Real-Time Component Integration Based on Transparent Distribution

Emilia Farcas, Claudiu Farcas, Wolfgang Pree and Josef Templ
Department of Computer Science, University of Salzburg, Austria
firstname.lastname@cs.uni-salzburg.at

ABSTRACT

This paper introduces a real-time component model that offers a separation of concerns which allows a straight-forward integration of independently developed components. So-called transparent distribution forms the backbone of the integration process. Transparent distribution means that (1) the functional and temporal behavior of a system is the same no matter on which node of a distributed system a component is executed and (2) the developer does not have to care about the differences of local versus distributed execution of a component. We first present the concepts of a component model for real time systems that is well suited for transparent distribution. The component model is based on logical execution time, which abstracts from physical execution time and thereby from both the execution platform and the communication topology. Then we discuss the resulting tool chain and integration process. A case study rounds out the paper.

1. INTRODUCTION

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, comprises up to 80 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. Dealing with time, however, is not covered appropriately by any of the existing component models for high-level languages.

A particularly promising approach towards a high-level component model for real time systems has been laid out in the Giotto project [5][12][13][14] by introduction of logical execution time (LET), which abstracts from the physical execution time on a particular platform and thereby abstracts from both the underlying execution platform and the communication topology. Thus, it becomes possible to change the underlying platform and even to distribute components between different nodes without affecting the overall system behavior. Giotto, however, is primarily an abstract mathematical concept and there exist only simple prototype implementations, which show some of the potential of LET.

This paper presents a component model, named TDL (Timing Definition Language) [7], that has been developed in the course of the MoDECS[1] project at the University of Salzburg, as a successor of Giotto. It shares with Giotto the basic idea of LET but introduces additional high-level concepts for structuring large real time systems.

In the following, we start with an explanation of LET and proceed with an overview of the TDL component model and its associated notion of transparent distribution. Then, we sketch the envisioned integration process of TDL components. The integration of two sample components rounds out the paper.

2. LOGICAL EXECUTION TIME (LET)

LET means that the observable temporal behavior of a task is independent from its physical execution [12]. It is only assumed that physical task execution is fast enough to fit somewhere within the logical start and end points. **Figure 1** shows the relation between logical and physical task execution.

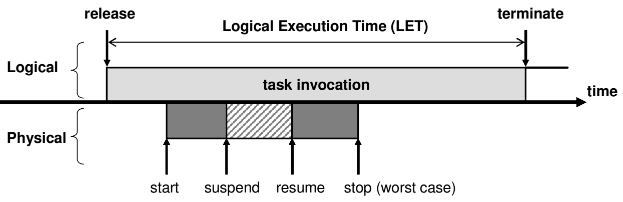

Figure 1 – Logical Execution Time

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 introduces a delay for observable outputs, which might be considered a disadvantage. On the other hand, however, LET provides the cornerstone to deterministic behavior, platform abstraction and well-defined interaction semantics between parallel activities [2]. It is always defined which value is in use at which instant and there are no race conditions or priority inversions involved. As we will see later, LET also provides the foundation for transparent distribution.

[1] The MoDECS project (www.MoDECS.cc) is supported by the FIT-IT Embedded Systems grant 807144 (www.fit-it.at).

Based on the concept of LET, Giotto introduces the notion of a *mode* as a set of periodically executed activities. The activities are task invocations (with LET semantics), actuator updates, or mode switches. All activities can have their own rate of execution and all activities can be executed conditionally. Actuator updates and mode switches are considered to be much faster than task invocations, thus they are executed in logical zero time. The set of all modes reachable from a distinguished start mode constitutes the Giotto *program*.

For the reader who knows Giotto in more detail, the following list summarizes the differences between the original and the successor (TDL) that we have created as a platform for real-time component development and integration.

- TDL defines a concrete syntax and a binary e-code file format, whereas Giotto is primarily a mathematical abstraction of real-time programming based on LET.
- TDL provides a named top-level program entity (module) and a component model based on the notion of modules. We shall go into more details of the component model in the subsequent chapters.
- TDL considers modules as units of distribution, whereas Giotto envisions to distribute individual tasks.
- TDL replaces global output ports by task output ports.
- TDL eliminates mode ports and replaces them by optional assignments to task ports upon a mode switch.
- TDL eliminates explicit task, update, and mode drivers and merges them into mode declarations.
- TDL adds named constants, which may be used to initialize ports.
- TDL introduces units for timing values and uses microseconds internally instead of milliseconds.
- TDL defines program start as mode switch to the start mode of a module without any special treatment for program start.
- TDL disallows non-harmonic mode switches, i.e. mode switches must not occur during the logical execution of a task.
- TDL introduces the new e-code instruction SWITCH, which is used for performing mode switches.
- TDL defines deterministic mode switches as the switch to the first mode in textual order whose guard evaluates to true rather then specifying that only one guard is allowed to return true.

3. TDL COMPONENT MODEL

Our successor of Giotto, named TDL (Timing Definition Language), extends the concepts introduced in Giotto by the notion of the *module*, which is a named Giotto program that may import other modules and may export some of its own program entities to other client modules. Every module may provide its own distinguished start mode. Thus, all modules execute in parallel or in other words, a TDL application can be seen as the parallel composition of a set of TDL modules. It is important to note that LET is always preserved, i.e. adding a new module will never affect the observable temporal behavior of other modules. It is the responsibility of internal scheduling mechanisms to guarantee conformance to LET, given that the worst-case execution times (wcet) and the execution rates are known for all tasks. **Figure 2** sketches a sample module with two modes containing two cooperating tasks each.

Parallel tasks within a mode may depend on each other, i.e. the output of one task may be used as the input of another task. All tasks are logically executed in sync and the dataflow semantics is defined by LET.

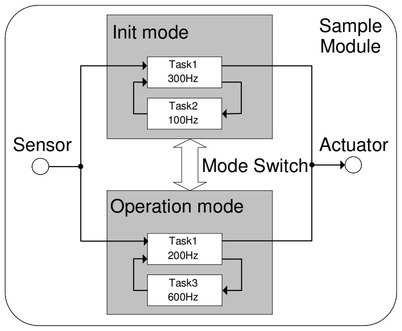

Figure 2 – Visual representation of a TDL module

Modules support an export/import mechanism similar to modern general purpose programming languages such as Java or C#. A service provider module may export a task's outputs, which in turn may be imported by a client module and used as inputs for the client's computations. All modules are logically executed in sync and again the dataflow semantics is defined by LET. Modules are a top-level structuring concept that serves multiple purposes: (1) a module provides a name space and an export/import mechanism and thereby supports decomposition of large systems, (2) modules provide parallel composition of real time applications, (3) modules serve as units of loading, i.e. a runtime system may support dynamic loading and unloading of modules, and (4) modules are the natural choice as unit of distribution because dataflow within a module (cohesion) will most probably be much larger than dataflow across module boundaries (adhesion). The possibility to distribute TDL modules across different computation nodes leads us to the notion of transparent distribution as discussed below. In the following we exemplify the syntax of a TDL module. The TDL language report [7] describes the language syntax and semantics in detail.

TDL module. The module construct starts with the keword module followed by the name of the module and a pair of curly brackets, which represent the namespace introduced by the module. The following example shows the skeleton of a module.

```
module EngineControl {
   // TDL code consisting of sensor, actuator, task,
   // and mode declarations
}
```

In order to provide for globally unique module names by prefixing with revers internet domain names (similar to Java) module names are allowed to contain '.'.

CPU partitioning. A module may provide a *start* mode, which is the mode the application is executing after loading the module into an Electronic Control Unit (ECU). Executing a module implies the reservation of a percentage of the available CPU time for execution of this module, given that the CPU is fast enough to execute this module in addition to possibly other modules loaded before. A module which needs to reserve a percentage of the CPU is called a 'partition' and splitting the CPU between multiple partitions is called 'CPU partitioning'. A module which does not provide a start mode will not be executed, which means, it will not need a CPU partition but it may still be meaningful for example to export common constants or types.

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 is dependent from a service module and to access public elements of the imported module. The import relationship forms a directed acyclic graph (DAG) between client and service modules.

```
module AdvancedCar{
  import EngineControl;
  import BrakeByWire;
  import ...;
  // sensor, actuator, task and mode declarations;
  // may access public elements of imported modules
}
```

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, modes 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 applications 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 applications write to the same actuator. The module construct comes in handy to solve this problem. We simply restrict actuator update 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.

Information hiding. According to popular programming languages we use the keyword 'public' to mark program elements as being publicly visible. There is no need (so far) for a corresponding keyword 'private', as this is the default anyway and there is no further level of visibility.

```
module EngineControl {
  public const maxRpm = 6500;
  //... more code
}
```

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. Currently, TDL supports language bindings for ANSI C and Java.

We view this separation of timing and functionality as a precondition of a component model in the automotive industry. It allows the protection of intellectual property rights of the supplier companies. The supplier companies still can implement the particular 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 shall discuss the integration process in more detail in section 5.

The TDL component model offers another separation of concerns: the behavior of a component is independent of the execution platform. The platform is considered *after* a component has been developed. This is in stark contrast to current development practice which produces software that is strongly intertwined with the platform it was developed for. The following sections discuss the advantages of what we call transparent distribution in the realm of component integration.

4. TRANSPARENT DISTRIBUTION

We define the term *transparent distribution* in the context of hard real-time applications with respect to two aspects. Firstly, at runtime a TDL application behaves exactly the same, no matter if all modules (i.e. components) are executed on a single node or if they are distributed across multiple nodes. The logical timing is always preserved, only the physical timing, which is not observable from the outside, may be changed. Secondly, for the developer of a TDL module, it does not matter where the module itself and any imported modules are executed. The TDL tool chain and runtime system frees the developer from the burden of explicitly specifying the communication requirements of modules. It should be noted that in both aspects transparency applies not only to the functional but also to the temporal behavior of an application.

The advantage of transparent distribution for a developer is that the TDL modules can be specified without having the execution on a potentially distributed platform in mind. The only place where distribution is visible is for the system integrator, who must specify the module-to-node assignment (see section 5).

Figure 3 shows an example of a set of four TDL modules distributed across three nodes.

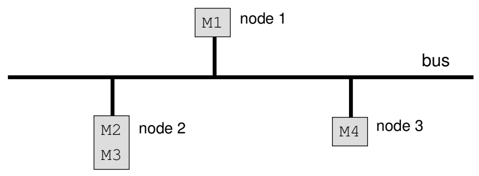

Figure 3 – Example of distributed modules

In order to illustrate transparent distribution of TDL modules, we start with a subset of **Figure 3**. Let us consider modules M1 and M2, which are located on two different nodes. For the sake of simplicity, we assume that each module has a single mode of operation, which invokes a single task. task1 runs within module M1 and task2 runs within module M2 using as input the output of task1. In this case, following the TDL semantics, module M2 has to import module M1, and task2 must have as input the output port of task1. The arrow between the two tasks from the modules M1 and M2 in **Figure 4** expresses this relationship.

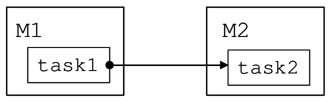

Figure 4 – Communication between two modules on separate nodes

For this example, we further assume that task2 runs twice as often as task1, i.e. the LET of task1 is twice as large as the LET of task2. Remember that the LET concept specifies that no matter when the task runs within its LET, the task gets its inputs at the beginning of LET and provides its outputs to other tasks or actuators only at the end of its LET. As a first step, Figure 5 shows a sample execution of the two tasks on a *single* node .

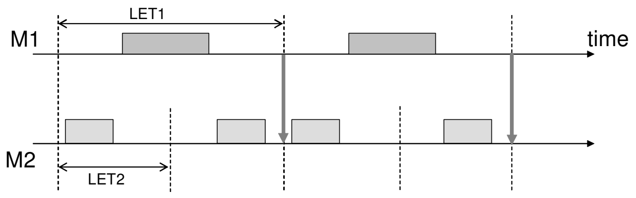

Figure 5 – Single node execution of two tasks

After task1 finishes its physical execution, the TDL run-time system buffers its output internally and provides it to task2 at the end of LET1. task2 reads its input at the beginning of the LET2, but the TDL run-time system schedules it for execution later. According to LET semantics, the first instance of task1 communicates its outputs to the third instance of task2 at end of LET1, as the vertical arrow indicates.

Copying values from one location of memory to another takes close to zero time on a single node. In a distributed setting, however, there is a delay because communication takes much longer and only one node can send at a time. **Figure 6** shows a sample communication pattern between the two tasks on *different* nodes. In order to implement this exchange of information between the two tasks, we need to add an auxiliary communication layer on both nodes that we call TDLComm. Its purpose is to send and receive messages at the *right* times.

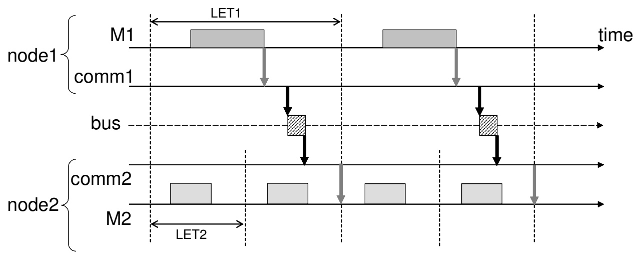

Figure 6 – Sample communication between two tasks

In order to be able to guarantee the timing of messages, we use a TDMA (Time Division Multiple Access) [6] approach. This means that any node is allowed to send messages in statically defined slots only. Furthermore, we implement the Producer-Consumer (i.e., Push) model. This means that the tasks that generate information, the producers, trigger the sending of a message. The consumers do not send any requests to the producers, as for example in the Client-Server model. In the previous example, the Push model avoids to resend messages without any value being changed.

The bus schedule generation tool automatically determines the communication pattern for a given set of modules and network properties. The resulting bus schedule is a statically defined table that specifies which node sends which package at which time. The table defines all network activities within one communication period (also named bus period), which is the least common multiple of all activity periods involved.

In order to achieve our goal of transparent distribution, after task1 finishes, the system copies the internal output value to the TDLComm layer on node1 (comm1) that buffers it. Afterwards, comm1 sends the value in a packet at the time specified in the bus schedule while the TDLComm layer from node2 (comm2) has to receive the packet and buffer it. We assume that network operations are executed by a dedicated network controller in parallel with task execution, which is the case in most systems. On node2, at the LET-end instant of task1, when the value should logically arrive, the system provides the value from the TDLComm layer. Clients of task1, such as task2, then use this value without making any difference between importing it locally or remotely. For a detailed description of the middleware implementation and the schedule generation we refer to [3].

5. TDL TOOL CHAIN AND INTEGRATION PROCESS

This section provides an overview of the core TDL tool chain and its implications for integrating components. **Figure 7** shows the tool chain as well as which inputs the tools require and which outputs they produce.

The compiler processes TDL source code and generates an abstract syntax tree (AST) representation of the TDL program as intermediate format as well as the so-called embedded code (e-code) [11], which describes when to release a task. The plug-in architecture of the compiler allows the extension of the tool with any number of tools that rely on the AST.

We also provide a VisualTDL editor that is seamlessly integrated in Simulink [4]. Thus a developer can visually and interactively model a TDL module and its functionality in Simulink, simulate it and once it fulfills the requirements generate the TDL source code for the timing behavior and C source code for the functionality.

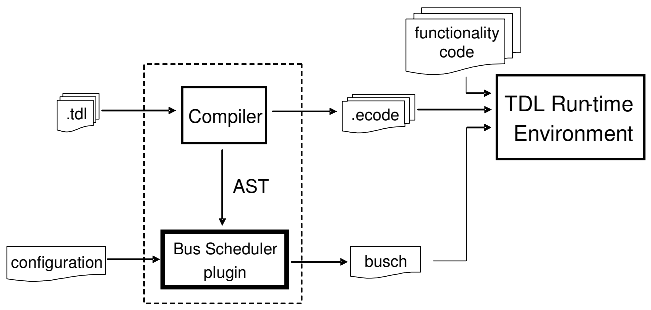

Figure 7 –TDL Tool Chain

The bus scheduler is a plug-in tool that generates the bus schedule, based on a configuration file. The configuration file simply contains a list of computing nodes that comprise the particular platform, the assignment of TDL modules to computing nodes, and the physical properties of the communication infrastructure.

The runtime environment of TDL is structured in several layers and is based on virtual machines. Tasks are executed according to the LET semantics under the control of the E-Machine[11]: a virtual machine that executes E-code instructions. Scheduling decisions generated by the bus scheduler can be executed by the OS scheduler or better by the S-Machine [10]: a virtual machine that executes scheduling-code (S-code) instructions.

TDL component integration process. A TDL module is supposed to encapsulate the functionality that is typically put on one ECU in current automotive system designs, such as the all-wheel-drive control system or the engine control system. **Figure 8** shows what we call the V-Cluster-Life-Cycle: Modules are developed independently of each other, potentially by different suppliers. Each V-Life-Cycle delivers a module. Each module can be tested by its supplier, typically independent of the other modules that are integrated into one system later. The supplier only needs other modules for testing if his particular module imports other ones.

The TDL component integration that takes place at an OEM would work as follows: The system integrator specifies the module-to-node assignment by means of a configuration file. The compiler generates the e-code and the bus scheduler generates a static description of the network activities according to the import relationships and the communication infrastructure. Due to transparent distribution the behavior (timing and functionality) of the modules is unchanged no matter how they are distributed on a specific platform.

It might happen that the platform resources are not sufficient to execute all modules that should be integrated into one system. In this case the compiler tool set does not generate code. In order to avoid such a situation at a late stage of system development, an OEM might want to specify the TDL modules in advance. In this case, the separation of timing and functionality greatly supports the integration process: The TDL modules can be specified without much effort in advance, probably in interaction with the component suppliers: they describe only the timing behavior, not the functionality. The functionality is then provided by the suppliers later.

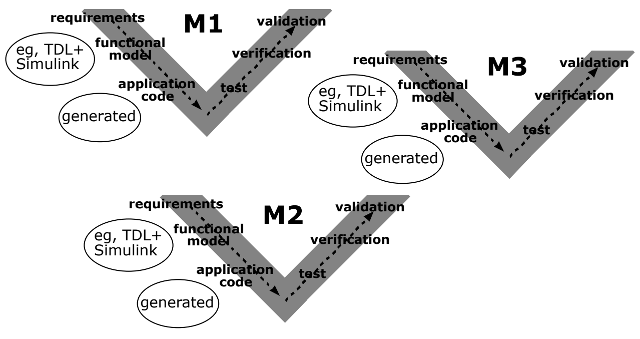

Figure 8 – V-Cluster-Life-Cycle

Though each module has to be tested, verified and validated, the advantage of the TDL component model is that the integration of the modules into one system, which causes significant costs today, comes for free through automatic code generation and guarantees that the behavior of each component is unchanged. The LET abstraction forms the basis that the 'glue code' can be generated automatically. The code generation process could be formally verified.

6. CASE STUDY

We illustrate transparent distribution by means of two modules with simple functionality. **Figure 9** shows the modules with their modes, tasks, and ports. Module M1 has one sensor input, two tasks called inc and dec, and two actuators connected to the output ports of the tasks. The inc task increments its output value by 10, starting with the initial value 50 up to the upper limit 200. The dec task decrements its output value by 10, starting with the initial value 200 down to the lower limit 50. The sensor is only used for switching between the two modes of the module. In mode f11 both tasks have the same LET, namely 10 ms. In mode f12 the task dec has a LET of 5 ms—it produces the output values twice as fast as task inc.

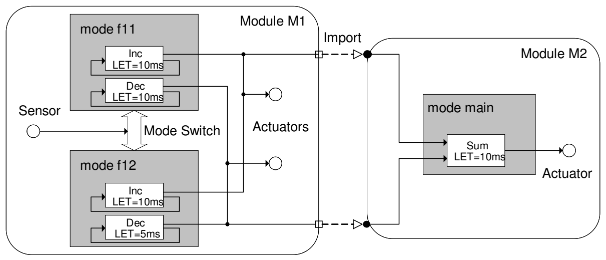

Figure 9 – Module M2 imports module M1

Module M2 imports module M1 and thus has access to the output ports of M1's tasks inc and dec. Module M2's task sum simply adds the outputs of M1's inc and dec tasks. The LET of task sum is 10 ms.

As a developer specifies only the timing behavior in TDL, the functionality of the tasks has to be implemented in another programming language. In this case study the functions invoked by the tasks and the drivers for reading sensors and updating actuators have been implemented in C as external functionality code. The TDL source code shown below indicates this by the keyword *uses*.

```
module M1 {

 public const
   c1 = 50; c2 = 200; refPeriod = 10ms;

 sensor int s uses getS;

 actuator
   int a1 := c1 uses setA1;
   int a2 := c2 uses setA2;

 public task inc [wcet=1ms] {
   output int o := c1;
   uses incImpl(o);     // inc. by step 10
 }
 public task dec [wcet=1ms] {
   output int o := c2;
   uses decImpl(o);    // dec. by step 10
 }
 start mode f11 [period=refPeriod] {
   task
     [freq=1] inc();    // LET of task inc is 10/1 = 10 ms
     [freq=1] dec();
   actuator
     [freq=1] a1 := inc.o;    // actuator a1 update every 10 ms
     [freq=1] a2 := dec.o;
   mode
     [freq=1] if switch2m2(s, inc.o) then f12;
 }
 mode f12 [period=refPeriod] {
   task
     [freq=1] inc();
     [freq=2] dec();   // LET of task dec is 10/2 = 5 ms
   actuator
     [freq=1] a1 := inc.o;
     [freq=2] a2 := dec.o;
   mode
     [freq=1] if switch2m1(s, inc.o) then f11;
 }
}

module M2 {
 import M1;

 actuator int a := M1.c2 uses setA;

 public task sum [wcet=1ms] {
   input int i1; int i2;
   output int o := M1.c2;
   uses sumImpl(i1, i2, o);
 }
 start mode main [period=M1.refPeriod] {
   task
     [freq=1] sum(M1.inc.o, M1.dec.o);
   actuator
     [freq=1] a := sum.o;
 }
}
```

Execution on two different computing platforms. We want to execute the two modules 1) on a single node platform and 2) on a distributed platform with two computing nodes. The single node platform consists of a Kanis Evaluation Board [8] based on the MPC555 processor, with 4MB RAM and the OSEKWorks [9] real-time operating system. For the distributed, two-node platform we add an identical board. The two boards communicate via the CAN bus [1]. We implemented a simple time-triggered protocol on top of CAN to avoid collisions. A simple push button serves as sensor for module M1. The actuators are four channels 8-bit DAC connected to each board. We connect the probes of a digital oscilloscope to the output channels of the DAC in order to visualize the output signals generated by the sample application. Due to transparent distribution, both the functional and temporal behavior of the modules have to be exactly the same no matter where the modules are executed. Remember that distribution is only visible for the system integrator, who must specify the module-to-node assignment by means of a configuration file.

If both modules should be executed on the single node platform, no configuration file has to be provided at all. The developer simply compiles each module. The compiler produces the TDL-specific output, in particular the E-Code. An OSEK-specific TDL compiler-plugin generates the so-called OIL file, which is required to execute the modules on the OSEK operating system. After compiling the functionality code with the DIAB C compiler the executables are uploaded and ready to run. **Figure 10** shows the outputs of module M1's inc and dec tasks and module M2's sum task. Module M1 is in mode f11 in the beginning while the sum task is producing a constant output. After pushing the sensor button, a mode switch occurs and task sum produces the corresponding output pattern. The delay between the output of the sum task and the output of the inc and dec tasks is due to the LET semantics.

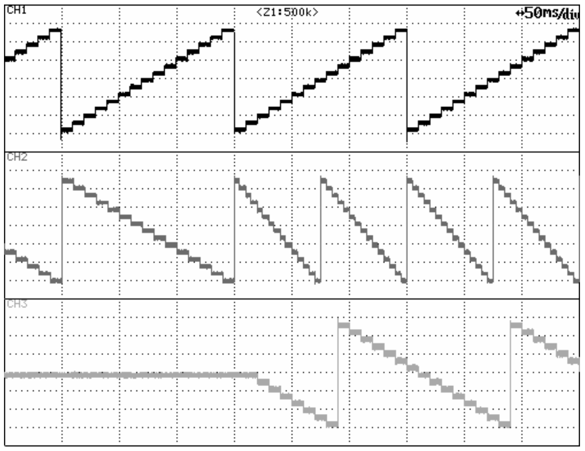

Figure 10 – Functional and temporal behavior of modules M1 and M2 (mode f11 and then f22)

In order to run the modules on two different nodes, we have to specify to which node a module should be assigned. The configuration file simply contains a list of computing nodes that comprise the particular platform and the assignment of TDL modules to computing nodes. The syntax of the configuration file adheres to the syntax of Java property files, which represent properties as key-value pairs. Indexed properties are used to express lists. For example, the assignment of module M1 to node1 and module M2 to node2 is specified as follows:

```
tdl.bus.nodes = 2
tdl.bus.nodes.0 = node1
tdl.bus.nodes.1 = node2
tdl.bus.modules = 2
tdl.bus.modules.0 = M1:node1
tdl.bus.modules.1 = M2:node2
```

The configuration file, which is one input to the bus schedule generator, contains further information about the communication system. For example, in case of the CAN bus with our simple time-triggered protocol, the configuration file specifies the following properties: envelope bits, gap bits, bus rate in Hz, minimum packet size, max packet size, and the clock resolution.

We now compile the two modules again, providing also the configuration file as input. As a result the corresponding compiler plugins produce in addition to the outputs obtained in the single-node case the stub module for Node2 and a separate Makefile and OIL file for each node. After recompiling the application using the two Makefiles we get two executables, one for each board. Each board now runs a TDL run-time environment that comprises TDLComm. Remember that the access to the shared communication medium is collision free via a TDMA approach. In order to support this we rely on a mechanism for clock synchronization over the network. In the set-up of our case study, this is not available a priori. Thus, we had to implement it in software: for this purpose, the TDLComm layer generates synchronization frames with timestamps for all other nodes that might be connected to the CAN bus. The TDLComm layer on each node uses the synchronization frames from the bus to synchronize the local OSEK clock to the remote clock.

After uploading both modules the functional and temporal behavior of modules M1 and M2 is exactly the same as when both modules are executed on one node. After connecting one oscilloscope probe to the appropriate DAC channel of the second board, the oscilloscope patterns are again the ones shown in **Figure 10**.

The TDL runtime environment composed of E-Machine, TDLComm, and some system dependent functions that handle the platform clock and task management, has a very low memory footprint, in our case around 68KB (or 24KB without OSEK debugging information).

7. CONCLUSIONS

The LET abstraction invented in the realm of the Giotto project paved the way for a lean component model for real-time systems that offers transparent distribution. The resulting tool chain and integration process could significantly reduce the costs of integration and system testing. Future research and implementation efforts are required to show the scalability of transparent distribution and the usability of the integration process in practice. It also needs to be investigated which other attributes besides timing and functionality have to be considered. An example would be the memory requirements of a component. Another challenge is the dynamic loading and unloading of modules which would lead to an extension of the run-time system and probably would require enhancements of the component model.

8. ACKNOWLEDGEMENTS

We thank the MoDECS project team at the University of Salzburg for providing valuable input during informal discussions and group meetings. This research was supported in part by the FIT-IT Embedded Systems grant 807144 provided by the Austrian government through its Bundesminsterium für Verkehr, Innovation und Technologie.

9. REFERENCES

[1] Bosch, 1991, *CAN Specification, Version 2*. Robert Bosch GmbH, http://www.can.bosch.com/docu/can2spec.pdf

[2] C.M. Kirsch, 2002, *Principles of Real-Time Programming.* In Proceedings of EMSOFT 2002,Grenoble LNCS, 2491.

[3] Emilia Coste, Claudiu Farcas, Wolfgang Pree and Josef Templ. *Transparent Distribution of TDL modules*. Technical Report, University of Salzburg, Austria, 2005.

[4] Gerald Stieglbauer and Wolfgang Pree. *Visual and Interactive Development of Hard Real Time Code.* Automotive Software Workshop San Diego (ASWSD 2004)

[5] Giotto Project, http://www-cad.eecs.berkeley.edu/~fresco/giotto/

[6] H. Kopetz, 1997, *Real-time Systems: Design Principles for Distributed Embedded Applications*. Kluwer, 1997.

[7] J. Templ, 2004, *TDL Specification and Report.* Technical Report C059, Department of Computer Science, University of Salzburg, http://www.cs.uni-salzburg.at/pubs/reports/T001.pdf

[8] OAK_EMUF Dev. Board, Ing. Buero W. Kanis GmbH http://www.kanis.de/home/products/oak_emuf/i_oak.htm

[9] OSEK Group, 2001, *OSEK/VDX Time-triggered Operating System Specification, Version 1.0*, http://www.osek-vdx.org/mirror/ttos10.pdf

[10] T.A. Henzinger, C.M. Kirsch, and S. Matic. *Schedule carrying code*. In Proc. of the Third International Conference on Embedded Software (EMSOFT), LNCS, Springer-Verlag, 2003.

[11] T.A. Henzinger and C.M. Kirsch, 2002, *The Embedded Machine: predictable, portable real-time code*. In Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pages 315–326.

[12] Thomas A. Henzinger, Benjamin Horowitz, and Christoph M. Kirsch. *Giotto: A time-triggered language for embedded programming.* Proceedings of the First International Workshop on Embedded Software (EMSOFT), Lecture Notes in Computer Science 2211, Springer-Verlag, 2001, pp. 166-184.

[13] Thomas A. Henzinger, Benjamin Horowitz, and Christoph M. Kirsch. *Embedded control systems development with Giotto.* Proceedings of the International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), ACM Press, 2001, pp. 64-72.

[14] Thomas A. Henzinger, Christoph M. Kirsch, Marco A.A. Sanvido, and Wolfgang Pree. *From control models to real-time code using Giotto*. IEEE Control Systems Magazine 23(1):50-64, 2003.