Modeling with the Timing Definition Language (TDL)

Wolfgang Pree and Josef Templ

C. Doppler Laboratory Embedded Software Systems
University of Salzburg, 5020 Salzburg, Austria
{pree,templ}@SoftwareResearch.net
preeTEC GmbH,
Nico-Dostal-Str. 6, 5020 Salzburg, Austria
www.preeTEC.com

Abstract. This paper describes the model-based development process of hard real-time software with the Timing Definition Language (TDL): modeling and simulation of TDL components in Matlab®/Simulink®, their mapping to a specific platform and finally the code generation.

1 TDL Components

Model-based development requires appropriate domain abstractions. They allow developers to ignore nasty details in the process of modeling automotive software systems. On the one hand, the challenge is to find abstractions that are high-level so that as many details as possible can be ignored. On the other hand, these abstractions must not be too disconnected from the underlying system so that efficient code can be generated out of the models.

In case of general purpose programming, imperative languages turned out to represent an appropriate level of abstraction from the von-Neumann computer architecture. In case of hard real-time systems, such abstractions have to consider the timing behavior as well as concurrency and have only been proposed recently. Synchronous languages such as Esterel [1] assume that infinitely fast computers exist that can immediately react to sensor input. Though composition of Esterel software is straightforward in theory, it encounters barriers in practice, in particular for distributed systems [2]. Giotto [3, 4, 5] and the Timing Definition Language (TDL) [6] share the same basic programming model which relies on the Logical Execution Time (LET) [5] abstraction. LET means that the observable temporal behavior of a task is independent from its physical execution. It is only assumed that physical task execution is fast enough to fit somewhere within the logical start and end points. The following figure shows the relationship 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 as basis of portability and well-defined interaction semantics between parallel activities. It is always defined which value is in use at which time instant and there are no race conditions or priority inversions involved.

M. Broy, I.H. Krüger, and M. Meisinger (Eds.): ASWSD 2006, LNCS 4922, pp. 133–144, 2008.

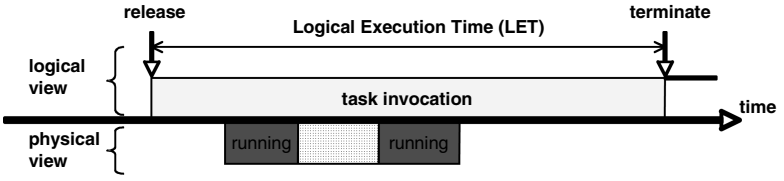

Fig. 1. Logical Execution Time (LET) abstraction

In addition to expressing LET semantics in a more convenient syntax than Giotto, and slightly simplifying the mode switching semantics, TDL has introduced the module (= component) as a top level language construct. This represents a significant step towards a component model for hard real-time systems. A TDL component provides a namespace for the definition of constants, types, sensors, actuators, tasks and modes. For the formal specification of the TDL constructs and their semantics we refer to the language report [6]. The TDL component construct serves multiple purposes:

1) it supports information hiding,
2) it acts as a static specification of components and dependencies,
3) it serves as the unit of distribution of functionality over a network of electronic control units.
4) it represents a partitioning of the set of actuators and control logic available in a system, and
5) it may serve as the unit of dynamic loading of system extensions

The TDL component construct is the precondition for another enhancement of LET-based languages. LET introduces a delay for observable outputs which poses a problem for controllers whose behavior would be better if outputs are provided as fast as possible without LET delays. With the TDL component construct it became possible to introduce globally asynchronous, locally synchronous (GALS) behavior. In this context 'globally' means between TDL components and 'locally' means within a TDL component. In order to avoid delays within a TDL component for the benefit of digital controller applications, a task's functionality code may be split in two parts: (a) a fast step and (b) 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 regularily. Output ports updated in the fast step are available immediately for a component's actuator updates or as inputs to other tasks within a TDL component.

Transparent Distribution. The TDL component in combination with the LET abstraction also forms the basis of what we call *transparent distribution*: Due to the LET semantics (1) the observable functional and temporal behavior of a system is the same no matter on which node of a distributed platform a TDL component is executed and (2) the developer does not have to care about the differences of local versus distributed execution of a TDL component. We refer to (1) and (2) as transparent distribution [7]. Transparent distribution facilitates, for example, what the automotive industry calls Electronic Control Unit (ECU) consolidation. The implementation of transparent distribution has required solutions of non-trivial communication scheduling problems as described in [10].

TDL is a textual language. We show the TDL code of a sample TDL component after Figure 8. In addition to the textual version we have developed a visual and interactive TDL editor tool, called the TDL:VisualCreator, that offers exactly the same constructs as the textual version of TDL. The user of the TDL:VisualCreator tool can view the corresponding textual version of the TDL component at any time. The TDL:VisualCreator and other TDL tools are available as products from preeTEC.com and have been built on the basis of research results in the realm of the Giotto project [3] at the University of California, Berkeley and the MoDECS project [8] at the University of Salzburg, Austria. The following section exemplifies TDL development with the TDL:VisualCreator tool. In section 3 we illustrate how TDL components are automatically mapped to a sample FlexRay platform, illustrating the benefits of transparent distribution. TDL with its component model and the transparent distribution of TDL components cut the overall development time of FlexRay software by a factor of 20 compared to tools that require a manual or slightly automated specification of communication schedules and that do not abstract from the platform.

2 Modeling Sample TDL Components

A case study for controlling an Active Rear Steering (ARS) system illustrates the advantages of the straight-forward modeling process with TDL. The ARS system is courtesy of MagnaSteyr Fahrzeugtechnik [9].

A TDL module (= component) corresponds to a control application that periodically reads sensor values, calculates output values and writes these to actuators. Thus, a TDL module consists of a set of sensors, a set of actuators and a set of modes. Each mode consists of a set of periodic task invocations and other periodic activities such as actuator updates or mode switch checks. A module can be in one mode at a time.

In the ARS case study we use the TDL:VisualCreator tool that also allows TDL modeling within Matlab®/Simulink®. This has the advantage, that TDL components can be simulated. Note that the developer specifies the TDL modules and their timing behavior, that is the LET of the tasks, independent of a specific execution platform.

To edit a TDL module we drag the TDL module block from the Simulink® Library Browser (see Figure 2) and drop it on to a model.

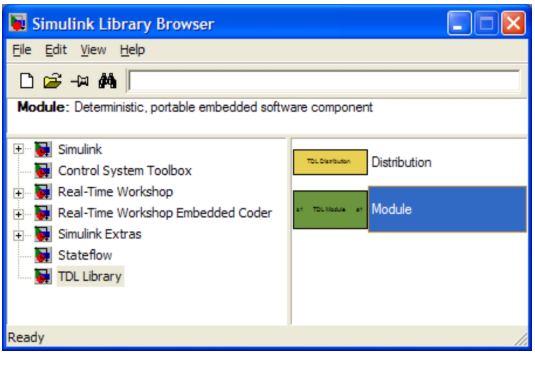

Fig. 2. TDL module as block in Matlab®/Simulink®

A double-click on the module block opens the TDL:VisualCreator where you can edit the various elements of a module (see Figure 3). The tree on the left hand lists the possible TDL constructs: the imported modules (see below) in folder Imports, the constants, types, sensors, actuators, the task declarations in folder Tasks, and the modes of the TDL module. Figure 3 shows that we have defined three sensors (delta_r_act, angular_rate, current) and one actuator (voltage) of the TDL module RearActuatorController. The developer edits the properties of a TDL construct by clicking on it. Figure 3 shows the properties of the selected sensor *current*. The corresponding properties and the corresponding values are displayed below the tree.

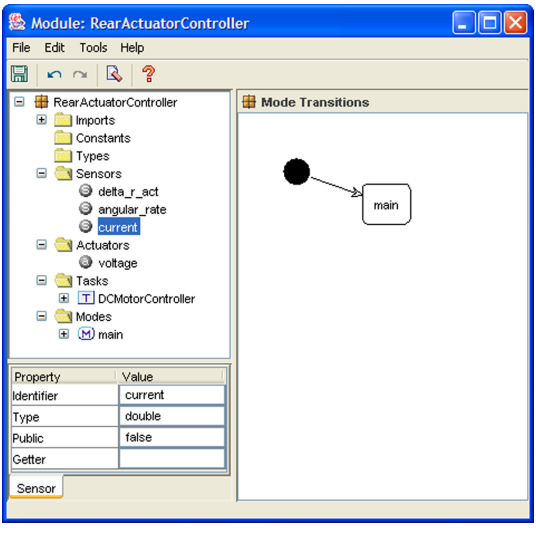

Fig. 3. Editing a TDL module in the TDL:VisualCreator

The TDL module RearActuatorController has only one task DCMotorController (see also Figure 3). In this simplified case study the mode main is the only mode of operation in this TDL module.

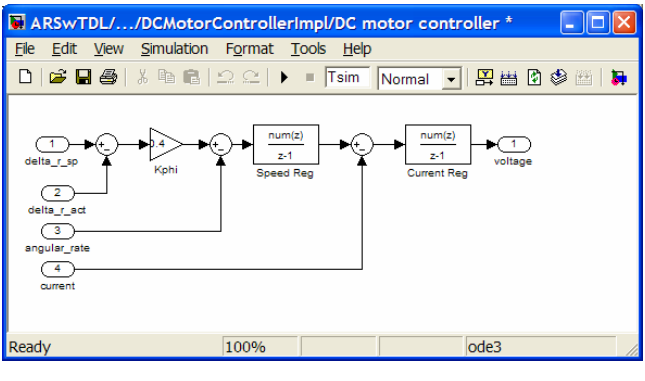

Fig. 4. Modeling a task's functionality with Matlab®/Simulink®

Let us now illustrate how we define the functionality of task DCMotorController and its timing behavior, that is, its LET. A double-click on the task opens a Simulink editor. The developer can use any of Matlab®/Simulink®'s discrete blocks to model the controller behavior. Figure 4 exemplifies this for the DCMotorController.

In the next step we define the timing behavior of the task DCMotorController in mode main and how it gets its inputs and to where it provides its output. For that purpose we click on mode main in the tree. Now we can define the period of mode main as one of its properties. In this example we set it to 1ms (see Figure 5). In the data flow editor, shown on the right-hand side of the window, we define the input and the output connections of task DCMotorController.

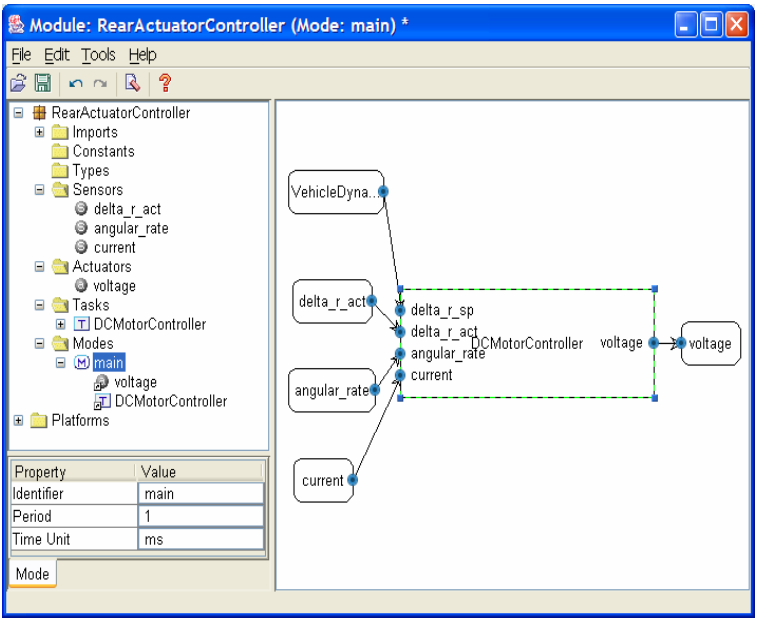

Fig. 5. Defining the timing of a mode

Finally, we define the LET of task DCMotorController within mode main. This is done by specifying the invocation frequency in relation to the mode period (see Figure 6). As the Frequency property is set to 1 it means that the LET of task DCMotorController is 1ms (mode period) divided by 1 (frequency), thus 1 ms.

Figure 7 shows the overall Matlab®/Simulink® model of the ARS system, consisting of the two TDL modules RearActuatorController and VehicleDynamics and a subsystem Vehicle that represents the 'plant', that is, the relevant aspects of the vehicle that needs to be controlled. What we did not show was the definition of the import relationship between the two modules. Module RearActuatorController imports VehicleDynamics and uses the output port delat_r_sp of VehicleDynamics' public task dynamicsController. TDL module imports are discussed below.

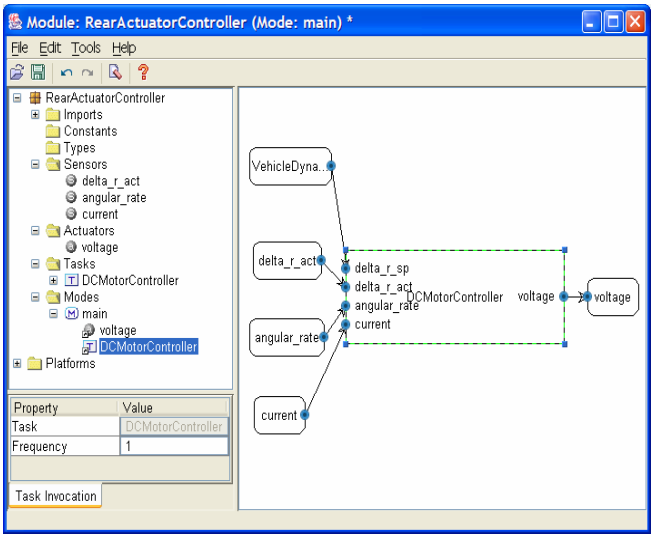

Fig. 6. Defining the LET of a task

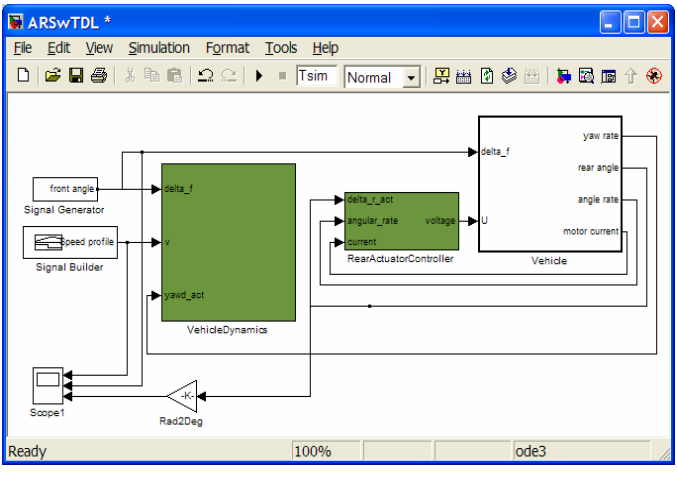

Fig. 7. ARS system model

The output of the Scope block (see Figure 8: speed, front angle, rear angle) illustrates the controller behavior: at low speeds steering operations cause the wheels on the rear axis to point in the opposite direction of the front wheels, whereas at higher speeds the wheels on the front axis and the rear axis point in the same direction.

The following listing shows the textual version of the TDL module RearActuatorController. As stated above this module imports the other TDL module VehicleDynamics. The sensor, actuator, task and mode declarations correspond exactly to the

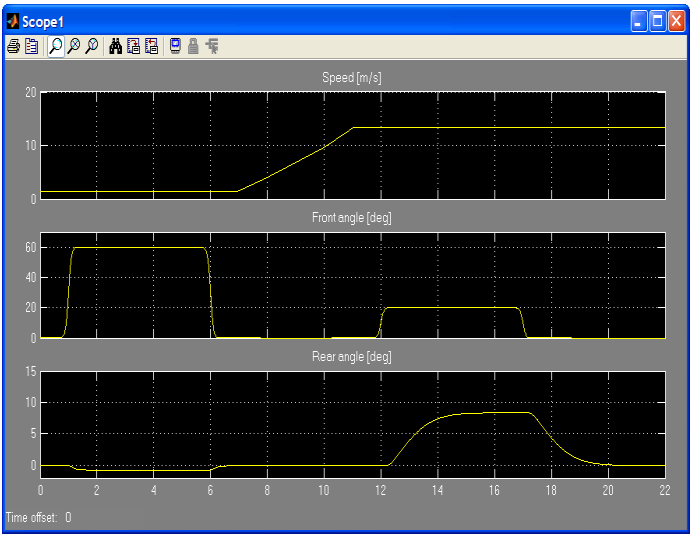

Fig. 8. ARS scenario

definition in the TDL:VisualCreator described above. The *uses* keyword marks an external function that implements platform-specific behavior. For example, the sensor delta_r_act is read by means of the external function getDelta_r_act. The implementation of sensor reading and actuator writing depends on the hardware and is thus separated from the platform-independent TDL code.

```
module RearActuatorController {

  import VehicleDynamics;

  public sensor double delta_r_act uses getDelta_r_act;
  sensor double angular_rate uses getAngular_rate;
  sensor double current uses getCurrent;

  actuator double voltage uses setVoltage;

  public task DCMotorController {  // task declaration
    input
      double delta_r_sp;
      double delta_r_act;
      double angular_rate;
      double current;
    output
      double voltage;
    uses DCMotorControllerImpl(delta_r_sp,delta_r_act,
                               angular_rate,current,voltage);
  }

  start mode main [period=1 ms] {
    task
      [freq=1] DCMotorController { // task invocation with LET = 1 ms
         delta_r_sp := VehicleDynamics.dynamicsController.delta_r_sp;
         delta_r_act := delta_r_act;
```

```
            angular_rate := angular_rate;current := current;
          }
      actuator
        [freq=1] voltage := DCMotorController.voltage;
    }

  }
```

Module Import. In order to allow the decomposition of large applications into smaller parts and to allow expressing dependencies between modules statically, the module concept provides an import mechanism, which allows a client module to specify that it depends on a service module and to access public elements of the imported module. In the ARS case study module RearActuatorController imports VehicleDynamics. Thus module RearActuatorController can use any of the exported items, that is, those that have the property Public set to true.

While it is obvious that using imported constants, types and sensors does not pose any semantic difficulties, it is not a priori clear how to treat constructs such as tasks and actuators. Multiple applications may read the same sensors, for example, but what happens if multiple applications write to the same actuators? Note that any of the parallel running TDL modules may be in one of several modes and it is not statically defined which actuators are under control of which application at which time. Therefore it must be prevented that multiple modules write to the same actuator. The module construct comes in handy to solve this problem. We simply restrict actuator updates to the module the actuator is declared in. Thus, the module construct also acts as a partitioning of the set of actuators. In a large application, sensors could be declared in a common service module, from where they can be used in any client module. A client module declares a subset of the actuators of the complete system and provides the functionality and timing to set their values.

Tasks form the units of computation. They are invoked periodically with a specified frequency. They deliver results through task output ports to actuators or to other tasks, and they read input values from sensor ports or from output ports of other tasks. A task whose visibility property Public is set to true exports all of its output ports. Thus, client modules can access the results delivered through a task's output ports, but it is not possible to invoke tasks from client modules.

Separation of Concerns. A TDL module expresses only the timing behavior with LET semantics: when tasks read inputs and when they provide outputs, when mode switch conditions are checked and when actuators are updated. The functionality is separated and specified as functions external to TDL: that is, how sensors are read, how actuators are updated, how tasks process their inputs. These external functions can be implemented in any programming language. In case of using TDL within Matlab®/Simulink® as illustrated in the realm of the TDL modeling by means of the TDL:VisualCreator above, the task functionality can be specified with Simulink® blocks (see Figure 5). For these Simulink® subsystems the developer generates C code with one of the available code generation tools so that the system can be mapped to a specific execution platform. Currently, TDL supports language bindings for ANSI C and Java.

We view this separation of timing and functionality as a precondition of an appropriate component model, in particular in the automotive industry. It allows the

protection of intellectual property rights of the supplier companies. The supplier companies still can implement the specific control laws and provide that functionality as object code. On the other hand, the Original Equipment Manufacturers (OEMs) can integrate the components from various different suppliers based on the TDL component model—they do not have to know about the implementation of the functionality. We will exemplify one aspect of the integration process, the TDL module-to-node mapping, in more detail in the next section.

3 Sample TDL Component Deployment on a FlexRay Cluster

The TDL component model offers another separation of concerns that we have called transparent distribution: the behavior of a component is independent of the execution platform. With TDL the platform can also be considered *after* a component has been developed. This is in stark contrast to the current development practice which produces software that is strongly intertwined with the platform it was developed for. The good news for the developer is that the mapping to a specific platform, no matter whether it is distributed or not, becomes a straight-forward assignment of TDL modules to nodes (ECUs). preeTEC's automatic schedule and code generators and the TDL run-time system guarantee that the executable code exhibits exactly the same timing behavior as in the simulation, provided that the target platform offers sufficient computing resources. If not, no code is generated and the developer gets hints why this was not possible.

In order to map a set of TDL modules to a specific platform, the user puts a Distribution block from the Simulink® Library Browser to the particular model by dragging it from the library and dropping it onto the model (see Figure 9). In our

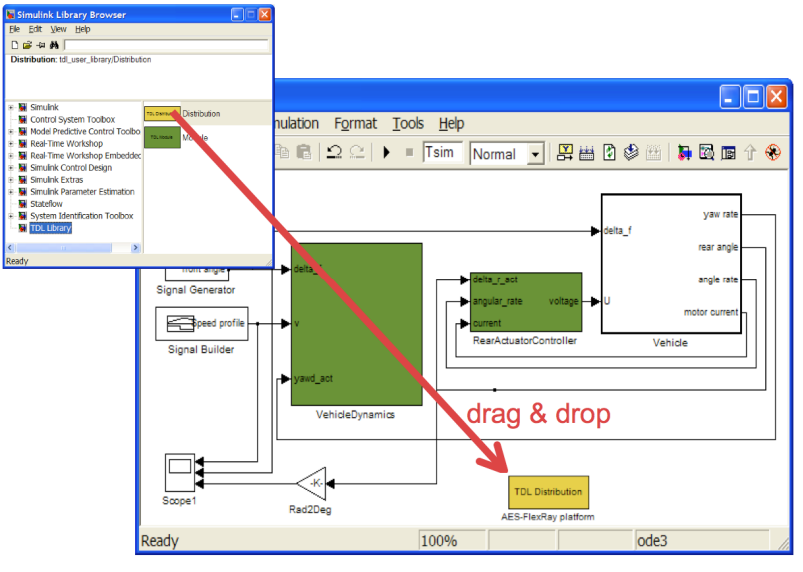

Fig. 9. Adding a Distribution block to the ARS system model

example, we name the block *AES-FlexRay platform* as we want to map the two TDL modules to a FlexRay cluster [11] with the socalled AES operating system [12]. Note that any number of platform mappings could be defined for a model, simply by putting further Distribution blocks to a model with TDL modules.

A double-click on the TDL Distribution block opens the TDL:VisualDistributor tool (see Figure 10). It already contains the TDL modules RearActuatorController and VehicleDynamics defined in the ARS model where the Distribution block was inserted.

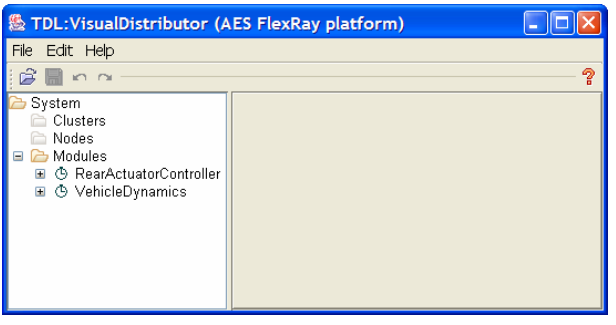

Fig. 10. TDL:VisualDistributor tool with the TDL modules defined in the Simulink® model

In order to assign the TDL modules to the nodes of a platform we now have to define the platform. The TDL:VisualDistributor offers the editing features to define the topology and properties of potentially distributed platforms that are common in the automotive domain. For a demo video that illustrates how to define a FlexRay platform we refer to the Web [13]. The TDL:VisualDistributor can also save and load platforms. In this case study we assume that a platform that describes a FlexRay cluster with two MPC5554 nodes as ECUs, each running the AES operating system, has already been defined. We load that platform and can then assign the two TDL modules by means of a straight-forward drag & drop operation to the two nodes ECU1 and ECU2 of that particular FlexRay cluster. Figure 11 shows the resulting view of the specified module-to-node assignment.

The TDL:VisualDistributor accomplishes the automatic generation of all files that are required to build the executable(s) for the specific platform. In case of the FlexRay-AES platform we need, for example, the platform-independent TDL source code for the modules RearActuatorController and VehicleDynamics, the C code (generated, for example, with the Real-Time-Workshop Embedded Coder) for each task function of each TDL module, the FIBEX file representing the communication schedule, the FlexRay-specific configurations and the makefiles. For that purpose we simply choose the Build All menu item in the File menu of the TDL:VisualDistributor tool.

After compiling the code and uploading it to each of the nodes the system behaves as simulated in the TDL:VisualCreator. As TDL modeling means basically setting the LET periods of tasks, and as the user does not have to define a communication schedule and the numerous FlexRay details, TDL together with the automatic generators reduces the development time by a quantitatively measured factor of 20 and more compared to state-of-the-art methods and tools, if a FlexRay-system is developed from scratch.

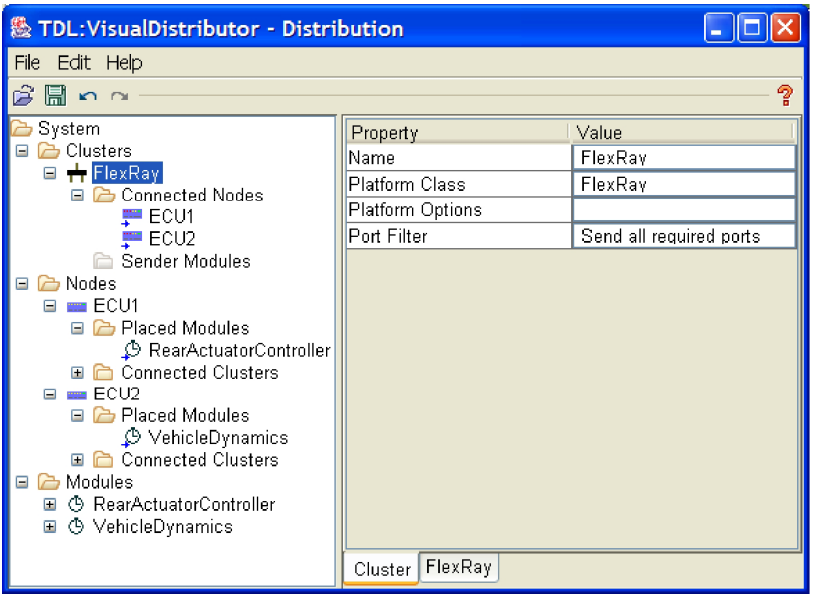

Fig. 11. TDL module-to-node assignment for a FlexRay cluster

4 Opportunities for the Automotive Industry Resulting from a TDL-Based Development Process

Besides significant development and maintenance cost savings, for example, for FlexRay software, a TDL-based development process could provide the desired flexibility for automakers to change the execution platform when required and at the same time redefine the OEM-supplier relationship.

Remember that the TDL components can be modeled and simulated without knowing on which platform they will be executed. One key benefit of the TDL-based development process is that an Original Equipment Manufacturer (OEM) would not have to worry from the beginning on which node of a distributed platform a TDL component will be executed.

Let us assume a sample scenario of how the automotive industry could harness the TDL technology: An OEM and a supplier agree about the coarse-grained system structure by means of TDL components. Each TDL component corresponds to a system function, such as automatically maintaining the distance to other vehicles. The OEM and its suppliers only have to refine each system so far as required by TDL, that is, the definition of the timing. The suppliers then implement the functionality of the TDL components, that is, typically the control laws, and error handling as well as accomplish the component validation and testing.

Parallel to this activity the OEM selects the computing and communication platform. For example, the OEM wants to reduce the number of ECUs (Electronic Control Units) and use more powerful computing nodes instead. The OEM can finetune the platform configuration with a tool such as the TDL:VisualDistributor.

Once the supplier companies return the TDL components together with the implemented functionality, the integration has already been accomplished and the system testing effort is significantly reduced—the TDL tools and middleware guarantee the time-safe execution of all TDL components.

Instead of specifying the platform in parallel to the development activities of the supplier companies, the OEM might prefer to define, in the traditional way, the platform upfront and give away the TDL components to the suppliers afterwards. The OEM would still benefit from the TDL approach: besides the guaranteed time and value determinism, future changes of the platform only require an automatic regeneration of the code and the communication schedules. This improves an OEM's flexibility, for example, to reduce the number of ECUs or to upgrade the hardware.

References

1. Esterel Technologies Inc., http://www.esterel-technologies.com
2. Kirsch, C.M.: Principles of Real-Time Programming. In: EMSOFT 2002. Grenoble LNCS, vol. 2491 (2002)
3. Giotto Project, http://embedded.eecs.berkeley.edu/giotto/
4. Henzinger, T.A., Horowitz, B., Kirsch, C.M.: Embedded control systems development with Giotto. In: Proceedings of the International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), ACM Press, New York (2001)
5. Henzinger, T.A., Kirsch, C.M., Sanvido, M.A.A., Pree, W.: From control models to real-time code using Giotto. IEEE Control Systems Magazine 23(1), 50–64 (2003)
6. Templ, J.: Timing Dfinition Language (TDL) 1.2 Specification. Technical Report, on the Web -> Technology -> Further Documents (2007), at http://www.preeTEC.com/
7. Farcas, E., Farcas, C., Pree, W., Templ, J.: Transparent Distribution of Real-Time Components Based on Logical Execution Time. In: Proc. of ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), pp. 31–39. ACM Press, New York (2005)
8. MoDECS (Model-Based Development of distributed, Embedded Control Systems) Project, http://www.MoDECS.cc/
9. MagnaSteyr Fahrzeugtechnik, a global brand-independent engineering and manufacturing partner of automakers, http://www.MagnaSteyr.com/
10. Farcas, E., Pree, W., Templ, J.: Bus Scheduling for TDL Components, Dagstuhl conference on Architecting Systems with Trustworthy Components. LNCS, Springer, Heidelberg (2006)
11. FlexRay Web site, http://www.FlexRay.com/
12. DeComSys/Elektrobit Web site, http://www.decomsys.com/
13. preeTEC Web site, http://www.preeTEC.com/