Int J Software Informatics, Volume 5, Issue 1-2 (2011), Part II, pp. 335–354 E-mail: ijsi@iscas.ac.cn
International Journal of Software and Informatics, ISSN 1673-7288 http://www.ijsi.org
©2011 by ISCAS. All rights reserved.
Tel: +86-10-62661040

TDL – Steps Beyond Giotto: A Case for Automated Software Construction

Wolfgang Pree, Josef Templ, Peter Hintenaus, Andreas Naderlinger,

and Johannes Pletzer

(C. Doppler Laboratory Embedded Software Systems, University of Salzburg, Austria)

Abstract We present the Timing Definition Language (TDL), which supports the development of dependable embedded real-time systems. TDL is conceptually based on the time-triggered programming model introduced by Giotto but provides a more convenient syntax, more control over the timing of periodic activities, an industrial strength tool chain, and, most importantly, adds a component model and supports the integration of asynchronous activities in a time-triggered system. We present the introduced language concepts and outline the TDL-based tool chain, which also includes support for simulation, distribution, and automatic code generation. Finally, we show an example that uses some of the extensions and compare TDL with other extensions of Giotto.

Key words: timing definition language; TDL; Giotto; logical execution time; LET; synchronous; asynchronous; component model; real-time; simulation

Pree W, Templ J, Hintenaus P, Naderlinger A, Pletzer J. TDL – steps beyond Giotto: a case for automated software construction. *Int J Software Informatics*, **Vol.5, No.1-2 (2011), Part II: 335–354. http://www.ijsi.org/1673-7288/5/i87.htm**

1 How It All Began

Manfred Broy represents one of the light towers of research in software engineering. I have been lucky to meet and cooperate with Manfred early in my scientific career: Back in 1993 the research branch of Siemens in Munich, located on the beautiful Perlach campus, set up a visionary research project called Automated Software Engineering (ASE). Manfred was one of the university research partners, together with Gust Pomberger from the Johannes Kepler University Linz and me as Assistant Professor at Washington University in Saint Louis. Though all partners and team members contributed to the success of the ASE project, it was definitely Manfred who set the sails. I'd summarize, a bit oversimplified, the direction he set as follows: Let us define a solid theoretical framework with appropriate abstractions that form the programming model for the target domain, which was ubiquitous computing. Based on these abstractions we should try to automatically generate executable code. In other words, the project should aim at the refinement of a higher level of abstraction that is appropriate for the particular domain, so that a more detailed and

Corresponding author: Wolfgang Pree, Email:wolfgang.pree@cs.uni-salzburg.at
Received 2010-10-09; Accepted 2011-01-03; Final revised version 2011-01-17.

platform-specific level can be automatically generated. Based on that general direction and his valuable ideas how such abstractions could look like, we came up with what would nowadays be called a model-based approach for ubiquitous computing. The ASE project was a huge success for Siemens and all its partners.

Later on in 2000-2001 while I spent a sabbatical with Tom Henzinger at the University of California, Berkeley, Manfred and I met at the first EmSoft (Embedded Software) conference that was initiated by Tom and Christoph Kirsch. The remote resort near Lake Tahoe was an ideal place to discuss ideas and get inspirations for future research directions. Manfred presented results from his exciting projects in the domain of embedded systems. Some of the research activities of my group already focused on embedded systems. Manfred nicely sketched the numerous opportunities he sees from a research perspective. For me this also corroborated that an uncompromised focus on the domain of embedded systems is worthwhile to pursue. This journal contribution summarizes some of the results that my research group produced during the decade that followed. The master mind behind the Timing Definition Language (TDL) is Dr.-ETH Josef Templ, a senior researcher in my group, whose exceptional talent for balancing theory and practice paired with a relentless strive for lean software made TDL a solid foundation for research as well as for its commercial success.

Since I moved to Salzburg in 2002 Manfred and his wife Karin together with my wife Ingrid and me enjoyed performances at the Salzburg Festival every year, paired with thought-provoking discussions about research directions and the state-of-the-art in software engineering. I thank Manfred for helping to point out what has proved to be a visionary research framework in software science, to motivate me as young researcher and become a much appreciated friend.

2 Introduction

The Timing Definition Language (TDL)[13] aims at supporting the development of deterministic, portable software for dependable, embedded real-time systems. It is conceptually based on the time-triggered programming model introduced in Giotto[16] but goes beyond Giotto in a number of aspects. While Giotto is basically an abstract mathematical model of a time-triggered language with a rather simple tool chain that primarily proofs that it can be implemented, the TDL project aims at a tool chain that makes Giotto's concepts available for real-world industrial projects.

This paper describes the main evolution steps that we found necessary for successfully applying the Giotto concepts in practice. We assume a basic knowledge of the underlying Giotto concepts but also try to repeat some of them (in prose form) in order to make the presentation more self-contained. The development of TDL started in 2003 and continued in a sequence of steps until today, where the integration of asynchronous activities in 2008 marked a significant milestone.

Giotto introduced the notion of Logical Execution Time (LET) for the semantics of a task invocation. The data flow (reading input ports and writing output ports) is done at well-defined time instants independent from the actual execution speed of the underlying platform given that the platform is fast enough for dealing even with the task's Worst Case Execution Time (WCET). The execution of the task's body is considered to be a long running operation that cannot simply be ignored. On the other side, arranging the data flow (e.g. reading input ports and writing output

ports) is considered a Logical Zero Time (LZT) operation. A compiler transforms a Giotto program, i.e. a timing definition, into instructions of a virtual machine (called E-code) which is executed by an appropriate runtime system (called E-machine)[19].

TDL inherits these basic concepts and adds new ones as described below. We shall start with a description of new language concepts and continue with tool related extensions. Finally, we shall give an example of a TDL application that uses some of the introduced extensions and we shall compare our work with other extensions of Giotto.

3 New Language Concepts

This section describes the language concepts that TDL introduces as extensions of Giotto. We will not describe the language syntax formally but focus on the underlying concepts. Note that Giotto is basically an abstract mathematical model of a language. There exists, however, a prototype Giotto compiler that introduced a concrete syntax which is also used in source code examples in the original Giotto papers[16,19]. A detailed TDL language specification including an EBNF grammar can be found in Ref.[13]. The syntax we have chosen can also be seen in the example section below.

3.1 General language refinements

TDL tries to be more user-friendly than Giotto by removing the notion of a *driver* from the language. A Giotto driver is essentially a task prolog or epilog, i.e. a sequence of memory copy operations that is either used for reading input parameters or writing output parameters. TDL specifies the input parameters directly with a task invocation as it is also done in a function call in normal programming languages. The drivers still exist internally at the implementation level but they are generated automatically.

Giotto requests to specify all task ports (input, output, state) globally. TDL allows treating a task as a name space for task ports. In addition, TDL also allows the usage of global output ports.

As additional syntactical refinements we mention the introduction of named constants and user defined data types in TDL. We refrain from going into any details here because these features are expected from any high level programming language.

3.2 Adding a component model

A Giotto program is an (anonymous) automaton that consists of a set of states (called modes), where every mode specifies a set of periodic activities. TDL encapsulates one such automaton in a (named) container called a *module* and allows multiple modules to run in parallel. Due to the LET semantics of TDL tasks, the timing behavior of a module is not affected by adding other modules to an application as long as the overall set of modules passes a time-safety check.

In the simplest case, modules are independent from each other and are used for ECU (electronic control unit) consolidation, for example.

In general, modules may depend on each other, i.e. the output produced by a task of one module may be the input of a task in another module. This is accomplished by means of an *import* declaration and a qualification of port names by a module identifier. TDL supports cyclic imports of modules in order to express a cyclic data

flow between modules. This requires some adjustments on the implementation level (E-code structure) as will be described later.

The following list summarizes the purposes that TDL modules serve.

• Modules provide a named program unit. Module names may be structured (e.g. com.my.app1.M2) in order to create globally unique module names.

• Modules introduce a name space and allow for exporting a program entity to other modules.

• Modules act as unit of composition. All modules are executed in parallel, so we have a parallel composition in the TDL component model.

• Modules partition the set of actuators. An actuator may only be assigned a value within the module it is defined in. This ensures deterministic actuator updates.

• Modules act as unit of static linking or dynamic loading, depending in the capabilities of the execution platform.

• Modules act as unit of execution. Every module may provide a start mode which will be executed at startup time. The execution of all modules is synchronized to a common time base even when the modules are executed on a distributed platform.

• Modules act as editing unit in our Simulink integration approach. We provide a Simulink library block that represents a module and behind this block is a graphical editor for the ingredients of the module. The task implementations can be done by normal Simulink subsystems. A module block can be connected to other Simulink blocks via its sensor and actuator ports.

• Modules act as unit of distribution. In the TDL tool chain a module is assigned to a node of a potentially distributed platform. The requirements for the (remote) data flow in a distributed system can be deduced automatically from the modules and a scheduler can compute the network schedule automatically.

3.3 Adding slot selection

The timing specification of a periodic activity in Giotto is based on the mode period p and the activity's frequency number f. The activity is performed$f$ times per mode period. In case of a task invocation activity, the LET of the invocation is implicitly specified as p / f and the mode period is filled with f such task invocations. As shown in Fig.1, TDL supports a more fine grained specification of the timing of periodic activities by means of a mechanism called *slot selection*, where the Giotto timing is only a special case (actually the default case).

A TDL mode period is divided into a sequence of f slots with length p / f. A task invocation is associated with a sequence of adjacent slots. This sequence defines the LET of the task invocation explicitly and decouples the task's period from its LET. In practice, it allows one to specify, for example, breaks between invocations or to define that a task should be invoked at the beginning or at the end of a mode period only. As a consequence, it may help the task scheduler to find a feasible schedule and it may reduce the latency in the data flow between invocations of different tasks.

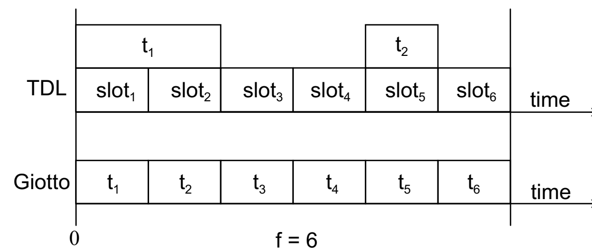

Figure 1. Giotto and TDL timing models

3.4 Adding digital controller support

Control theory states that as a rule of thumb the reaction time of a (digital) controller should be below 10% of the sample time in order to achieve stable controller behavior. In the Giotto model, the reaction time equals 100% of the sample time, which requires a high degree of oversampling in order to achieve the same result. Since this oversampling increases the CPU load unnecessarily, we added special support for digital controllers in TDL by means of *task splitting* and *task sequences*.

A Giotto task is associated with a single external function that represents the task's body. Similar to Simulink's S-functions [20], a TDL task may be associated with two external functions (also called task splitting), a long running function that corresponds with Giotto's task function and Simulink's Update function, and an LZT function that acts as Simulink's Output function. The basic idea is that the Output function is called first at the LET start and provides the new output values in a very short time, closely approximating LZT. The Update function is executed during the LET and prepares the task's internal state by some advance calculations such that the next call of the Output function can be done fast. This can be utilized e.g. for digital controllers which need to evaluate a polynomial as the core of their implementation.

In addition, TDL allows for performing an actuator update right after the call of the Output function by means of so-called task sequences. A task sequence consists of a task invocation followed by a set of actuator updates. The actuator updates of a task sequence are carried out right after the task's outputs are available, which is at the LET start if an Output function is available.

3.5 Semantic differences to Giotto

There are some small but noticeable differences in the semantics of the time-triggered programming model between Giotto and TDL. The following list describes all of them.

• [Program start] In Giotto, the actuator updates and mode switches of the start mode are executed at time zero. There are, however, no further actuator updates or mode switches after a mode switch at time zero. A TDL module is started by simply switching to the start mode. This means that at time zero, there are neither actuator updates nor mode switches.

• [Non-harmonic mode switch] Giotto allows a mode switch even if there are running tasks as long as those tasks exist with the same task period in the target mode. TDL does not allow non-harmonic mode switches because there may be delays involved when switching to the target mode. Furthermore, the task will deliver output

values to the target mode, which do not correspond to inputs specified there.

• [Deterministic mode switch] Giotto requests that among all mode switch guards of a mode only one may return true at a particular point of time. In contrast, TDL evaluates mode switch guards in textual order from top to bottom and performs the first mode switch whose guard returns true. This definition allows a more efficient implementation without compromising determinism.

• [Guarded actuator update] A guarded actuator update in Giotto means that the actuator setter is called independently of the guard's result. In TDL, actuator update and actuator setter are both guarded and performed only if the guard returns true.

• [Mode port assignments] Assignments of task output ports upon a mode switch is done as an initialization in the affected target task in TDL. In Giotto it is performed before the target task is invoked, thus, it is visible to clients earlier and thereby implies problems for distributed execution.

• [Sensor read] Giotto defines that sensors are read right before task invocations and, as a consequence, sensor values used e.g. for actuator updates or mode switch guards are old values. TDL uses current values for sensors in all places in order to provide deterministic behavior even in the case that multiple modules access a shared sensor. However, a sensor is read only once for every logical time instant.

3.6 Adding asynchronous activities

In addition to time-triggered (alias synchronous) activities, there is often a need for executing event-triggered (alias asynchronous) activities as well. TDL adds support for asynchronous activity sequences consisting of task invocations and actuator updates. An asynchronous activity sequence is triggered by the occurrence of one of the following events.

• [Hardware interrupt] A (non maskable) hardware interrupt has the highest priority in the system. It may even interrupt synchronous activities. The TDL semantics therefore takes care that the impact of hardware interrupts on the timing of synchronous activities is minimized. In addition, it is assumed that a maskable interrupt is switched off until the associated event is executed. This reduces the danger of denial of service due to a large number of interrupts. Hardware interrupts may be used e.g. for connecting the system with asynchronous input devices.

• [Asynchronous timer] A periodic asynchronous timer may be used as a trigger. Such a timer is independent from the timer that drives the synchronous activities because it introduces its own time base. An asynchronous timer may for example be used as a watchdog for monitoring the execution of the time-triggered operations.

• [Port update] Updating an output port may trigger an asynchronous activity. We assume that both a synchronous and an asynchronous port update may be used as a trigger event. In case of a synchronous port update, i.e. a port update performed in a time-triggered activity, the TDL semantics takes care that the impact on the timing of the synchronous activities is minimized. Port update events may e.g. be used for limit monitoring or for change notifications.

Events may be associated with a priority and are registered in a priority queue when they arrive. Processing the event is delayed and supposed to be performed by a single background thread that runs whenever there are no time-triggered activities to

perform. Reading input ports is done as part of the asynchronous execution, not at the time of registering an event. Output ports are updated right after an asynchronous task invocation has been finished. If an event reoccurs before it has started processing it will not be executed twice but remains registered once.

The TDL runtime system ensures correct synchronization of the data flow between synchronous and asynchronous activities. It has been shown in Ref. [12] that a lock-free synchronization approach is possible. In case of a distributed system, the communication of asynchronous output values to remote nodes is supposed to rely on asynchronous network operations, i.e. it may be delayed.

4 Implementation

This section outlines the TDL tool chain, which builds on core ideas introduced in Giotto[19] but provides a clean room implementation that supports all of TDL's extended features. We shall start with an overview and then go deeper into the format and structure of the E-code we introduced for TDL.

4.1 Tool chain overview

Figure 2 outlines the TDL tool chain. It shows as a central component the TDL compiler, which offers a plug-in-architecture based on an abstract syntax tree (AST) for generating target platform specific output in addition to platform independent .ecode files. The E-code together with platform specific output and the functionality code is used by the E-machine, which provides the runtime system for executing TDL programs. As an optional graphical front-end we provide a tool named TDL Visual Creator, which can also be integrated in MATLAB/Simulink. In case of the Simulink integration, the functionality code can be generated automatically from the Simulink model by a standard MATLAB tool named Real-Time Workshop Embedded Coder (RTW-EC). For organizing the various build steps, in particular for distributed systems, we provide a tool named TDL Visual Distributor, which is not shown in the figure.

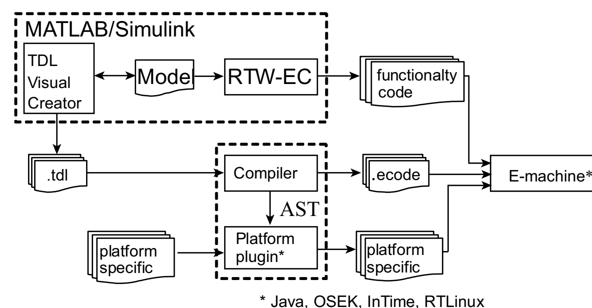

Figure 2. TDL tool chain overview

4.2 E-code structure

The TDL compiler generates E-code for each mode of a module. The E-code covers a single mode period and is repeated then by means of a *jump* instruction to the beginning of the mode. For every logical time instant which needs to execute

E-code one E-code block is generated. An E-code block is a list of E-code instructions terminated by a *return* instruction. It specifies for one logical time instant the actions that must be taken by the E-machine in order to comply with the timing specifications and LET semantics.

The structure of a TDL E-code block follows from the requirement for parallel execution of a set of modules rather than executing only a single module as it is the case in Giotto. If two modules execute E-code at the same logical time, we must take care that all modules use the current values of output ports because modules may depend on each other, i.e. there may be a data flow between modules and this data flow may also be cyclic. Therefore we use a multi-phase strategy and start every E-code block with a task termination section whose end is marked by a special instruction in the E-code. The execution of a set of modules always starts with executing the task termination section of all modules (that need to execute E-code at a particular time) first. Then we know that all output ports are updated and can be used in any order. The rest of the E-code instructions could in principle be executed in a second phase but for the purpose of simulation (e.g. with Simulink) we introduce three sections in total. The second section covers all instructions that are necessary for performing actuator updates, and the third section checks for mode switches and releases tasks. The following sequence of actions comprises one E-code block for a logical time instant t:

1. Update output ports of task invocations logically terminating at t with the result values from its execution.
2. Mark end-of-task-termination (EOT) section.
3. Update actuators that are defined to be updated at t.
4. Mark end-of-actuator-update (EOA) section.
5. Switch mode if a mode switch is defined at t.
6. (*) Update input ports of tasks that are defined to be released at t.
7. Release tasks that are defined to be released at t.
8. Advance the module's future time to the next logical time instant t + *delta-Time* and register the address at which to continue.
9. Return from E-code block.

In case of a mode switch (and also for startup) execution starts at (*) of the first E-code block of the target mode. This is the mode's entry point.

Note that it is not practical to generate E-code that covers all modules at once because modules may switch mode independently. An enormous code explosion would result from generating E-code for all possible combinations.

4.3 E-code instruction set

The E-code instruction set is defined in Table 1. It differs from the E-code instruction set introduced in Giotto by (1) using only up to 2 arguments and thereby saving some memory, (2) by adding the *switch* and *repeat* instructions, and (3) by adding an argument to the *nop* instruction, which is used as a marker for the various sections in one E-code block.

The *repeat* instruction has been added as a means for E-code compression because it turned out that in some cases there are repeating patterns that can be compressed effectively by an iteration construct.

Table 1 E-code instructions

Instruction	***Meaning***
nop(f)	A dummy (no operation) instruction. The argument f is used as a marker for identifying different sections in the E-Code.
call(d)	Executes the driver d.
release(T)	Marks the task T as ready for execution.
future(a, dt)	Plans the execution of the E-Code block starting at address a in dt microseconds.
if(g, *elsePC*)	Proceeds with the next instruction if g evaluates to true else jumps to *elsePC*.
jump(a)	Jumps to the instruction at address a.
return	Terminates an E-Code block.
repeat(a, n)	Uses a counter per module for jumping n times to instruction a. After that it continues with the next instruction.
switch(M)	Performs a mode switch to mode M, i.e. the E-machine continues at the entry point of M. In addition, the module's repeat counter is set to zero.

The *switch* instruction has been added because a mode switch differs from an ordinary *jump* instruction and detecting a mode switch at runtime by checking the target address of a *jump* is an unnecessary overhead. A *switch* may need to perform additional internal housekeeping work and it needs to reset the counter used by the *repeat* instruction.

4.4 Adding distribution support based on modules

The original Giotto papers already envision the possibility of distributing a Giotto program as one of the core advantages of the LET concept. Distribution is supposed to be based on tasks, which are assigned to different nodes of a distributed system. TDL differs regarding the unit of distribution. It takes the module as the distribution unit, i.e. it assigns a module to a node of a distributed system. This follows naturally because a module also contains sensor and actuator ports, which are bound to specific I/O devices that need to be available on a node. More details about distribution in TDL can be found in Ref.[9].

4.5 Simulation and code generation

Simulation of LET-based systems is particularly useful since simulation results of the synchronous activities exactly match the behavior on any target platform. The MATLAB/Simulink environment[20] offers support to model, simulate, and analyze plant and controller dynamics, and also supports code generation (e.g. Real-Time Workshop Embedded Coder).

Giotto's S/G tool applies a transformation to Simulink models in order to adhere with the timing specification and LET semantics in the simulation[17]. More precisely, the data flow is adjusted using standard Simulink blocks such as Zero-Order-Hold and Unit-Delay. However, this approach fails for complex mode switching logic and for tasks with individual execution rates as has been shown in Ref.[11].

The Simulink integration in the TDL tool chain uses a model transformation with an E-machine implementation for Simulink at its core. While the task and guard functionality itself is modeled in Simulink, the timing behavior is specified

in TDL. Drivers and wrappers for tasks and guards are automatically generated as function-call subsystems and are connected via Simulink signals. The input ports of a driver block are directly connected to its output ports, which corresponds with assignments in an imperative program. We implemented an E-machine using the S-Function mechanism provided by Simulink to timely trigger their execution and thus to ensure TDL semantics. To avoid restrictions on the set of supported blocks (e.g. for the plant) caused by Simulink's block execution strategy, we split duties of the E-machine among two collaborating S-Functions. This allows Simulink to execute the plant or other blocks after actuators are updated and before sensors are read. As a precondition, E-code instructions for actuator updates must be separated from all other instructions, which motivates the E-code trisection as outlined earlier. An additional delay block between the release and the termination driver of a task and between the two E-machines enables Simulink to resolve algebraic (feedback) loops without affecting the timing behavior. Figure 3 outlines the Simulink model that results from our model transformation. To ensure that data flow between synchronous and asynchronous activities follows TDL semantics and to preserve the different priority levels, also the asynchronous part is handled by the E-machines.

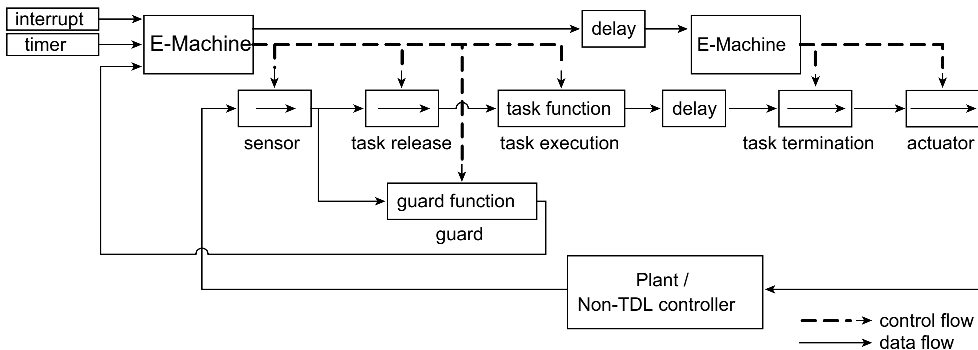

Figure 3. Transformed simulation model

In case of simulating asynchronous activities, the simulation is not guaranteed to match the behavior of a specific target platform because the simulation is not aware of any scheduling strategy, distribution topology, or CPU speed of the target platform.

The Real-Time Workshop Embedded Coder (RTW-EC) was already used as part of Giotto's S/G tool to generate C code for tasks and guards from Simulink subsystems. Additionally, for supporting TDL's task splitting we make use of the possibility to split a Simulink task function implementation into an Output and an Update function.

4.6 Threading model for integrating asynchronous activities

Besides the TDL component model, the integration of asynchronous operations is the most significant extension of TDL over its predecessor Giotto. The detailed description of the implementation, in particular the synchronization of the data flow between synchronous and asynchronous operations, is beyond the scope of this paper. We shall, however, give an overview of the basic ideas. It has been shown in Ref.[12] that a lock-free synchronization is possible with the semantics we have chosen.

Figure 4 outlines the involved threads including their priority and the critical regions that need synchronization. The time-triggered activities are represented by a thread named E-machine. This thread may need further internal threads but we assume that all synchronization issues are concentrated in a single thread that coordinates the time-triggered activities.

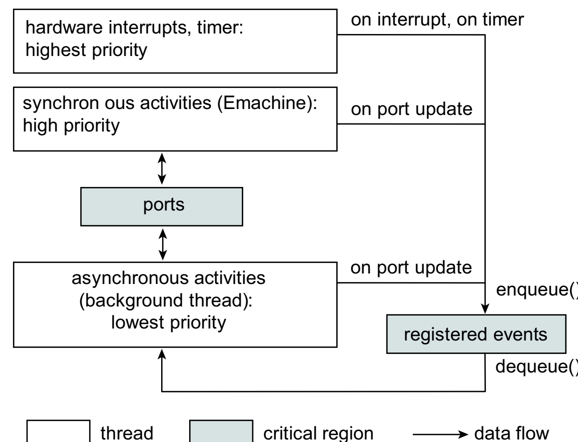

Figure 4. Threads and critical regions

Elements are enqueued in the queue of registered events when an asynchronous event occurs and the event is not yet registered. As mentioned earlier, an event can be a hardware interrupt, an asynchronous timer event, or a port update event. Port updates may originate from an asynchronous task or from a synchronous task that is executed by the E-machine. enqueue will never be preempted by dequeue, however, enqueue may be preempted by other enqueue operations. Elements are dequeued by the single background thread that executes asynchronous activities. This thread may be preempted by interrupts and by the E-machine. Thus, dequeue may be preempted by enqueue operations.

Since enqueue operations that originate from interrupts or synchronous port updates affect the timing of the E-machine, it is important that enqueue is an efficient, constant time operation. This can be achieved by using an array representation for the registered events with a simple Boolean flag per event for expressing that the event is pending.

Access to ports (which are essentially global variables) must be synchronized such that asynchronously reading a set of input ports respectively writing a set of output ports always appears as an atomic action and it must interfere with the timing of the E-machine as little as possible.

5 Example

As an example for a real-world TDL application we present an augmented strap down inertial navigation system (INS)[12] designed for computing the position, velocity, and attitude of a sailing vessel at sea. The example uses asynchronous activities for connecting asynchronous I/O with the time-triggered navigation system core.

An INS determines the position of a vehicle with respect to some (inertial) reference system by measuring the three accelerations along and the three angular velocities around the vehicle's axes with respect to the reference system, using three accelerometers and three gyroscopes which are firmly attached to the vehicle's body. By solving the equations of motion the INS computes the position, velocity, and attitude of the vehicle. An augmented INS uses additional inputs, such as position information from a GPS receiver and compass headings, to correct the drift of the inertial sensors.

5.1 Hardware

The hardware (see Fig.5) for the augmented INS consists of an Analog Devices ADSP-21262 Signal Processor[4], an LAN interface with TCP/IP functionality in firmware, an ADIS family micromechanical inertial sensor[5] and a two axis fluxgate compass[6]. Besides a floating point signal processing core with a peak SIMD performance of 1.2 GFlops, the ADSP-21262 contains an I/O processor that is capable of managing several block transfers between memory and periphery simultaneously. The inertial sensor is connected to the signal processor using an SPI bus[10]. It samples the rotations around the three axes of the vehicle and the accelerations along these axes 819.2 times per second. The excitation coil of the fluxgate compass is attached to the ADSP-21262 using a sampling DA converter. The two sense coils of the compass are connected to two sampling AD converters. All three converters operate at 48K samples per second. For determining the heading of the vehicle the compass has to be excited periodically via the DA converter and its response measured via the two AD converters.

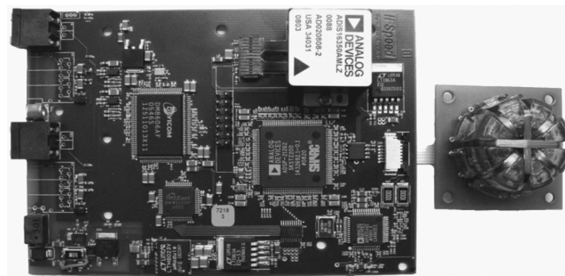

Figure 5. INS hardware

5.2 TDL definitions

A TDL module starts with its name and the list of imported modules. When importing a module it is possible to define an abbreviation for it:

```
module INS {
    import Kalman as K;
... //constants, types, ports, tasks, modes, asyncs
}
```

Next, constants and types can be declared. Besides the basic types as in Java, TDL supports structures and arrays of constant size. By denoting a name public any importing module is allowed to refer to this name: **public const** NavPeriod = 1220us;

```
public type Vector = struct {
 float x, y, z;
 };
 type FluxBuffer = int[120];
```

The sensor and actuator declarations that follow define the hardware inputs and outputs used by the module. With the uses clause one specifies the name of the external getter or setter function that the E-machine calls to access the hardware:

```
public sensor InSens in uses getInertial;
```

The global output ports come next. A port is updated at the end of the LET of the task that writes it:

```
public output Vector pos;
```

Next the tasks with their inputs are declared. In the uses clause the name of the external function providing the task's functionality is specified. The last four parameters in the example below refer to global output ports:

```
task solveMotion {
    input InSens in; Vector cPos; Vector cVel; Quaternion cAtt;
   uses deadReconing(in, cPos, cVel, cAtt, pos, vel, att, time);
}
```

A mode is a set of activities, i.e. task invocations, actuator updates and mode switches, which are executed periodically with the mode period p. For each activity a frequency f and, optionally, a guard can be specified. For a task invocation the LET of this invocation is p/f unless slot selection is being used for specifying the LET explicitly. In the following mode declaration, the period is set to NavPeriod. Both, the solveMotion and acquireMagHeading tasks are invoked once per period so that the LET of both tasks is NavPeriod. The mode Navigation is declared as start mode which means that the execution of the module starts with this mode.

The names of objects imported from some other module are qualified either by the name of the imported module or by its abbreviation (e.g. K.pos):

```
start mode Navigation [period = NavPeriod] {
   task [freq = 1] solveMotion(in, K.pos, K.vel, K.att);
   task [freq = 1] acquireMagHeading();
}
```

Finally, asynchronous activities can be specified as in the following code fragment. Once the interrupt named iGPS occurs, the task receiveGPS is enqueued for later processing and executed by a background thread. The mapping of the logical interrupt name iGPS to a particular interrupt line is platform specific and part of the TDL Visual Distributor tool:

```
asynchronous {
   [interrupt = iGPS, priority = 2] receiveGPS(INS.time);
}
```

5.3 Complete TDL modules

In our hardware three independent asynchronous timing sources are visible to the software: the processor clock, the sampling events of the inertial sensor, and the sampling events of the DA and AD converters. Choosing the sampling events of the inertial sensor as the time base for the E-machine allows us to solve the equations

of motion and to consider other sensor inputs using Kalman filters[15] synchronously with the inertial data stream.

The module INS processes the inputs of the inertial sensor and of the fluxgate compass. For each new inertial measurement the task solveMotion advances the estimates for the position, the velocity, and the attitude of the vehicle. Quaternions are used for the representation of attitudes.

The excitation of the fluxgate compass is supplied with a continuous data stream by the I/O processor of the ADSP-21262. The data streams from the two sense coils are captured and transferred to buffers in memory by I/O processor. The size of the array type FluxBuffer is made large enough to hold the data acquired during one period of the mode Navigation for both sense coils. A state port containing two buffers, one for capturing and one for processing, is introduced for avoiding any array copy operations. The task-release function exciteFluxGate restarts the data stream to the fluxgate compass and switches between the two buffers at the start of the LET of task acquireMagHeading. By invoking acquireMagHeading with the same frequency as solveMotion the compass is synchronized to the inertial sensor.

The module INS counts the sampling events in the task solveMotion to provide a time base for the other modules. The period of 1220 microseconds for the mode Navigation is the time that passes between two consecutive samples of the inertial sensor.

```
module INS {
  import Kalman as K;

  public const NavPeriod = 1220 us;

  public type Vector = struct {float x, y, z;};
  public type Quaternion = struct {float x0, x1, x2, x3;};
  public type InSens = struct {float aX, aY, aZ, omegaX, omegaY, omegaZ;};
  type FluxBuffer = int[120];
  type FluxDoubleBuffer = struct {byte bufState; FluxBuffer flux1, flux2;}

  public sensor InSens in uses getInertial;

  public output Vector pos; Vector vel; Quaternion att;
  public output long time; Vector mHead;

  task solveMotion {
    input InSens in; Vector cPos; Vector cVel; Quaternion cAtt;
    uses deadReconing(in, cPos, cVel, cAtt, pos, vel, att, time);
  }
  task acquireMagHeading {
    state FluxDoubleBuffer flux;
    uses [release] exciteFluxGate(flux);
    uses integrateFluxGate(flux, mHead);
  }
    start mode Navigation [period = NavPeriod] {
```

```
      task [freq = 1] solveMotion(in, K.pos, K.vel, K.att);
      task [freq = 1] acquireMagHeading();
    }
  }
```

The module GPS receives position and velocity information from a GPS receiver via the LAN interface typically once per second. The LAN interface chip has an internal memory buffer. It activates interrupt iGPS of the signal processor to demand service.

To maintain a timing relationship with the inertial data each dataset from the GPS receiver is time stamped as soon as it is received.

```
module GPS {
  import INS;

  public output INS.Vector pos; INS.Vector vel; long timeStamp;

  public task receiveGPS {
    input long time;
    uses getGPSData(time, pos, vel, timeStamp);
  }

  asynchronous {
    [interrupt = iGPS, priority = 2] receiveGPS(INS.time);
  }
}
```

On power on the module Kalman aligns the estimates for the vehicle's position, velocity, and attitude. Once a good initial fix has been achieved it switches to Filter mode. It then combines the inertial measurement, the GPS position and velocity, and the compass heading into an estimate of the vehicle's position, velocity, and attitude.

```
module Kalman {
  import INS; GPS;

  public output INS.Vector pos; INS.Vector vel;
  public output INS.Quaternion att; long stamp;

  public task align {
    input INS.InSens in; INS.Vector mHead; long time;
    uses doAlign(in, mHead, time, pos, vel, att, stamp);
  }
  public task filter {
    input INS.Vector nPos; INS.Vector nVel; INS.Quaternion nAtt;
    input INS.Vector mHead; long time;
    input INS.Vector gpsPos; INS.Vector gpsVel; long gpsStamp;
    uses doKalmanFilter(nPos, nVel, nAtt, mHead, time,
    gpsPos, gpsVel,gpsStamp, pos, vel, att, stamp);
  }
```

```
    start mode Align [period = INS.NavPeriod] {
      task [freq = 1] align(INS.in, INS.mHead, INS.time);
      mode [freq = 1] if isAligned() then Filter;
    }
    mode Filter [period = INS.NavPeriod] {
      task [freq = 1] filter(INS.pos, INS.vel, INS.att, INS.mHead,
      INS.time, GPS.pos, GPS.vel, GPS.timeStamp);
    }
  }
```

The module NavReporter finally communicates the navigational solutions to the outside world. Whenever a new measurement is available, indicated by a port update on the port Kalman.stamp, it makes it available on the LAN. The asynchronous operation uses the default priority, which is the lowest priority (0). Reading the input ports (K.pos, K.vel etc.) is an atomic operation.

```
  module NavReporter {
    import Kalman as K; INS;

    public task reportNav {
      input INS.Vector pos; INS.Vector vel; INS.Quaternion att; long stamp;
      uses doReporting(pos, vel, att, stamp);
    }

    asynchronous {
      [update = K.stamp] reportNav(K.pos, K.vel, K.att, K.stamp);
    }
  }
```

Figure 6 depicts the dataflow between the modules INS and Kalman. Arrows of the same style indicate measurements that are combined by the Kalman filter into one navigation solution. Note that it takes two sampling periods of the inertial sensor until the data arrives at the output ports of the Kalman filter. For slow moving vehicles like sailing vessels this deems satisfactory. For faster moving vehicles one would combine the two functions solveMotion and doKalmanFilter in one task.

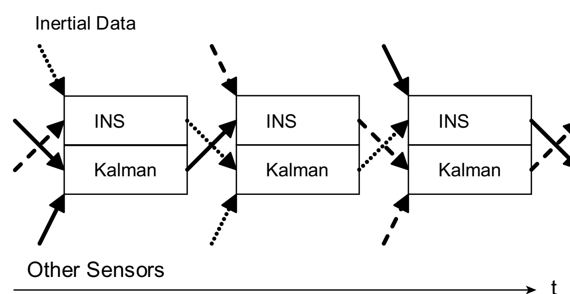

Figure 6. Data flow

6 Related Work

This section gives an overview of other evolution lines of Giotto and compares them with TDL.

6.1 xGiotto

xGiotto[14,2] extends Giotto by (1) adding an implementation language for the body of a task and (2) by adding asynchronous event handling by means of a completely new syntax for expressing time-triggered and event-triggered activities. In contrast to TDL, xGiotto does not provide a component model and it is not targeted at distribution. To our knowledge, there is also no simulation support available for xGiotto.

Adding a new language for the functionality code significantly increases the complexity of xGiotto and its tool chain. It is not clear to us what the advantage of this extension for a real-time system is, given that it is supposed to be compiled into so-called F-code, which is an instruction set for a virtual stack machine that needs to be interpreted at run-time.

The new syntax for specifying time-triggered and event-triggered activities is based on a mechanism called event scoping. An event scope (also called a reaction block) defines the actions to be taken in a given time span which will be terminated after the specified time or by the occurrence of the specified event. xGiotto builds on the assumption that asynchronous events reoccur only after a certain waiting time. However, this assumption is not expressed explicitly but encoded implicitly in an xGiotto program. Event scopes may be nested and, by means of special statements and options, they allow a variety of patterns to be specified for the activities inside an event scope. Besides some exceptions with non-harmonic mode switches, this includes all possibilities of Giotto programs and it adds the execution of LET-based asynchronous task invocations. Event scoping also separates the LET of a task invocation from its execution period and thereby goes beyond Giotto. This is similar to TDL's slot selection approach and, in fact, many xGiotto examples can be transformed to TDL in a straight-forward way, including the xGiotto asynchronous activities, which can be expressed as guarded synchronous task invocations within selected slots. xGiottos's event scoping syntax looks somewhat verbose and requires about 3 times the space of a corresponding Giotto program. In particular, the timing behavior of an asynchronous task invocation is hard to read because it depends on all reaction blocks within the same container scope as the asynchronous task invocation. In contrast, TDL sticks more closely to the lean Giotto syntax for specifying synchronous activities and adds additional constructs for specifying asynchronous activities.

The handling of events differs between TDL and xGiotto. There is no guarantee when and if at all an event is handled in TDL whereas in xGiotto the time until an event is processed is bounded according to the specification of the event scope. Also in contrast to xGiotto, in TDL there is no LET assigned to an asynchronous activity as ports are read and written right before and after its execution. TDL's advantage is that it can also express long-running background tasks for which a reasonable worst case execution time is not available.

xGiotto's compiler is supposed to perform a static check for the absence of race conditions, which occur when a port is updated multiple times at the same logical

time instant. Due to the specific design of xGiotto, a precise check is possible but not in polynomial time. Therefore, only a conservative check is done in the compiler. We do not need such a check at all as we defined appropriate semantics for event-triggered activities and use appropriate synchronization mechanisms for their integration into a time-triggered system. Furthermore, the schedulability analysis is also expensive in xGiotto as it involves solving a two-player safety game. For TDL programs the check is only slightly more complicated (due to slot selection) than for Giotto, for which it can be done by a simple utilization test in polynomial time[18]. Note that asynchronous activities are not taken into account in this test, and need not be taken into account, as TDL provides no guarantees for their execution.

6.2 HTL

The Hierarchical Timing Language (HTL)[3] extends Giotto in the following two aspects. Firstly, HTL adds parallel composition of multiple Giotto programs similar to TDL. Secondly, HTL adds abstract task invocations which may be refined later in a separate program. This is called hierarchical refinement and gives HTL its name. Refinement does not add expressiveness as every refined program is expressible by an equivalent non-refined one. However, it results in a much more compact representation, and simplifies program analysis and schedulability tests.

Although HTL follows the time-triggered programming model of Giotto, it does not specify the LET of a task invocation explicitly. The timing is derived implicitly from data dependencies between task invocations.

Data flow in HTL is based on so-called communicators – typed variables that are accessible only at particular, periodic time instants. They define a program-wide fixed communication matrix. The LET of a task results from the communicator instances it reads from and writes to. This allows for decoupling the LET from the execution period and also provides support for task sequences. TDL uses slot selection to achieve the same goal but provides even more flexibility because in TDL a task may be invoked several times per mode period and each invocation may specify its own LET.

Tasks with the same frequency form a mode in HTL. Within a mode, tasks can communicate directly via ports without the need for communicators. Ports are not bound to a particular timing.

Similar to TDL, HTL uses modules for parallel composition and as units of distribution. However, HTL modules are neither independent nor self-contained and thus not reusable as they depend on globally defined communicators and their timing. In contrast to TDL, HTL lacks the support for asynchronous activities.

HTL is either compiled into Giotto-style E-code or into a newly defined HE-code (hierarchical E-code)[1]. As E-code cannot express hierarchies, HTL programs must be flattened, which results in E-code of exponential size. HE-code supports hierarchical structures by extending the original E-code instruction set and thus trades off code size for runtime performance and a lean E-machine.

There is also a Simulink integration of HTL[7]. In contrast to our approach, the simulation results do not match the HTL description exactly. The deviations are due to introducing additional blocks for breaking algebraic loops. These blocks influence the observable timing. Additionally, the HTL integration in Simulink trades

off accuracy for performance since it requires the sample rate of some blocks to be at least one decimal order of magnitude higher than actually required by the HTL description.

7 Conclusions

TDL goes beyond its predecessor Giotto in a number of aspects. Besides syntactical refinements, it separates the LET from the period of a task invocation, it provides better support for digital controllers, it adds a component model, and it integrates asynchronous activities into a time-triggered system. TDL's component model makes use of Giotto's concept of LET and allows for parallel composition of multiple synchronized Giotto programs without any timing interferences. The component model together with the LET concept also builds the foundation for transparent distribution. It is not observable if a set of TDL modules is executed on a stand-alone node or on a distributed platform. The multi-phase structure of the TDL E-machine and the trisection of the E-code follow naturally from Giotto's concept of an E-machine that executes E-code instructions. Only small instruction set changes have been necessary for efficiently supporting the TDL component model. Also, TDL's approach for integrating asynchronous activities preserves the basic idea of LET for the time-triggered activities but adds background processing while the CPU is idle otherwise. This simplifies many programming tasks that are not time critical and thereby paves the way for applying the time-triggered programming model in real-world embedded software systems. Compared to other extensions of Giotto, the TDL approach preserves the simplicity of the original concepts and is the only one which aims at an industrial strength tool chain. It is fully implemented for multiple platforms, supports simulation in a MATLAB/Simulink environment, and distribution based e.g. on a FlexRay network.

Acknowledgements

We want to thank Gernot Turner for providing us with the hardware for the INS.

References

[1] Ghosal A, Iercan D, Kirsch CM, Henzinger TA, Sangiovanni-Vincentelli AL. Separate Compilation of Hierarchical Real-Time Programs into Linear-Bounded Embedded Machine Code. Online Proc. Workshop on Automatic Program Generation for Embedded Systems (APGES). 2007.

[2] Ghosal A, Henzinger TA, Kirsch CM, Sanvido MAA. Event-driven programming with logical execution times. Hybrid Systems Computation and Control. Lecture Notes in Computer Science 2993. Springer, 2004.

[3] Ghosal A, Henzinger TA, Iercan D, Kirsch CM, Sangiovanni-Vincentelli AL. A Hierarchical Coordination Language for Interacting Real-Time Tasks. Proc. ACM International Conference on Embedded Software (EMSOFT). 2006.

[4] Analog Devices. SHARC Embedded Processor ADSP-21261/ADSP-21262/ADSP-21266, Data Sheet, Rev. E. 2008. Analog Devices, One Technology Way, P.O. Box 9106, Norwood, MA, 02062-9106, USA.

[5] Analog Devices. Six Degrees of Freedom Inertial Sensor ADIS16364, Data Sheet, Rev PrA. 2008. Analog Devices, One Technology Way, P.O. Box 9106, Norwood, MA, 02062-9106, USA.

[6] Autonnic. AR45 Two Axis Magnetometer Component with Floating Core, Data Sheet, 2008. Autonic Research, Woodrolfe Road, Tollesbury, Essex CM9 8SE, Great Britain.

[7] Iercan D, Circiu E. Modeling in simulink temporal behavior of a real-time control application specified in HTL. Journal of Control Engineering and Applied Informatics. 2008,10(4).

[8] Titterton DH, Weston JL. Strapdown Inertial Navigation Technology. 2nd Ed. IEEE Radar, Sonar, Navigation and Avionics Series 17. ISBN 978-0863413582.

[9] Farcas E, Farcas C, Pree W, Templ J. Transparent Distribution of Real-Time Components Based on Logical Execution Time. Proc. LCTES, 2005.

[10] Freescale Semiconductor: Serial Peripheral Interface (SPIV3) Block Description.

[11] Stieglbauer G. Model-based Development of Embedded Control Software with TDL and Simulink [PhD thesis]. University of Salzburg, 2007.

[12] Templ J, Pletzer J, Naderlinger A. Extending TDL with Asynchronous Activities. Technical Report, University of Salzburg, 2008. http://softwareresearch.net/pub/T022.pdf.

[13] Templ J. The Timing Definition Language (TDL) Specification 1.5. Technical Report, University of Salzburg, 2008. http://softwarere search.net/pub/T024.pdf.

[14] Sanvido MA, Ghosal A, Henzinger TA. xGiotto Language Report. Technical Report, University of California, Berkeley, July 2003. http://www.eecs.berkeley.edu/Pubs/TechRpts/2003/CSD-03-1261.pdf.

[15] Kalman RE. A new approach to linear filtering and prediction problems. Journal of the ASME – Journal of Basic Engineering. 1960,82:35–45.

[16] Henzinger TA, Horowitz B, Kirsch CM. Giotto: A Time-Triggered Language for Embedded Programming. Proc. the IEEE. 2003, 91(1).

[17] Henzinger TA, Kirsch CM, Sanvido M, Pree W. From control models to real-time code using Giotto. IEEE Control Systems Magazine, 2003.

[18] THenzinger TA, Kirsch CM, Majumdar R, Matic S. Time-Safety Checking for Embedded Programs. Embedded Software. Lecture Notes in Computer Science 2491. Springer, 2002.

[19] Henzinger TA, Kirsch CM. The Embedded Machine: Predictable, Portable Real-time Code. Proc. the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). 2002:315—326.

[20] The MathWorks. Simulink Online Product Documentation. http://www.mathworks.com/access/helpdesk/help/toolbox/simulink/.