Visual Modeling of Real-Time Behavior

Andreas Naderlinger, Wolfgang Pree, Josef Templ

C. Doppler Laboratory
Embedded Software Systems
University of Salzburg
Austria

Technical Report T023
October 2008

Visual Modeling of Real-Time Behavior

Andreas Naderlinger, Wolfgang Pree and Josef Templ
C. Doppler Laboratory Embedded Software Systems
University of Salzburg
Jakob-Haringer-Str. 2, 5020 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 first presents the core concepts of TDL and then for each TDL construct 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 from a user's point of view.

1. Motivation

Traditional development of software for embedded systems is highly platform specific. The hardware costs are reduced to a minimum whereas high development costs are considered acceptable in case of large quantities of devices being sold. However, with more powerful processors even in the low cost range, we observe a shift of functionality from hardware to software and in general more ambitious requirements. A luxury car, for example, contains about 70 electronic control units interconnected by multiple buses and driven by more than a million lines of code. In order to cope with the increased complexity of the resulting software, a more platform independent "high-level" programming style becomes 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] allows a paradigm shift. The timing behavior is defined independent of a specific execution and communication platform. In other words, TDL 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. Automatic code generators create the efficient code from the TDL program for a specific platform. Figure 1 illustrates this portability of TDL components (= modules). TDL components could, for example, run on a single-node system or could be deployed on a FlexRay [2] cluster. The TDL compiler tools guarantee that the behavior specified in TDL is exactly the same on any target platform. If a target platform does not offer sufficient computing and communication resources, the code generators would report that problem and no code would be generated.

TDL is also well integrated in MathWorks' MATLAB and its visual-interactive modeling environment Simulink so that the behavior of TDL components can easily be simulated. The visual-interactive modeling of TDL components is the focus of this paper. Where appropriate we illustrate how TDL and MATLAB/Simulink modeling interact. Beforehand we present the core concepts of TDL in the next section.

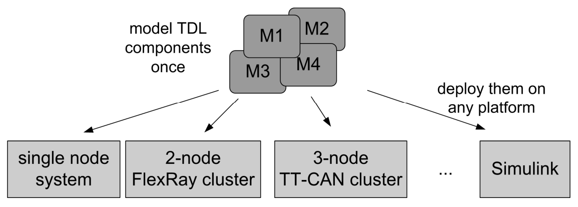


Figure 1. Mapping of TDL components to a specific platform

2. What is a TDL component (module)?

TDL itself offers a textual notation for defining the timing behavior of periodically executed activities. TDL offers a language construct, the so-called module construct, which allows the definition of a TDL component. We use the terms module and component synonymously. 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. Any suitable imperative language can be chosen to implement the functionality. If MathWorks' tools are used, the functionality is modeled in MATLAB/Simulink.

Actuator updates, sensor readings and mode switches are considered to be much faster than task invocations, thus they are executed in logical zero time. Whereas task invocations adhere to the so called Logical Execution Time (LET) semantics explained below.

Logical Execution Time (LET). LET means that the observable temporal behavior of a task is independent of its physical execution [1, 3, 4]. It is only assumed that physical task execution is fast enough to fit somewhere within the logical start and end points. Figure 2 shows the relation between logical and physical task execution.

The inputs of a task are read at the release event and the newly calculated outputs are available at the terminate event. Between these, the outputs have the value of the previous execution. LET provides the cornerstone to deterministic behavior, platform abstraction and well-defined interaction semantics between parallel activities. It is always defined which value is in use at which time instance and there are no race conditions or priority inversions involved.

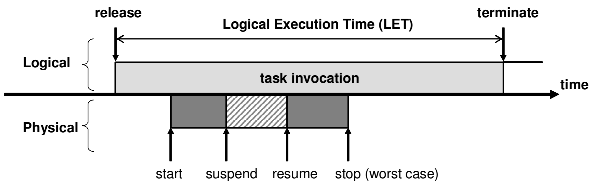


Figure 2. Logical Execution Time

LET introduces a delay for observable outputs, which might be considered a disadvantage. We have extended TDL to allow the avoidance of delays within a TDL component for the benefit of digital controller applications: A task's functionality code may be split in two parts, (1) a fast step and (2) a slow step, where the fast step is executed in logical zero time right at the release time of the task and the slow step is executed regularly. Output ports updated in the fast step are available immediately for actuator updates in such task sequences.

3. TDL modules in MATLAB/Simulink

Modules are the top-level structuring element, the unit of compilation and provide a closed namespace for all language constructs [1]. In particular, modules allow the definition of sensors and actuators, as well as tasks that encapsulate the functionality of the components. Figure 3 lists the textual representation of a sample module *Sender*.

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

Figure 3. Textual representation of a TDL sample module

This simple module comprises one sensor *s1*, one actuator *a1*, 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 task *inc* is executed 5 times (frequency=5). Thus the LET is 2ms (10/5). The actuator *a1* is updated with the output value of *inc* with a frequency of 1 ms. Additionally, once every period the sensor *s1* is evaluated in a mode switch condition (implemented in *exitMain*). When *exitMain* evaluates to *true*, the module switches to the second mode, *freeze*. A detailed description of the single constructs will follow below. This textual description of TDL covers three different aspects: (i) declarations, (ii) data-flow semantics, and (iii) state transitions.

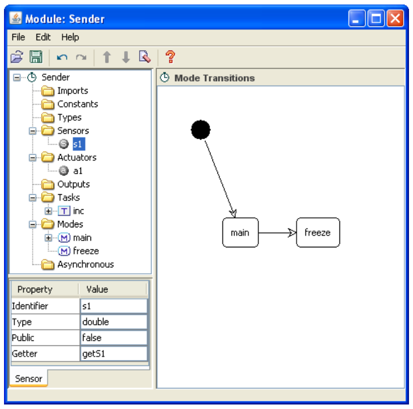
For supporting a visual and interactive modeling of TDL applications we designed and implemented the so-called TDL:VisualCreator tool [5] (see Figure 4). It is a syntax-driven editor that offers exactly the same TDL constructs as in the textual TDL version.

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

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 from *The MathWorks*. The MATLAB extension Simulink 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) [6] or TargetLink [7], it has become the de facto standard in many domains.

When using TDL in combination with Simulink, the functionality of the application (i.e. the implementation of the external functions) can be modeled with Simulink. This has the advantage that the behavior of the application can be simulated.

Figure 5 illustrates a Simulink model that contains the collapsed TDL module Sender (represented as green block called *TDL Module block*) as well as two blocks (*Step*, *Scope*) that form the physical environment which interacts with the module. A TDL module uses sensors respectively actuators to communicate with the environment.



Property	Value
Identifier	s1
Type	double
Public	false
Getter	getS1

Figure 4. The TDL:VisualCreator tool

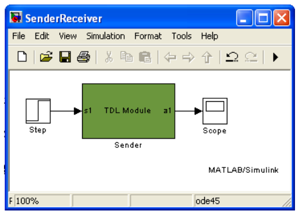


Figure 5. TDL module Sender in a Simulink model

The graphical representation of sensor *s1* and of actuator *a1* in the Simulink model in Figure 5 corresponds to the definition of sensors and actuators in the TDL:VisualCreator tool (Figure 4) and the textual specification in Figure 3. A TDL module block masks the timing specification as well as the functional implementation of the module. A TDL module block is provided by the TDL Library which is integrated in the Simulink Library Browser.

A double-click on the Sender module block in Figure 5 opens the TDL:VisualCreator tool where one can edit the module (Figure 4).

4. Visual representation of TDL

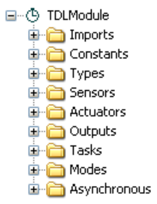
The following section describes the visualization of a TDL module for each language construct. We present them in the order shown in the tree pane in Figure 4, *imports* however are discussed in the end. The first part of a TDL program (up to but excluding the mode section) is purely declarative. Therefore, the visualization capabilities are limited and are restricted to a tree based representation of TDL constructs in combination with a tabular listing of their properties in the property pane.

Module. A module is the top-level element. It forms the root of the tree representation. It provides a namespace for the definition of constants, types, sensors, actuators, global outputs, tasks and modes.

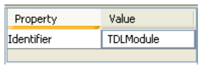
```
module TDLModule {
 //import declarations...

 //constant, type,
 //sensor, actuator,
 //global output
 //and task declarations...

 //mode declarations...
}
```



The only property of a module is its identifier (name).

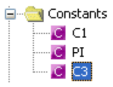


Property	Value
Identifier	TDLModule

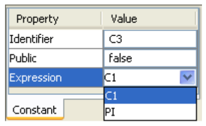
Constants. A constant declaration associates a name with a constant value. The constant value may be denoted as a literal or as the name of another constant.

Constants may be used, for example, for initialization of ports (see below) or for timing attributes.

```
const C1 = 0;
public const PI = 100;
const C3 = c1;
```



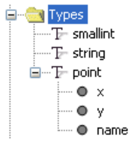
Beside the identifier (name), a constant has two further properties: In the Expression field one can enter a new value or choose one of the previously defined constants from a drop-down box. The property Public determines the constant's visibility outside the enclosing module. In general, public items can be referred to in other modules (see subsection about *imports*).



Property	Value
Identifier	C3
Public	false
Expression	C1
	C1
	PI

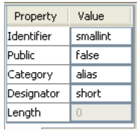
Types. A type declaration associates a name with a type, which may either be an existing type or a new (user-defined) type. A new type is an alias, an array or a structure (similar to struct in C or RECORD in Pascal). TDL provides a set of basic types, which are similar to those found in the programming language Java: *byte*, *short*, *int*, *long*, *float*, *double*, *char*, and *boolean*.

```
type smallint = short;
type string = char[32];
type point = struct{int x,int y,
                    string name};
```



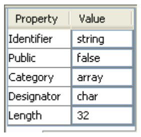
A type declaration requires the definition of the following properties: Identifier (name), Public (enable or disable access from other modules) and Category. Depending on the Category value, additional properties may have to be defined:

Alias. Alias types (e.g. smallint) require the definition of the supplemental property Designator (basic- or user-defined type).



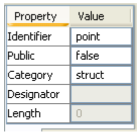
Property	Value
Identifier	smallint
Public	false
Category	alias
Designator	short
Length	0

Array. Array types (e.g. string) additionally require also the Designator and Length property to be set.



Property	Value
Identifier	string
Public	false
Category	array
Designator	char
Length	32

Struct. Struct types (e.g. point) need no additional properties. Elements, i.e. basic types, arrays or already defined structured types, are added by right-clicking on the type in the tree and require the definition of the Identifier (name) and Type property.



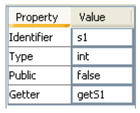
Property	Value
Identifier	point
Public	false
Category	struct
Designator	
Length	0

TDL types are mapped to a corresponding Simulink type. Arrays are represented as *multiplexed signals* and structured types are expressed by *buses*.

Sensors. A sensor (port) declaration defines a read-only variable which represents a particular value of the physical environment of a TDL program.

```
sensor int s1 uses getS1;
```

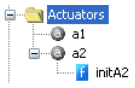
Sensors are typed variables which may be connected with the environment by using a so-called *getter* function. A getter is an external function which returns a value compatible with the sensor's type.



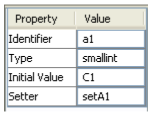
Property	Value
Identifier	s1
Type	int
Public	false
Getter	getS1

Actuators. An actuator (port) declaration defines a write-only variable which controls the setting of a particular value of the physical environment of a TDL program.

```
actuator smallint a1 := C1
  uses setA1;
actuator int a2 init initA2
  uses setA2;
```



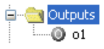
Actuators are typed variables which may be connected with the environment using a so-called *setter* function. An actuator may be initialized either with a constant value or with an external function. *Initializer* functions are added as subnodes of the corresponding (actuator) port.



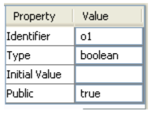
Property	Value
Identifier	a1
Type	smallint
Initial Value	C1
Setter	setA1

Global Outputs. Global output ports may be used as output ports by any task of the module. This declaration on module level is in contrast to ports that are declared within the scope of a task (see below).

```
public output boolean o1;
```



The properties of global output ports are similar to those of an actuator. However, output ports do not have a setter function. Additionally, output ports may be public.

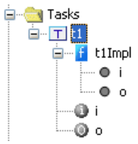


Property	Value
Identifier	o1
Type	boolean
Initial Value	
Public	true

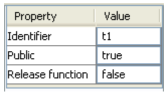
Tasks. A task declaration defines a task, which encapsulates a computation. Tasks provide a namespace for the declaration of input, output and state ports. In addition, a task uses associated external procedures (including arguments), which perform the task's computation.

```
public task t1 {

  input double i;
  output double o;
  uses t0Impl(i, o);
}
```



The corresponding task function is created automatically. Tasks have an identifier and can be public. Public tasks export all of their output ports. The Release function property is explained below.



Property	Value
Identifier	t1
Public	true
Release function	false

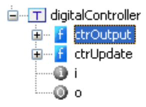
Task ports have a name and type property. In addition, state and output ports can have an initializer, which is again either a constant value or an external function. When adding a new task in- or output port, the corresponding function parameters (represented as function subnodes) are created automatically. However, they can be removed, recreated and reordered by a drag and drop operation to the corresponding task port node. Task functions may also use global output ports as parameters.

Task functionality. The actual implementation of the task functionality is not part of TDL. Instead we use MATLAB/Simulink to implement task functions. By double-clicking a task function tree node, a Simulink subsystem window shows up. For each function parameter an appropriate *In-* respectively *Outport* from the Simulink Library Browser is created. A parameter that refers to a task input port is mapped to an Inport and a parameter that refers to task- or global output port is mapped to an Inport/Outport pair. This extra Inport is optional and allows obtaining the port's value of the previous function invocation. 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. Figure 6 shows an example implementation for the task function *t1Impl* of *task1* with one input port *i* and one output port *o* (green blocks).

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.

Task splitting. For the benefit of digital controller applications, a task's functionality code may be split in two parts: A fast step which is executed in logical zero time, and a slow step, executed regularly (following LET semantics).

```
task digitalCtrl {
 input int i; output int o := 0;
 uses [release] ctrlOutput(i,o);
 uses ctrlUpdate(i, o);
}
```



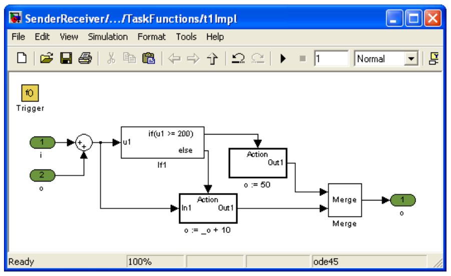


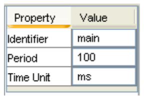
Figure 6. Using MATLAB/Simulink to model the task functionality

This splitting is achieved by setting the task's Release function property to true. This inserts an additional task function node.

Modes. A mode declaration defines a mode, which is a particular state of operation of a TDL module. Modes consist of a set of activities that are executed periodically and potentially in parallel. Activities comprise task invocations, actuator updates, task sequences and mode switches.

```
start mode main [period=100 ms] {
  //activities
}
```

Modes have an identifier and a period which is expressed by the properties period and time unit which is either ms or µs.



Property	Value
Identifier	main
Period	100
Time Unit	ms

Typically, applications consist of multiple modes with a dedicated start mode. When selecting the root node in the tree pane (module node) or the Modes folder node, the editor pane shows the *mode transition editor* (Figure 7). This editor shows the set of declared modes in the module in the form of a state transition diagram.

When a particular mode is selected in the tree pane, the editor pane shows the so-called *mode editor*. This editor contains the graphical representation of all activities of the mode.

Task invocations. The execution of a task in a particular mode is referred to as a task invocation (activity). To add a task invocation, the corresponding task is dragged from the tree pane and dropped onto the mode editor (see Figure 8).

A task invocation is represented as a block with all its in- and output ports on the left respectively right border. Global output ports that are used as parameters in the function of this particular task are also part of this view. The mode editor is intended to visualize the data-flow within a specific mode. When a task is invoked, its input ports are updated according to the specified assignments.

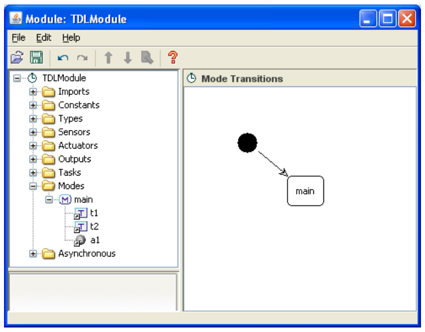
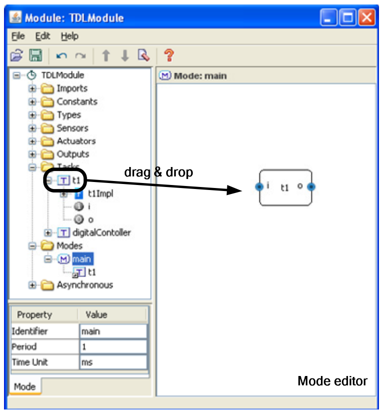


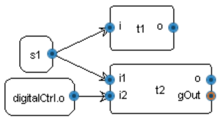
Figure 7. The mode transition editor



Property	Value
Identifier	main
Period	1
Time Unit	ms

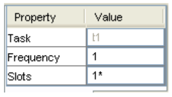
Figure 8. Adding a task invocation to a mode

```
task
 [freq=1] t1{i := s1;}
 [freq=1] t2{i1 := s1;i2 := digitalCtrl.o;}
```



Assignments are represented as arrows from the source port to the task's target input port. The source ports for such an assignment must either be sensors, task output ports, or global output ports. These source ports are also dragged from the tree pane and dropped in the editor pane. Only assignments between type compatible ports are allowed.

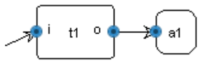
The logical execution time (LET) of a task invocation is determined by (1) the period *p* of the enclosing mode, and (2) by the frequency *f* of the invocation: *LET* = *p/f*. The frequency is a property that has to be set for every activity. It is specified in the property pane. Basically, the frequency divides the mode period into equal time slots. By default one task execution is performed in each slot. Using the *slots* property, the execution intervals can be specified more precisely: The execution in particular slots can be omitted, but consecutive slots can also be combined to one execution. Figure 9 shows the slot configuration of a task invocation with frequency 10, where the task is executed three times. The LET of the first execution is modePeriod/10, whereas the LET of the two following executions is three times longer.



Property	Value
Task	t1
Frequency	1
Slots	1*

Actuator updates. An actuator update sets the value of an actuator according to the specified assignment. To add an actuator update to the mode definition, the corresponding actuator is dragged from the tree pane and dropped onto the mode editor. Again, the source ports for an assignment must either be sensors, task output ports, or global output ports.

```
actuator
  [freq=1] a1 := t1.o;
```



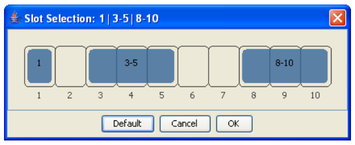
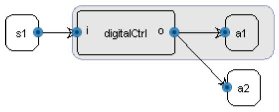
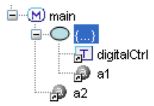


Figure 9. Slot selection

Task sequences. A task sequence combines a task invocation and subsequent actuator updates. As mentioned above, digital controller applications might require splitting the functionality of tasks into two functions, in order to provide output as fast as possible. The first step of the invocation (the execution of the *release* function) is executed in logical zero time, whereas the second function adheres to LET semantics. Actuator updates that are declared within a task sequence are updated after the first step of the invoked task. Actuators that depend on one of the task's output ports and that are not part of the sequence are updated after the second step.

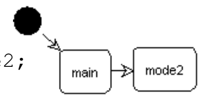


```
task [freq=1, slots=1*] {
 digitalCtrl {i := s1;};
    a1 := digitalCtrl.o;
}
actuator
 [freq=1, slots=1*] a2 := digitalCtrl.o;
```

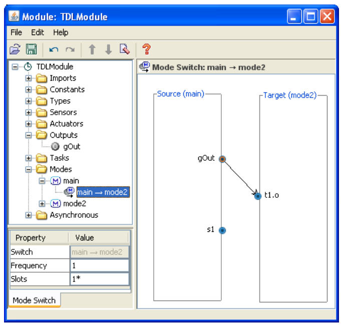


Mode switches. Typically, a TDL application declares multiple modes of operation. At run-time the application is able to change its current mode. This is called a mode switch. Mode switches are declared as activities of the source mode. The transition from one mode to the other is modeled using the mode transition editor (Figure 7).

```
start mode main [100 ms] {
  mode [freq=1, slots=1*] mode2;
}
```



This would cause the application to change its state at the end of the mode period. In order to perform mode switches conditionally, mode switches – but also any other activity – can be guarded (see below).



Property	Value
Switch	main → mode2
Frequency	1
Slots	1*

Figure 10. The mode switch editor

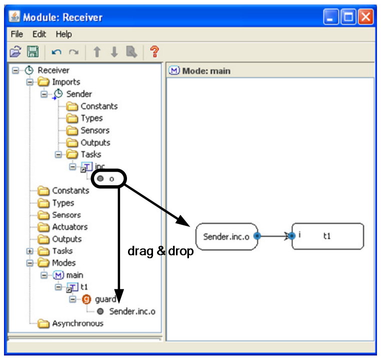


Figure 11. Assignments with imported constructs

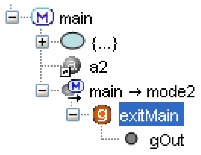
When a mode switch is performed, data calculated in the source mode can be transferred to the target mode as initial values of computations there. The data-flow from one mode to the other is expressed by mode switch assignments. Every output port of a task that is executed in the target mode is a possible candidate for an assignment.

```
start mode main [period=1 ms] {
  //...
  mode [freq=1] mode2 {t1.o := gOut;}
}
mode mode2 [period=1 ms] {
  task [freq=1] t1 {}
}
```

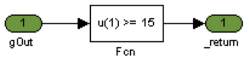
Mode switch assignments are visualized and modeled using the *mode switch editor* (see Figure 10), which again follows the data-flow paradigm.

Guards. Every mode activity can be guarded by an externally implemented function, called *guard*. A guard returns a boolean result and may take sensors, task output ports or global output ports as arguments. A guarded activity is only performed when its corresponding guard evaluates to true.

```
mode
[freq=1] if exitMain(gOut)
  then mode2;
```

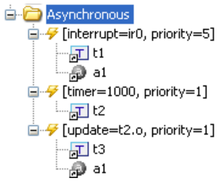


After creating a guard by a context menu entry of the mode switch activity, the guard function is shown as a subnode. Guards can be defined for task invocations, actuator updates, task sequences, and mode switches. Guard parameters can be added using drag & drop. As with task- and initializer-functions, guards are defined in Simulink. Every guard parameter is mapped to an Inport. Additionally, a dedicated Outport *(_return)* provides the return value:



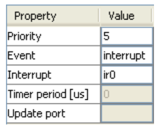
Asynchronous sequences.
An asynchronous sequence [13] is a group of asynchronous activities that are executed in consequence of a trigger event. Asynchronous activities are carried out in the spare time between the executions of timed (synchronous) activities. An asynchronous activity may be a task invocation or an actuator update.

```
asynchronous {
  [interrupt=ir0, priority=5]
    t1(s1); a1 := t1.o;
  [timer=1000, priority=1]
    t2(t1.o);
  [update=t2.o, priority=1]
    t3(t2.o); a1 := t3.o;
}
```



Asynchronous sequences have a property for the priority and the trigger event:

interrupt. requires the definition of an interrupt identifier (Interrupt).



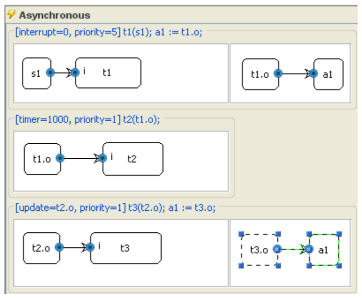
Property	Value
Priority	5
Event	interrupt
Interrupt	ir0
Timer period [us]	0
Update port	

timer. requires the definition of the period in microseconds (Timer period).

update. requires the definition of a port (Update port). Whenever the port is updated, the sequence is registered for execution.

Like mode activities, asynchronous activities may be guarded.

When selecting the *Asynchronous* node in the tree pane, the editor pane shows the *asynchronous editor*. This editor is used to add asynchronous activities and to define the data-flow between asynchronous sequences. Similar to the mode editor, ports and activities are added by drag- and drop operations from the tree pane.

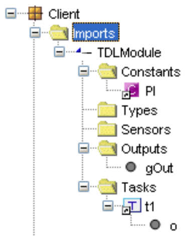

Asynchronous
[interrupt=0, priority=5] t1(s1); a1 := t1.o;
s1
i
t1
t1.o
a1
[timer=1000, priority=1] t2(t1.o);
t1.o
i
t2
[update=t2.o, priority=1] t3(t2.o); a1 := t3.o;
t2.o
i
t3
t3.o
a1

Imports. TDL modules can import from each other. Dependencies between modules are expressed by import declarations. A *client* module can import other modules *(service modules)*.

```
module Client {

  import TDLModule;

  // ...
}
```



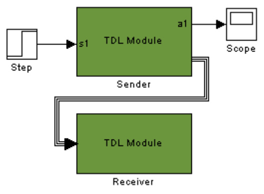
A module can import any other module that is located in the same Simulink model. Subsequently, a client module may access all TDL constructs which were declared as public in an imported module. Exportable constructs comprise constants, types, sensors, global output ports, and tasks (actually output ports of tasks).

Imported constants, for example, might be used to initialize local ports. Sensors and output ports might be used in assignments or as guard parameters. TDL modules are allowed to cyclically import each other. However, for all but mode declarations, the import relationship between modules must form a directed acyclic graph.

Figure 11 exemplifies the usage of imported constructs of the module Sender (Figure 3) in a second module Receiver:

```
module Receiver {
 import Sender;
 public task t0 {
  input int i;
  ...
 }
 start mode main [period=10ms] {
  task [freq=1] if g(Sender.inc.o)
    then t1 {i := Sender.inc.o;}
  }
}
```

Intrinsically, module dependencies stemming from import relationships are not visible in Simulink. However, in order to visualize also this TDL dependency, we indicate the data-flow between TDL modules by means of so-called Simulink buses.



5. 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. When the simulation shows satisfactory behavior, code generators like the Real-Time-Workshop Embedded Coder 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 application exhibits exactly the same behavior as during the simulation.

6. Related Work

The MathWorks [6] 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. Without extensions such as the TDL:VisualCreator tool, however, the inherent fusion of functional and temporal aspects does not allow modeling of systems that can efficiently be mapped to distributed platforms.

Visual modeling environments are also available for synchronous languages such as Lustre or Esterel, most notably SCADE from Esterel Technologies [8] and SyncCharts [9]. As the synchronous programming model assumes an immediate reaction to events (that is in logical zero time) it is also not suited for the development of distributed systems.

The Ptolemy [10] project is an ambitious approach that is not limited to a specific paradigm but allows modeling and simulating heterogeneous computation models. From the perspective of TDL, Ptolemy represents another attractive visual environment. The status of the integration of TDL in Ptolemy is currently 'work in progress'.

The SimTools [11] from Decomsys/Elektrobit and analogous tools such as TTP-Matlink from TTTech [12] 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.

7. Conclusion and Outlook

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. We outlined the core concepts of the Timing Definition Language (TDL), and presented a straight-forward mapping of textual TDL constructs to their visual representation. 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 described the integration into the widely used visual environment MATLAB/Simulink, which offers optimal synergy.

Both the visual representation and the Simulink integration have quite evolved over the last years and emerged to a stable tool which proved flexible enough to fit with latest TDL language extensions and ongoing MATLAB/Simulink releases. However, although the combined data-flow based representation of all mode activities within one diagram (mode editor) perfectly shows the relation and interplay of all involved TDL constructs, this representation also entails scalability shortcomings. An alternative separated representation of each activity would scale better, albeit hinder grasping the continuous data-flow from sensors to actuators.

References

[1] J. Templ. Timing Definition Language (TDL) 1.5 Specification. Technical Report, University of Salzburg, 2008. Available at www.softwareresearch.net

[2] FlexRay Web site. www.FlexRay.com

[3] Giotto Project. http://embedded.eecs.berkeley.edu/giotto

[4] T. Henzinger, B. Horowitz, and C. Kirsch. Giotto: A time-triggered language for embedded programming. In Proc. of EMSOFT, LNCS 2211, pages 166–184. Springer, 2001

[5] preeTEC. www.preeTEC.com

[6] The MathWorks. www.mathworks.com

[7] dSpace. www.dspace.com

[8] Esterel Technologies. www.esterel-technologies.com

[9] C. André, Computing SyncCharts Reactions, Electronic Notes in Theoretical Computer Science, 2004

[10] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, Y. Xiong, Taming Heterogeneity - The Ptolemy Approach, Proc. of the IEEE, v.91, 2003

[11] Decomsys/EB. www.decomsys.com/simtools

[12] TTTech. www.tttech.com

[13] J. Templ, J. Pletzer, A. Naderlinger. Extending TDL with Asynchronous Activities. Technical Report, University of Salzburg, 2008. Available at www.softwareresearch.net