL modules, declare constants, types, sensors, actuators, global ports, and tasks. Import relations are visualized by Simulink bus connections between the individual modules (see Fig. 2). TDL's built-in data types are mapped to corresponding Simulink types, arrays and structured types correspond to multiplexed signals respectively Simulink buses. Sensors and actuators are synchronized with *In-* and *Outports* of the *TDL Module block*.

The actual implementation of the task functionality is not part of TDL. Instead we use MATLAB/Simulink to implement task functions. Fig. 4 shows a Simulink subsystem containing a sample implementation for the function *incImpl*. A function parameter is automatically mapped to an *In-* respectively *Outport*. Application developers are free to use any of Simulink's non-continuous library blocks with inherited sample time for modeling the computational part of the application. The Simulink subsystem instantaneously reflects changes which were applied to the TDL model (e.g. deleting or renaming a port), and vice versa. For a more convenient editing, the TDL:VisualCreator tool also supports the automatic creation of TDL task declarations based on existing (legacy) Simulink subsystems.

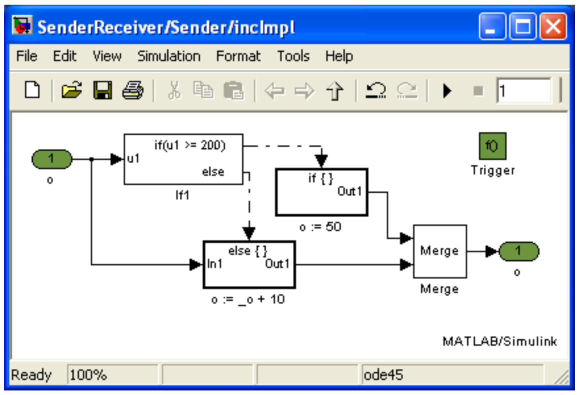


Fig. 4. Using MATLAB/Simulink to model the task functionality

2.1 Mode Transitions and Time-Triggered Activities

The TDL:VisualCreator in Fig. 3 shows the mode transitions of module *Sender*. This view summarizes all mode switch activities that are defined in any of the modes. In this example there is only one transition from the start mode *main* to *freeze*.

The right part of the figure shows a screenshot of the contents of mode *main*. This view contains every task sequence, task invocation, and actuator update that is defined within a mode. To add activities, the corresponding task, actuator, etc. is dragged from the tree pane and dropped onto the panel. This example, shows the task sequence of task *inc* followed by the actuator update of *a2* as well as the actuator update of *a1*. Data flow is expressed by arrows and the timing behavior is defined in the tabular view. As all activities in a mode run in parallel, the alignment of the individual elements has no influence on the application's behavior.

Activities may be executed conditionally, i.e. they may be guarded by an external function. Such *guards* are created in the tree view and implemented in Simulink subsystems.

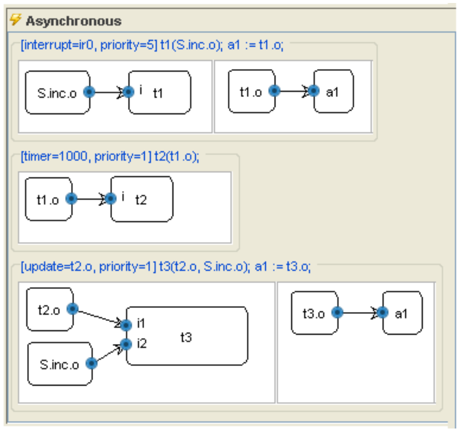
2.2 Event-Triggered Activities

In addition to time-triggered (synchronous) activities, TDL also supports the definition of event-triggered (asynchronous) activities. An asynchronous activity may be a task invocation or an actuator update. Multiple activities may be grouped to a sequence that is triggered by an interrupt, a port update event, or a timer.

Fig. 5 lists a second module *Receiver* that imports module *Sender* and uses its port *inc.o* as data source in asynchronous activities. Fig. 6 shows the corresponding representation in the TDL:VisualCreator tool. The graphical representation exactly follows the structure of the textual description. This separation of activities avoids ambiguities with respect to their ordering.

```
module Receiver {
 import Sender as S;
 // …
 asynchronous {
  [interrupt=ir0, priority=5]
   t1(S.inc.o); a1 := t1.o;
  [timer=1000, priority=1]
   t2(t1.o);
  [update=t2.o, priority=1]
   t3(t2.o, S.inc.o); a1 := t3.o;
 }
}
```

Fig. 5. Module *Receiver*



[interrupt=ir0, priority=5] t1(S.inc.o); a1 := t1.o;

[timer=1000, priority=1] t2(t1.o);

[update=t2.o, priority=1] t3(t2.o, S.inc.o); a1 := t3.o;

Fig. 6. Asynchronous Activities

3 Simulation & Code Generation

When the application developer has finished the modeling phase, that is, timing behavior has been specified using the TDL:VisualCreator and functionality has been implemented with Simulink blocks, the application can be simulated. For this purpose, we automatically generate a simulation model that translates the TDL model into Simulink blocks and links timing and functionality [6]. When the simulation shows satisfactory behavior, code generators like the Real-Time-Workshop Embedded Coder [4] can be used to transform the Simulink block representation into C code. Executed on a (potentially distributed) hardware platform, the TDL runtime system ensures that the time-triggered part of the application exhibits exactly the same behavior as during the simulation.

4 Comparisons and Conclusions

The MathWorks [4] troika consisting of MATLAB, Simulink and Stateflow together with the code generator Real-Time Workshop Embedded Coder is widely used for the visual modeling of dynamic systems with mode logic. However, the inherent fusion of functional and temporal aspects does not allow modeling of systems that can efficiently be mapped to distributed platforms.

The SimTools [7] from Decomsys/Elektrobit and analogous tools such as TTP-Matlink from TTTech [8] represent a Simulink extension that facilitates the simulation and code generation for distributed real-time systems. However, these tools require the tedious modeling of platform details, as they lack the LET abstraction that allows a platform independent development.

This paper describes a graphical notation and its tool implementation which allows for a visual and interactive development of embedded real-time applications in a platform independent way. For the presented modeling tool, TDL:VisualCreator, it was sufficient to combine existing diagram types, i.e. data-flow and state transition diagrams as well as tree- and tabular views, rather than introducing new visual concepts.

We applied two different representations for the data-flow. The first combines all mode activities in a single diagram. While this perfectly shows the relation and interplay of all involved TDL constructs and helps grasping the continuous data flow from sensors to actuators, this representation also entails scalability shortcomings. The second representation, which was used for asynchronous TDL activities, is more structured and follows the textual description.

We described the integration into the widely used visual environment MATLAB/Simulink, which offers optimal synergy. For further details, we refer to [9].

References

1. Templ, J. *Timing Definition Language (TDL) 1.5 Specification.* Technical Report, University of Salzburg, 2009. Available at www.softwareresearch.net
2. Henzinger, T., Horowitz, B., Kirsch, C. *Giotto: A time-triggered language for embedded programming.* In Proc. of EMSOFT, LNCS 2211, pages 166–184. Springer, 2001
3. AUTOSAR. Automotive Open System Architecture. http://www.autosar.org.
4. The MathWorks. www.mathworks.com
5. preeTEC. www.preeTEC.com
6. Naderlinger, A., Templ, J., Pree, W. *Simulating Real-Time Software Components based on Logical Execution Time*. In SCSC '09: Proceedings of the 2009 Summer Computer Simulation Conference, 2009.
7. Decomsys/EB. www.decomsys.com/simtools
8. TTTech. www.tttech.com
9. Naderlinger, A., Pree, W., Templ, J. *Visual Modeling of Real-Time Behavior*. Technical Report, T023, University of Salzburg, 2008. Available at www.softwareresearch.net