Transparent Distribution of Real-Time Components Based on Logical Execution Time

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 the notion of transparent distribution of real time software components. Transparent distribution means that (1) the functional and temporal behavior of a system is the same no matter where a component is executed, (2) the developer does not have to care about the differences of local versus distributed execution of components, and (3) the components can be developed independently. We present the design and implementation 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.

Categories and subject descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming - Distributed programming; J.7 [*Computers in other systems*]: Industrial control; Process control; Real time.

General Terms Design, Languages, Reliability

Keywords LET, MoDECS, TDL, Timing Definition Language, component model, distribution, embedded systems, logical execution time, real-time, transparent.

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 about 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.

LCTES'05, June 15–17, 2005, Chicago, Illinois, USA.

A particularly promising approach towards a high-level component model for real time systems has been laid out in the Giotto project [6][21][22][23] 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) [9], 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 shall start with an explanation of LET and proceed with an overview of the TDL component model. Then, we introduce the notion of transparent distribution and present the design and implementation of the TDL tool chain. A simple example and a section on related work round out the paper.

2. Logical Execution Time (LET)
LET means that the observable temporal behavior of a task is independent from its physical execution [21]. 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.

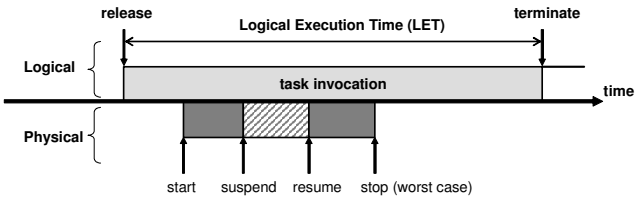

Figure 1 – Logical Execution Time

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.

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

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 [3]. It is always defined which value is in use at which time instant and there are no race conditions or priority inversions involved. As we will see later, LET also provides the foundation for transparent distribution.

3. TDL Component Model

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 (according to 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*.

Our successor of Giotto, named TDL (Timing Definition Language), extends these concepts 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.

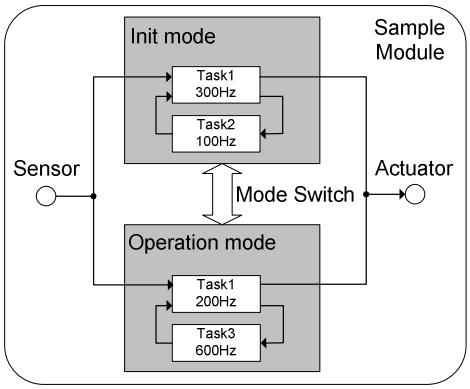

Figure 2 – Visual representation of a TDL module

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.

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 input 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 explained below.

4. Transparent Distribution

We define the term *transparent distribution* in the context of hard real-time applications with respect to two points of view. 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 mapping of modules to computation nodes is defined separately. Nevertheless, the functional and temporal behavior of a system is exactly the same no matter where a component is executed.

The only place where distribution is visible is for the system integrator, who must specify the module-to-node assignment by means of a configuration file. This file, which is based on the available processing nodes with their peripherals (e.g., directly connected sensors and actuators) and network resources, is used as input for the TDL tool chain. **Figure 3** shows an example of a set of four TDL modules distributed across three nodes.

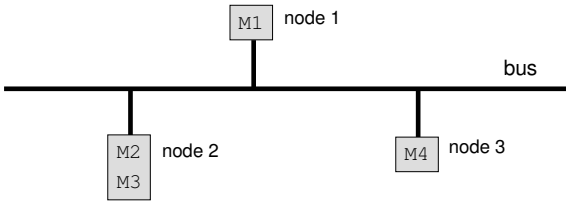

Figure 3 – Example of distributed modules

It should be noted that Giotto considers individual tasks as the units of distribution. TDL, however, uses the higher-level construct of a module, which encapsulates a set of tasks, as the unit of distribution and thereby deviates from Giotto.

5. TDL Tool Chain

Before we present the implementation of transparent distribution, we provide an overview of the core TDL tool chain. **Figure 4** 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) [20], 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.

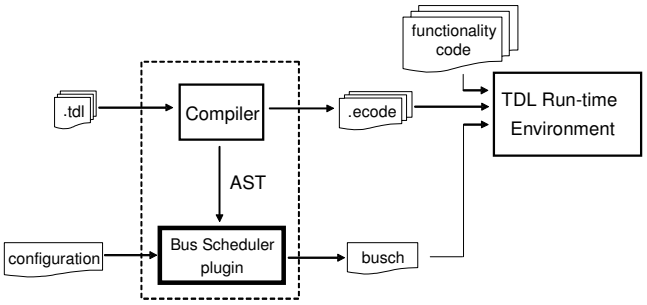

Figure 4 –TDL Tool Chain Core

The bus scheduler is such 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 [20]: a virtual machine that executes E-code instructions. Scheduling decisions can be executed by the OS scheduler or better by the S-machine [19]: a virtual machine that executes scheduling-code (S-code) instructions.

6. Implementation of Transparent Distribution

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 5** expresses this relationship.

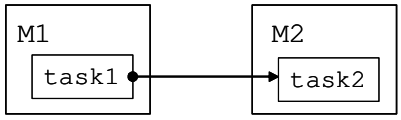

Figure 5 – Communication between two modules on separate nodes

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 6 shows a sample execution of the two tasks on a *single* node .

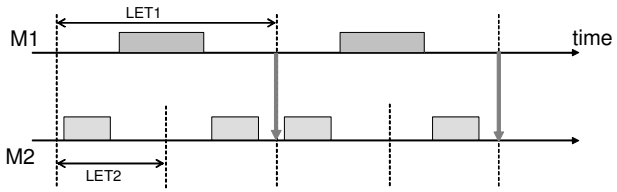

Figure 6 – Sample execution of two tasks

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 the 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 7** 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.

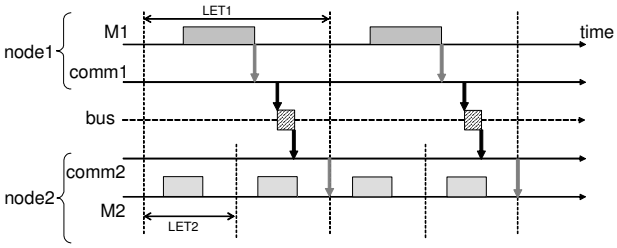

Figure 7 – Sample communication between two tasks

In order to be able to guarantee the timing of messages, we use a TDMA (Time Division Multiple Access) [7] 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 (see section 7) determines automatically 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. A feasible schedule for the network activity guarantees the deterministic behavior of the application in the distributed setup.

When the tool cannot find such a feasible schedule that maintains the properties of the single node system, the TDL compiler does not compile the application. It notifies the system integrator that the current distribution of modules is not appropriate (e.g. due to strong coupling between modules or insufficient network bandwidth).

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 on the TDLComm layer. Clients of task1, such as task2, then use this value without making any difference between importing it locally or remotely.

Optimizations

When the consumer has a lower frequency than the producer, an obvious optimization is to suppress messages that are not used at all. For example, if task1 had twice the frequency of task2, the communication period would be equal to the LET of task2, and we can skip sending the message produced by the first instance of task1 within the communication period, as task2 does not use it. This optimization saves network bandwidth.

For non-harmonic periods of the producer and the consumer, we can actually delay the message until the release time of a consumer. We can send the values even after the LET of the producer, but they still have to arrive before they are needed by the consumers.

Figure 8 presents such a scenario.

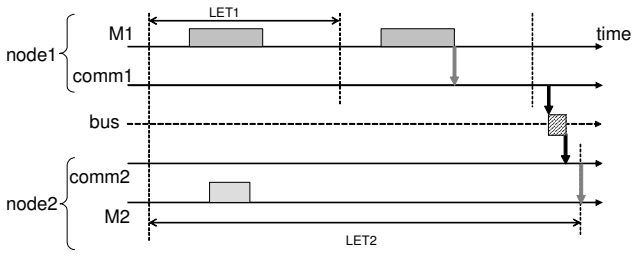

Figure 8 – Tasks with non-harmonic periods

This optimization not only provides more freedom for the bus scheduler but also for the task scheduler. On node1, the task scheduler now has more time available for scheduling the execution of task1.

Module communication modes

Figure 3 has presented a possible distribution of a set of four modules across a network consisting of three nodes. Now let us assume the following import relationships: M2 imports M1 and M4 imports M1 and M3, which results in the import graph shown in **Figure 9**.

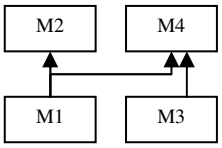

Figure 9 – Sample import graph

Since M1 is running on node1 but used at node2, we introduce a *stub* module for M1 on node2. This stub module abstracts the differences of local versus remote execution and provides the same interface as a locally executed module to clients such as M2 on node2. The details of communication with the remotely imported module via the comm layer are covered in the stub implementation, which is generated automatically.

If there are no remote clients of a module, as it is the case for M2 and M4 in our example, there is no need to actually send any messages on the bus. Such modules are said to work in *local* mode, and modules that have remote clients operate in *push* mode.

A module's communication mode is orthogonal to a module's operation mode, i.e. a module can operate in one of the TDL modes specified for the module and communicate in one of the three communication modes. In the current implementation state, there are some limitations for mode switches of remote modules, but we expect to lift these limitations in the future.

Figure 10 shows the stub modules that we create, and the modes of communication for all modules, in our example.

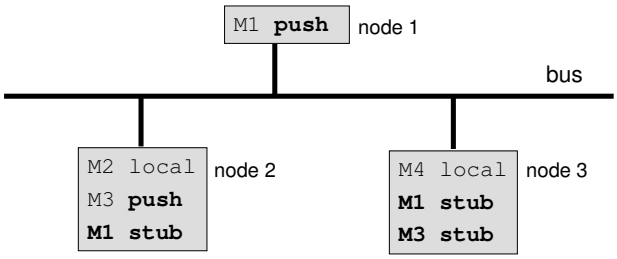

Figure 10 – Module communication modes

The implementation of the various communication modes is based on the driver concept as originally introduced by Giotto and generalized for TDL as outlined below.

Driver architecture

Figure 1 shows the principle of logical execution time and at the same time the main events that occur when executing a TDL task. *Release*, *Start*, *Stop*, and *Terminate* constitute events that trigger the execution of a so-called driver, which is a procedure that performs some action on behalf of the TDL run-time system. A Release driver, for example, copies the input parameters of a task, a Start driver calls the task implementation function, which is not

part of the TDL run-time system but part of the functionality code provided by the developer. Thus, the drivers are the glue code between the TDL run-time system and the environment. It may be worth to note that in Giotto drivers had to be specified explicitly by the developer whereas in TDL drivers are generated automatically by the tool chain. This leads to considerably shorter and more readable program texts in TDL.

The drivers also play the central role for implementing the communication modes of a module. The TDL run-time system is essentially not aware of any communication modes. It is the preparation and selection of drivers that realizes the variants as shown in Table 1.

Table 1 – Driver usage

	Local	*push*	*stub*
Release	Copy inputs	copy inputs	--
Start	execute task function	execute task function	--
Stop	--	copy results to TDLcomm	--
Terminate	copy outputs from task	copy outputs from task	copy outputs from TDLcomm

7. Bus Schedule Generation Tool

This tool, which is a core part of the TDL tool chain, generates the bus schedule, which is a static description of the network activities within a single communication period that is executed repeatedly. It uses as input a platform configuration file that specifies the assignment of modules to nodes and describes the physical properties of the network such as the bandwidth, the protocol overhead and the payload size.

Currently we use a broadcast model for communicating over the network. This is not a limitation as the tool could generate message-scheduling decisions for point-to-point communication as well, but broadcasting is the dominating model for existing embedded systems. 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. If this is not available a priori, it must be implemented in software as done for example in our case study (see section 8).

As mentioned earlier, the cooperation model used is the Producer-Consumer model (i.e., Push model). This means that the nodes that generate information – the producers – trigger transactions on the bus. The nodes that need the information – the consumers – retrieve it from the bus. The consumers do not send any requests to the producers, as for example in the Client-Server model.

The tool generates a bus schedule that contains the information regarding which node has to send which packet at which time. Each packet contains one datagram that comprises one or more messages that components have to exchange. From the TDL modules and module to node assignment, the tool automatically detects who has to communicate with whom, and then which messages are needed within a communication period.

In order to identify all messages a component must send on the bus, the tool scans the TDL modules and checks for all activities if the source and destination ports are on different nodes. The result is a set of messages, and for each message, the tool identifies the producer, the consumer(s), and the size of the data being sent. Producers can only be sensors and task output ports. Consumers can be actuators, task input ports and guard arguments. Guards allow carrying out activities conditionally.

As the communication pattern repeats after a bus period, the result is a finite set of message instances, with individual timing constraints: a Release Offset and a Deadline. The Release Offset is the worst case execution time of the producer, which is the earliest time we could send the message. The tool computes the message deadlines in different ways, depending on the optimizations performed. As a straight forward approach to implementing the Producer-Consumer model, we can send messages with the frequency of the producer, thereby ignoring knowledge about the consumers. In this case, the deadline of a message is the end of the LET of the producer and the tool computes the bus period as the LCM (Least Common Multiple) of the logical execution times of the producers. As an optional optimization, the tool additionally considers which messages are actually used by consumers. Therefore, the bus period is now equal to the LCM of the periods of the producers and the consumers, which may result in larger bus schedules. For each consumer instance, we find the last instance of the producer, because the consumer needs only this message (see the case when the consumer runs slower and we avoid redundant messages). If more consumer instances need the same producer instance, then in order to save bandwidth we send the message only once, with the deadline as the earliest release of these consumers.

Currently we generate the message and task schedules in two steps. We first schedule the messages and then we schedule the tasks with deadline constraints from the bus schedule: the producer tasks have to finish before the corresponding message is sent on the bus. A one step approach would be to schedule the messages and the tasks together, but this becomes complicated because we have multiple modules on the same node, and each one could perform independent mode switches.

Because the bus schedule is a static schedule, our strategy is to schedule messages as late as possible, which allows more flexibility for the task scheduler. For task scheduling we use the Earliest Deadline First (EDF) with precedence constraints algorithm [5]. For message scheduling, we use an heuristic algorithm, adapted from Reversed EDF, also called in the literature Latest Release Time (LRT) [11] The main idea is to treat deadlines as release times and vice versa. Therefore, we sort the list of messages by hierarchical keys: the message deadline, release time, and then the producer deadline. The bus scheduler is non-preemptive and just schedules the messages in that order, starting from the end of the bus period and going backwards. The bus scheduler merges messages if they are sent by the same node, and are adjacent. Furthermore, the bus scheduler has additional constraints which result from the physical properties of the communication infrastructure. For example, it includes gaps in the schedule, because it has to align the sending time according to the inter frame gaps and the clock resolution on the nodes. The bus scheduler generates also control messages, such as messages for time synchronization.

8. Case Study

We illustrate transparent distribution by means of two modules with simple functionality.

Figure 11 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.

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.

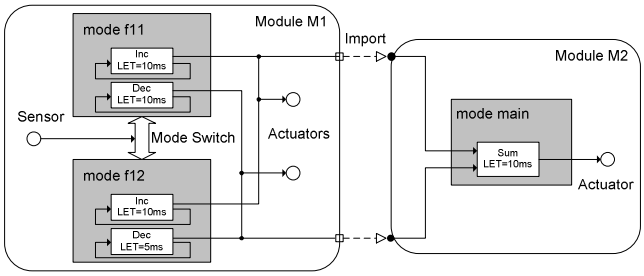

Figure 11 – Module M2 imports module M1

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 f11.period/inc.freq = 10ms
     [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 = 5ms
   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 [14] based on the MPC555 processor, with 4MB RAM and the OSEKWorks [15] real-time operating system. For the distributed, two-node platform we add an identical board. The two boards communicate via the CAN bus [2]. 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 12 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.

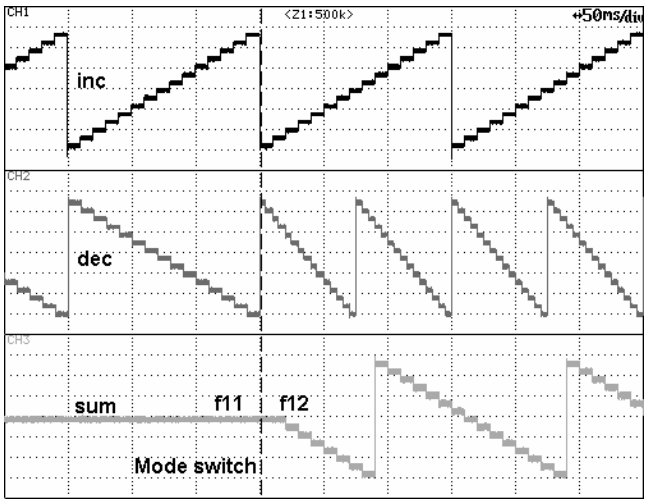

Figure 12 – Functional and temporal behavior of modules M1 and M2 (mode f11 and then f22)

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 12**.

9. Related Work

In general, state-of-the-art methods and tools for distributed system development do not abstract from the distributed platform. Furthermore, only a few tools support the development of distributed software which has to meet hard real-time constraints. For example, the Java [10] and .NET [13] approach is to simplify distributed programming: access to local and remote objects should be as similar as possible. But the remote access will take a longer time than a local access. As the timing behavior is not specified explicity, these approaches are not suitable for real-time applications.

Below we compare in particular TDL with tools that focus on the real-time domain. We do not take ACE/TAO [16] - a real-time extension of CORBA dating back to the mid of the 1990s - into consideration, as we regard its overhead, in particular in terms of memory footprint, as prohibitive.

Giotto. The Giotto language focuses on task distribution. The developer has to annotate the Giotto source code with network specific parameters such as the hostname and ports. This mixes platform-independent and platform-dependent code. The actual implementation of the communication has to be coded manually with so-called scheduling-code (s-code) instructions. In other words, no tools automatically generate the message schedules for the bus communication. Though Giotto's LET forms the basis for transparent distribution, its prototype implementations do not provide this feature.

TTTech Tools. TTTech is a Vienna-based company [24] which sells hardware and development tools for time-triggered system development. Both rely on the time-triggered protocol (TTP) [7] that features membership services, time triggered transmission of messages and distributed clock synchronization. As opposed to TDL, a developer has to consider in advance the target platform. Besides the fact that the computing platform topology, that is the number of nodes, is then hard-wired in the code, the TTTech hardware itself has to be used. Based on the computing platform topology the communication is specified with the TTPplan tool, which generates the message schedule. The components executed on the nodes can be developed independently using TTPbuild as

long as the communication requirements are unchanged. Additional messages require in most cases a change of the bus schedule. This represents a major obstacle to provide reusable components. Another limitation of TTTech's tools is that currently they only support a single application mode.

DaVinci and SysDesign tool suites. The DaVinci [25][12] and the SysDesign [4] tools are further examples of state-of-the-art software development tools that do not abstract from the distributed platform. Compared to TDL the developer still has to perform the activities in a platform-centric manner. Both tools support the developer by providing means for simulating the behavior of the control systems on single or distributed platforms. The execution of the components depends on the priorities set for the task functions. The tools do not restrict the usage of semaphores. That might lead to priority inversions or race conditions. The resulting software might run into deadlocks and is not deterministic. DaVinci and SysDesign could form a synergy with the TDL tools by providing a test environment for the execution of TDL modules. The developer could check whether the expected behavior and the behavior on a specific platform are identical.

Honeywell's MetaH [17] is a specification language for real-time distributed avionics software. It describes how different elements of a system such as software components, hardware, and communication subsystems are integrated to form the final application. MetaH provides periodic tasks and multiple modes. MetaH's intertask communication semantics is similar to TDL. Unlike MetaH the LET abstraction does not constrain the implementation to a particular scheduling paradigm. Moreover, the undelayed communication in MetaH is limited to local nodes only, that is the tasks that have dependencies must run on the same processor. According to [1]: "... MetaH is very low level. Today, it is rather a language for assembling existing pieces of code". A suite of visual tools help the MetaH developer to add components, edit them, define the scheduling, partition the application, and analyze the timing behavior.

10. Conclusions

The LET abstraction invented in the realm of the Giotto project paved the way for transparent distribution in real-time systems. Based on the implementation of transparent distribution described in this paper we are convinced that this novel approach is a breakthrough that will lead to significantly more robust embedded software and will also reduce the costs of integration testing. The first case studies and experiments have corroborated that transparent distribution is feasible. Future research and implementation efforts are required to show the scalability of transparent distribution. Another set of challenges comprises the generic and portable implementation of the TDLComm layer, improved heuristics for generating communication and task execution schedules, and strategies for avoiding the re-generation of schedules when components are added or modified.

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.

References

[1] ARTIST Embedded 2004, Selected topics in Embedded Systems Design:Roadmaps for Research, IST-2001-34820, p114, http://www.artist-embedded.org/Roadmaps/ ARTIST_Roadmaps_Y2.pdf

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

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

[4] CADENCE Design Systems, California, USA http://www.cadence.com/

[5] G.C. Butazzo, *Hard real-time computing systems, predictable scheduling – algorithms and applications*. Kluwer Academic Publishers, 1997, ISBN 0-7923-9994-3

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

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

[8] H. Kopetz, 1998, The Time-Triggered Model of Computation. *Proceedings of the 19th IEEE Systems Symposium (RTSS98), December 1998*.

[9] 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

[10] Java Technology – JMX Remote API http://jcp.org/aboutJava/communityprocess/final/jsr160/index.html

[11] Jane W. S. Liu. *Real-Time Systems*. Prentice-Hall, 2000.

[12] M. Wernicke: *New Design Methodology from Vector simplifies the Development of Distributed Systems*, Vector Informatik Press Release, June 2003, http://www.vector-informatik.com/pdf/press/ PND_DaVinci_PressRelease_200306_EN.pdf

[13] Microsoft .NET platform, http://www.microsoft.com/net/

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

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

[16] Real-Time Corba ACE/TAO http://www.cs.wustl.edu/~schmidt/TAO.html

[17] S. Vestal. *MetaH Users Manual*. Honeywell Technology Center, 3660 Technology Drive, Minneapolis, MN 55418, version 1.27 edition. http://www.htc.honeywell.com/metah/uguide.pdf

[18] T. Fuhrer, B. Muller, W. Dieterle, F. Hartwich, R. Hugel, and M. Walther, 2000, Time Triggered Communications on CAN (Time Triggered CAN - TTCAN). In *Proceedings 7th International CAN Conference*, Amsterdam, Netherlands.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] TTTech - Time-Triggered Technology http://www.tttech.com

[25] Vector Informatik, VECTOR GROUP, Stuttgart, Germany http://www.vector-informatik.com/english/