Bus Scheduling for TDL Components

Emilia Farcas, Wolfgang Pree, and Josef Templ

Department of Computer Science, University of Salzburg, Austria
firstname.lastname@cs.uni-salzburg.at

Abstract. This paper describes a solution for bus scheduling of distributed multi-mode TDL (Timing Definition Language) components. The TDL component model is based on the concept of Logical Execution Time (LET), which abstracts from physical execution time and thereby from both the execution platform and the communication topology. The TDL component model allows the decomposition of hard real-time applications into modules (= components) that are executed in parallel. A TDL module runs in one particular mode at a time and may switch to another mode independently from other modules. This is in contrast with global modes as introduced by other available hard real-time systems and introduces new challenges for bus scheduling.

1 Introduction

Traditionally, the development of software for embedded systems is highly platform specific. However, with more powerful processors available, there is a shift of functionality from hardware to software and the requirements are becoming more ambitious. 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.

A particularly promising approach towards a high-level component model for real time systems has been laid out in the Giotto project [5][8][9][10] at the University of California, Berkeley, 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.

This paper refers to a component model, named TDL (Timing Definition Language) [15], which 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.

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

In the following, we shall start with an explanation of LET and proceed with an overview of the TDL component model. Then, we focus on the distribution of TDL components and describe the problems related to independent mode switches. The description of our approach to automatic bus schedule generation for these requirements is the core contribution of the paper.

2 Logical Execution Time (LET)

LET means that the observable temporal behavior of a task is independent from its physical execution [8]. It is only assumed that physical task execution is fast enough to fit somewhere within the logical start and end points. Fig. 1 shows the relation between logical and physical task execution.

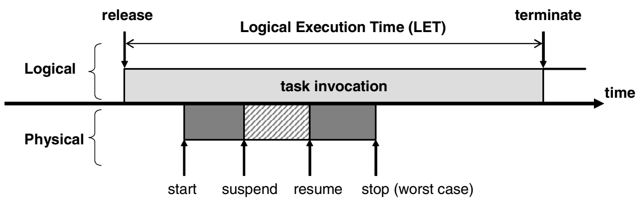

Fig. 1. Logical Execution Time

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 [11]. It is always defined which value is in use at which time instant and there are no race conditions or priority inversions involved. LET also provides the foundation for what we call transparent distribution [3] (see Section 4).

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, and 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, that is, 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.

Parallel tasks within a mode may depend on each other, that is, 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 are the unit of mode switching, that is, every module executes in its own mode and may switch to a different mode independently from other modules,
4. modules serve as units of loading, that is, a runtime system may support dynamic loading and unloading of modules, and
5. 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 fact that modules are the unit of mode switching implies that an application consisting of multiple TDL modules is not in a single global mode. This is in contrast to state-of-the-art systems, which support only global mode switches. Furthermore, the possibility to distribute TDL modules across different computation nodes leads us to the notion of *transparent distribution* as explained in more detail in Section 4 and in [3].

Example TDL Modules

The following TDL source code shows two modules M1 and M2. M1 exports three named constants and two tasks, and M2 imports M1 and may therefore access the exported entities. Module M1 defines two modes of operation, f11 and f12, where f11 is the start mode. Both modes invoke two tasks inc and dec and check the mode switch condition once per mode period, which in both cases is 10ms. The difference between the two modes is that in f12 the task dec will be invoked twice as fast as in f11. Module M2 defines a single mode, which uses the outputs of tasks inc and dec in order to calculate the sum and update an actuator. Depending on the mode of M1, the output will be a constant value or it will change over time. As a developer specifies only the timing behavior in TDL, the functionality of the tasks has to be implemented in another programming language. The functions invoked by the tasks, the drivers for reading sensors and updating actuators, and the guards for conditional execution can be implemented in any imperative programming language such as C. The external functionality code is indicated by the keywords uses and if.

```
module M1 {

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

  sensor
    int s uses getS;

  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 = 10ms
      [freq=1] dec();
    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
    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;
  }
}
```

Fig. 2 shows the outputs of module M1's inc and dec tasks, and module M2's sum task. Module M1 is in mode f11 at the beginning, therefore 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.

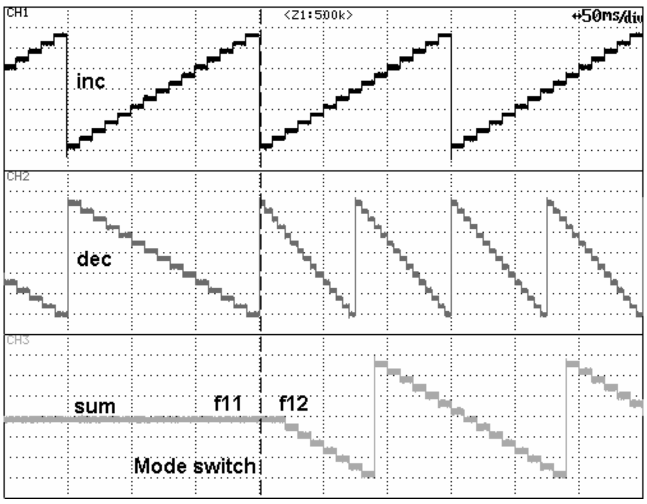

Fig. 2. Functional and temporal behavior of modules M1 (mode f11 and then f22) and M2

4 Transparent Distribution

The term transparent distribution in the context of hard real-time applications is defined with respect to two points of view. Firstly, at run-time a TDL application behaves exactly the same, no matter if all modules (that is, 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 run-time system frees the developer from the burden of explicitly specifying the communication requirements of modules. The mapping of modules to computation nodes is defined separately in a platform configuration file, which also contains the physical properties of the communication infrastructure (e.g., bandwidth, protocol overhead and payload size). It should be noted that in both aspects transparency applies not only to the functional but also to the temporal behavior of an application.

In order to illustrate the importance of LET for transparent distribution, we consider an example of two modules M1 and M2, 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 other words, module M2 imports module M1, and task2 has as input the output port of task1. For this example, we further assume that task2 runs twice as often as task1, that is, the LET of task1 is twice the LET of task2.

Fig. 3 shows an example for the communication required between the two tasks. In order to implement this exchange of information, we assume a communication layer on both nodes that we call TDL-Comm [3]. Its purpose is to send and receive messages at *appropriate* times so that the LET constraint of task1 is met. This means that the output value of task1 has to arrive at node2 before LET1 ends.

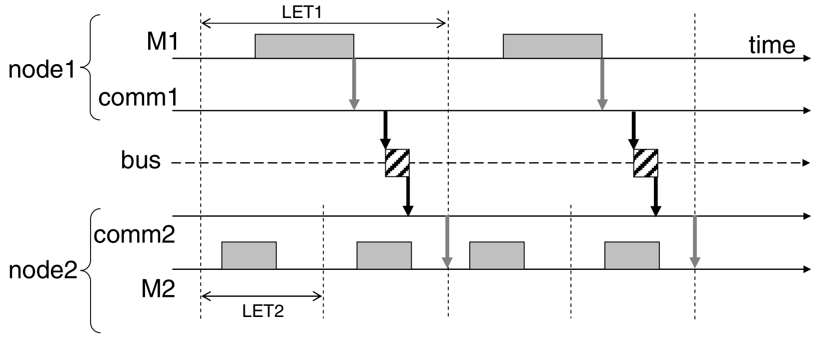

Fig. 3. Sample communication between two tasks

5 Bus Schedule Generation

This section presents the basic concepts, terminology and the algorithm which generates the bus schedule for the TDL component model. The bus schedule is generated at compile time. We do not describe how the TDL tasks are scheduled on the particular node where a TDL module is executed.

5.1 Preliminaries for Bus Scheduling

We assume a network infrastructure based on broadcast semantics, that is, a frame sent by one node can be received at the same time by all other nodes. Furthermore, we assume that packets sent by different nodes cannot be combined into a single packet but are sent as individual network frames according to some protocol. This rules out special support for systems such as EtherCAT, where a frame can be shared by multiple nodes.

The access to the shared communication medium is collision free via a TDMA (Time Division Multiple Access, [12]) approach. In order to support this, we rely on a mechanism for clock synchronization over the network. Furthermore, we adhere to the Producer/Consumer model. This means that the nodes that generate information—the producers—trigger the sending of information over the network. The nodes that need the information—the consumers—do not send any requests to the producers as it is the case in the Request/Response model.

5.2 Mode Switch Instants Per Module

TDL restricts mode switches such that task invocations are never interrupted by a mode switch. Thus, mode switches are said to be *harmonic*, that is, a mode switch must not occur during the LET of every task invocation of the currently active mode. Therefore, the period of a mode switch must be a multiple of the LCM (least common multiple) of the period of tasks invoked in this mode. This check is done during compilation. Furthermore, the mode period is always a multiple of the periods of task invocations and mode switches.

For a given module *M*, we define $mspGCD_M$ as the GCD (greatest common divisor) of mode periods and mode switch periods in all modes in *M*. We know that within the time span $[N*mspGCD_M\ ..\ (N+1)*mspGCD_M]$ there will not be a mode switch within module *M*. In other words, we can express the mode switch instants as an integer multiple of $mspGCD_M$.

5.3 Bus Period

As we generate a static schedule, the size of the schedule needs to be finite. Thus, the schedule is repeated periodically. We call the time span covered by the schedule the *bus period*.

As each mode in every module may have its specific communication requirements, an obvious candidate for the bus period is the longest time span without a mode switch in any module. Thus we calculate the bus period as GCD of the $mspGCD_M$ of each module *M* which communicates on the bus.

Each mode period consists of an integer multiple of bus periods and we introduce the term *phase* in order to distinguish these mutually exclusive parts of a mode.

5.4 Messages

We define the term *message* as the collection of all values of the task output ports produced by a task invocation. Each task invocation produces one message. Note that if a task is invoked *N* times per mode period, *N* messages are produced.

As an optimization, task output ports that are not used by any client are ignored. Furthermore, tasks that are not public or that have no clients produce no messages.

A message has a unique *tag*. The reason for that is explained below. The tag defines the node, module, mode, task invocation, and the phase of the mode in which the message has been produced.

The size of a message is measured in bytes as the sum of the size of the contained values and the size of the tag.

Each message has individual timing constraints. The *release* constraint is the earliest time instant message sending can be started. The *deadline* constraint of the message is the latest time instant when the message sending must be finished.

A simple approach is to set the release constraint to the release time of the task invocation that produces that message plus its worst case execution time (wcet). The deadline constraint results from the end of the LET of the producer task invocation. The release and deadline of a message are relative to the phase where the task invocation ends.

5.5 Frames Per Module

In order to use the communication medium efficiently, we map the messages of a phase to one or more reserved communication windows within the bus period such that these communication windows can be used for all phases of a module. A reserved communication window corresponds to a *frame*, which is the unit of information to be sent on the bus. The exact point in time when the frame will be scheduled within this communication window is computed later, see Sect.5.7

The schedule generator determines the frames and binds each message to exactly one frame. At run-time, the phase of a module determines which subset of the messages bound to a frame is actually sent. As the content of a frame varies at run-time, we need a means to identify messages. For that purpose we have introduced the message tag as described above.

The *release (r)* constraint of a frame is the maximum of the release constraints of the bound messages. The *deadline (d)* constraint of a frame is the minimum of the deadline constraints of the bound messages. The schedule generator guarantees that the frame size and constraints are sufficient for the communication requirements of all phases.

To exemplify this, we consider a module with a mode of execution that has three phases, and we assume that it produces a message of 4 bytes in phase0, a message of 3 bytes in phase1, and two messages of 1 byte each in phase2. Depending on their size and timing constraints, all messages may be bound to the same frame in the schedule, as seen in Fig. 4. The left and right bounds of the message and frame boxes represent the release and deadline constraints.

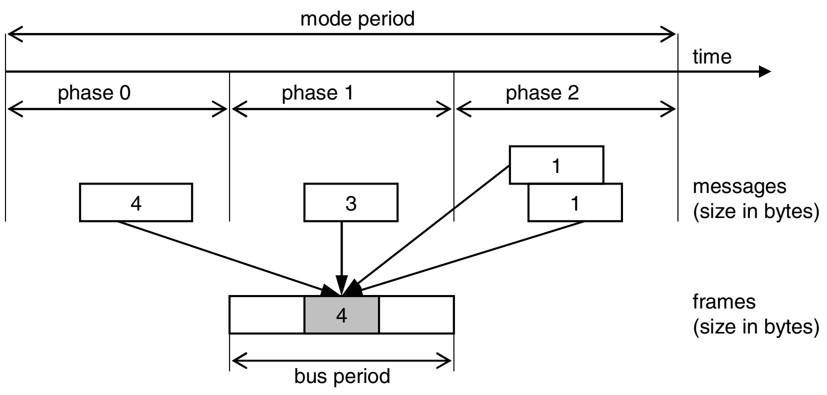

Fig. 4. Sample binding of several messages to one frame

The following pseudo code shows how messages are extracted and bound to frames. We assume that the bus period is globally available.

```
createFrames(Module M) returns Set {
    let frames be an empty set
    for each mode m of module M {
        for each phase p of m {
            let msgs be an empty set
            for each task invocation instance t that ends in p {
                add new Message(M, m, t, p) to msgs
            }
            bindMsgs(msgs, frames)
        }
    }
    return frames
}
```

The following pseudo code refines bindMsgs, which associates a message with an existing frame if possible. Otherwise a new frame is created and the message is bound to the new frame. The method createFrame creates a frame, binds the message to that frame, sets the size of the frame to the size of the message, and adds the frame to the set frames. It also checks if the size does not exceed the maximum allowed on the network and if the frame transmission time fits in the bus period.

The decision of binding a message to a frame depends on the result of metric computation and on how many bytes we still have available from the size of the frame. We define for each frame the instance variable *available*, which is reset at the beginning of each phase to the size of the frame. The method bind binds a message to a frame and reduces the available bytes of the frame by the message size. The concept of computing metrics is explained below.

```
bindMsgs(Set msgs, Set frames) {
    reset the available bytes of all frames to the size of each frame
    for each msg in msgs {
        if (frames is empty) {
            createFrame(msg, frames)
        } else {
            for each frame in frames {
                computeMetric(msg, frame)
            }
            select the frame selFrame with the highest metric
            if (selFrame.metric > threshold) {
                bind(msg, selFrame)
            } else {
                createFrame(msg, frames)
            }
        }
    }
}
```

5.6 Heuristics

The method computeMetric calculates a real number between 0 and 1 and stores that number in the instance variable metric of a frame. For each message, we choose the frame that has the highest value for the metric, and if that value is higher than a threshold (e.g, equal to 0.5), then we bind the message to the frame. The allocation of messages to existing frames introduces a tradeoff between saving bandwidth and tightening the timing constraints. Hence, the topic is subject to further optimizations and heuristics.

The metric measures the degree of overlapping between the message and frame windows. We define the *window*, for a message or a frame, as the time interval between the release and deadline. If we allocate the message to this frame, then the new timing constraints for the frame will be the window of the overlapping section. Therefore, we want this to be as close as possible to the message and to the existing frame, otherwise the timing constrains would be too restrictive and we reduce the chance to find a feasible schedule. The overlapping and the metric as an average percentage are defined by the following formulas:

$$overlapping = Min(frame.d, msg.d) - Max(frame.r, msg.r) \quad (1)$$

$$metric = \frac{\frac{overlapping}{frame.d - frame.r} + \frac{overlapping}{msg.d - msg.r}}{2} \quad (2)$$

5.7 Bus Schedule

For each module in the system, we have identified the required messages and mapped them to frames, but the communication windows of different frames could overlap. Therefore we collect the frames required by all modules and apply on this global set a variation of the Reversed EDF scheduling algorithm, that is, the Latest Release Time (LRT) [13].

This decides when each frame must be sent on the network, depending on the release, deadline and worst case transmission time of each frame. Furthermore, the bus scheduler has additional constraints that 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 computing nodes. The bus scheduler also generates extra frames, for example for time synchronization. Furthermore, it merges adjacent frames in the sorted list of frames if they are sent by the same node. This leads to the remapping of the corresponding messages to the merged frame.

6 Related Work

The state-of-the-art methods and tools for the development of distributed systems support at most global mode switches. By our knowledge, there is no other available system that allows real-time components to switch modes independently. Furthermore, the LET abstraction is the only model that leads to predictable real-time applications in both value and time determinism [11], thus we will emphasize the distribution approach in Giotto. Then we will present an example of static off-line scheduling, the TTP/C protocol. Another scheduling approach, especially in the automotive industry (DaVinci [18], dSPACE[2]) is to use a real-time kernel with dynamic scheduling (e.g. OSEK[14]) and a communication system based on static priorities (e.g., CAN[1]), therefore the system cannot be predicted and it has to be simulated as whole.

The Giotto language [8][9][10] focuses on task distribution, therefore it provides support only for global modes, and only one program runs in the system. [6] presents a methodology for distributed real-time code generation, thus multiple suppliers can independently compile different parts of a Giotto program to run on multiple CPUs. A system integrator assigns each task a particular host and supplier, by annotating the Giotto source code. Each supplier receives a part of the Giotto program, and a timing interface specifying the time slots that can be used for the task and communication scheduling. Given these, each supplier produces code, and then the integrator checks the interface compliance and the time safety, that is, if the code meets the Giotto timing requirements (e.g., release and deadlines) on a given platform. The schedule is generated off-line in form of virtual machine code, that is, S code [7]. The timing interface provides the exclusive time windows for scheduling, but not exactly when to perform the actions within the windows, so the supplier still has some flexibility. However, these timing interfaces are currently generated manually. Furthermore, the approach [6] is described by means of a single mode Giotto program. So it is unclear if a distributed multi-mode Giotto system has ever been implemented, though that would still stick to the global mode switch approach.

The time-triggered protocol (TTP) [12] is a communication protocol for fault-tolerant distributed hard real-time systems. It provides time-triggered communication, distributed clock synchronization and a membership service. The communication on the bus is done with static, periodic TDMA rounds. In TTP/C [17] the schedule is implemented as a message description list (MEDL), specifying exactly when a node has to send a certain message and when it has to receive messages from other nodes. A TTP cluster cycle consists of multiple TDMA rounds and the messages sent in a TDMA round can differ throughout the cluster cycle. A task descriptor list describes the cyclic scheduling of application tasks, thus at run-time the scheduler is a simple dispatcher. The TTTech [16] tool chain consists of two main tools for application development. The TTPplan tool generates the bus schedule (cluster level design). The TTPbuild tool generates the task schedule (node level design). The developer must specify in TTPplan every message that is sent from any node, and then in TTPbuild every periodic task and the messages it consumes. This is in contrast to our approach where the required messages are automatically identified from the TDL code. Furthermore, the TTP/C protocol supports global mode switches. The length of the cluster cycle can be changed from cluster mode to cluster mode and the messages transferred in the rounds of each node can be changed as well. However, although the protocol is designed for mode switches per subsystem, the current limitation of the TTTech tools is that they only support a single global mode of execution.

Regarding off-line scheduling and flexibility in real-time systems, [4] describes algorithms to support also aperiodic messages and to switch modes at run-time. When the condition for a mode change is enabled, the mode change request is communicated within a message on the network. All nodes receive the request at the same time, and perform the mode switch at the same time (that is, there is a consistent view of the mode switch requests). The duration of the mode switch results from the delay in the current schedule until it gets to a slot where the switch is feasible, and the duration of a transition schedule. This mode switch delay can be computed off-line and tested to be lower than some deadline set at design time.

7 Conclusions

The LET abstraction invented in the realm of the Giotto project paved the way for transparent distribution in real-time systems. We think this novel approach will lead to significantly more robust embedded software and will reduce the costs of integration testing. The TDL component architecture implies that modes may switch independently in each component, which is a radical innovation in real-time systems. We presented a scheduling algorithm for message communication, to support these independent mode switches, while maintaining transparent distribution. Future research and implementation efforts are required to show the scalability of transparent distribution and the scheduling algorithm. Another set of challenges comprises optimizations, improved heuristics and metrics for generating the communication schedules, considering the feedback from the time safety check for task execution, 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. Bosch, 1991, *CAN Specification, Version 2*. Robert Bosch GmbH, http://www.can.bosch.com/docu/can2spec.pdf
2. dSPACE GmbH: http://www.dspace.de
3. E. Farcas, C. Farcas, W. Pree, J. Templ. Transparent Distribution of Real-Time Components Based on Logical Execution Time, *Proc. of ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (LCTES), ACM Press, 2005, pages 31-39
4. G. Fohler, Flexibility in Statically Scheduled Real-Time Systems, PhD Thesis, Technisch-Naturwissenschaftliche Fakultaet, Technische Universitaet Wien, Austria, April 1994
5. Giotto Project, http://www-cad.eecs.berkeley.edu/~fresco/giotto/
6. T.A. Henzinger, C.M. Kirsch, and S. Matic, Composable Code Generation for Distributed Giotto, *Proc. of ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (LCTES), ACM Press, 2005, pages 21-30
7. T.A. Henzinger, C.M. Kirsch, and S. Matic. Schedule carrying code, *Proc. of the Third International Conference on Embedded Software* (EMSOFT), LNCS, Springer-Verlag, 2003
8. 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.
9. 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.
10. 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.
11. C.M. Kirsch, 2002, Principles of Real-Time Programming. *In Proceedings of EMSOFT 2002,Grenoble* LNCS, 2491.
12. H. Kopetz, 1997, *Real-time Systems: Design Principles for Distributed Embedded Applications*. Kluwer, 1997

13. Jane W. S. Liu. *Real-Time Systems*. Prentice-Hall, 2000
14. OSEK Group, 2001, OSEK/VDX Time-triggered Operating System Specification, Version 1.0, http://www.osek-vdx.org/mirror/ttos10.pdf
15. 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
16. TTTech - Time-Triggered Technology http://www.tttech.com
17. TTTech. Time-Triggered Protocol TTP/C High-Level Specification Document. Edition 1.0.0, July 2002.
18. 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