Virtual Execution Environment for Real-Time TDL Components

Claudiu Farcas
Calit2, University of California
San Diego, USA
cfarcas{at}soe.ucsd.edu

Wolfgang Pree
C. Doppler Lab Embedded Software Systems
University of Salzburg, Austria
wolfgang.pree{at}cs.uni-salzburg.at

Abstract

Existing embedded software development methodologies can hardly cope with platform changes such as CPU upgrades, different RTOS or distributed systems. The Timing Definition Language (TDL) enables the development of deterministic real-time components regardless of the deployment platform. The proposed virtual execution environment enables parallel execution and deterministic run-time composition of real-time TDL components with data dependencies. We focus on embedded code generation and the virtual machine that controls the execution of TDL components and their interaction patterns.

1. Introduction

In recent years, advances in hardware devices, networks, and related technologies are hardly matched by correspondent advances in software development. Using traditional real-time software development methodologies it is hard to perform the migration of embedded software from one platform to another, even if the platform change is only for hardware upgrade to a faster processor. The main problems are timing as an accidental consequence of the implementation, conceptual and functional differences between the underlying real-time operating systems, lack of compilers and corresponding run-time environments, and additional testing and verification phases for the behavior of the new system. New development methodologies focus on modeling the application using visual tools such as Matlab/Simulink, supplemented by automatic code generators. However, for improved platform performance most real-time control applications require manual fine-tuning that breaks the correspondence between the model and the implementation, and makes the resulting code hardly portable.

An improved development methodology pioneered by the high-level time-triggered Giotto language [9] makes the timing an explicit part of the real-time software design. The *Timing Definition Language* (TDL) [14] goes further with a component model, streamlined syntax, improved semantics, and full support for distribution. It allows deterministic software composition out of individual components, which run in parallel, may exchange information, and switch execution modes independently. In addition to its virtual machine presented in this paper, it employs a fast on-line scheduler [4] using precompiled schedules and a platform abstraction layer [4] to decouple the components from the underlying platform architecture and enable transparent distribution [6].

In most embedded systems, the interplay between tasks defining the application behavior is tightly integrated into the source code of the application itself, whereas in the case of TDL, the behavior is defined externally and can be easily changed without altering the functionality code. This translates into great benefits for the embedded market as precompiled real-time software libraries can be used for a variety of applications. For instance, in the automotive domain, decoupling the development of individual functionalities for subsystems such as engine control, braking, navigation, from their various integrations options into product lines reduces the development cost and time-to-market of the final product. In addition, late fixes in the application behavior can be easily performed as the developer could simply provide a different set of TDL embedded code without recompiling the application.

In this paper, we focus on the virtual machine of TDL and its accompanying embedded code. We introduce the set of algorithms that perform the compilation and run-time handling of the intra- and inter-component interaction patterns. We go significantly beyond previous work on Giotto by handling parallel run-time composition, data dependencies between components, and support for distribution. In contrast with Giotto's generated E-Code that preempts the application at GCD of all activities, we optimize the TDL E-code for size and minimal number of preemptions, i.e. only at LET instances.

We begin with an overview of TDL and its component model with import relationships. In Section 3, we present the embedded code generation. Section 4 contains the algorithms of the virtual machine. We round out the paper with an evaluation section, related work, and conclusions.

2. Timing Definition Language (TDL)

TDL is a high-level description language for specifying the explicit timing requirements of a time-triggered application, which may be constructed out of several independently developed components. Although, it relies on the same core concept of the Giotto language - the *Logical*

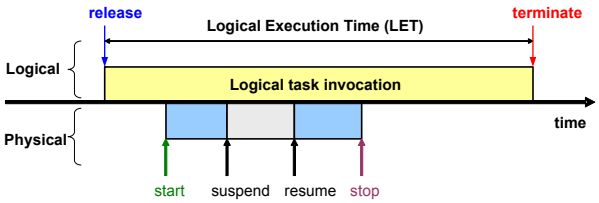


Figure 1. The LET concept

Execution Time (LET) abstraction - it introduces a component model and improves the syntax and semantics, making it more adequate for the development of complex applications. [14, 6, 4] summarize the differences between the two languages. The actual functionality can be implemented in a language such as C/C++, and later linked with the compiled TDL source [6].

2.1. Logical Execution Time (LET)

The LET concept means that the observable temporal behavior of a computational task is independent of its physical execution on a platform. From the logical point of view, depicted in the upper part of the Figure 1, a task reads its inputs at the *release* time, then it runs continuously until its *termination* time, when its computation results are available to the environment and other tasks. From the platform point of view, the task starts at some point in time after it was released, may be preempted by some other tasks or the operating system, and completes before the end of its LET. Tasks are purely computational, do not share internally any resources except at their release and terminate time, and have no internal synchronization points, such as locks, mutexes, etc.

The LET concept introduces a *unit-delay* behavior [9], which may appear as a disadvantage. However, it provides determinism, composition [11], and platform abstraction, which are more relevant for safety-critical systems. It also prevents by design race conditions, deadlocks, and priority inversions. The underlying assumption, which we have to verify [5], is that the run-time system and the scheduling mechanisms used for the physical execution allow each task to complete before its deadline.

2.2. Component Model

As a major improvement over Giotto, TDL introduces a component model that allows the decomposition of complex applications into a hierarchical set of components, i.e. modules. Figure 2 illustrates the syntax of the language.

Modules. The TDL component model relies on the concept of a *module*, which may encapsulate an entire application or parts of a complex application. It provides a separate name-space for all contained TDL entities and acts as a unit of composition and distribution. The TDL components that form an application may work independently, each addressing a specific part of the functionality of the application, or may collaborate to implement a complex behavior. In addition, the component model allows for decomposing existing complex applications into smaller, more manageable parts, each with specific timing and functionality, and provides the means for deterministic component interaction. Developers can reuse existing components to extend the functionality of an application or create new applications.

From a Giotto perspective, a TDL module is conceptually a complete Giotto program, with improved syntax and semantics. However, a TDL module may *import* one or more other modules to build complex applications. This feature introduces data dependencies between modules. The developer may declare a task as *public* and then may access its output port values from any other module importing the task's parent module. For the following algorithms, M represents a module, which may contain a set of types and constants, $\mathsf{Modes}[M]$, $\mathsf{Tasks}[M]$, $\mathsf{Sensors}[M]$, $\mathsf{Actuators}[M]$, and $\mathsf{Guards}[M]$.

Modes. The *mode* of a module is a set of periodically executed activities. At run-time a TDL module may change its mode independently of the others. With Giotto, the mode-changes happen after a time interval depending on the state of the program (number of running tasks) and the program logic, whereas TDL mode changes are regarded as instantaneous. This feature increases the flexibility of the application but introduces constraints on the moments when mode switches may occur in a module - TDL enforces *harmonic* mode switches, i.e. the mode switch must not break the LET of any task invocation within that mode. This apparent restriction improves the determinism and schedulability analysis of the mode-switch, and allows us to perform distributed mode switches in complex applications. A module has a unique *start* mode (initial state), noted m_s.

A mode encompasses a set of activities such as task invocations, actuator updates, with the corresponding frequencies relative to the mode period ω_τ, ω_α, or changes of the application state into a new state, i.e. mode switches η with ω_η. The period of an activity a within a mode m is equal with the *mode period* Π_m divided by the relative frequency ω_a of that activity within the mode.

Tasks. The *task* represents the computational unit of TDL. It has a set of input, state, and output ports, along with an external implementation referred through symbolic linking. A *task invocation* within a mode represents the execution of a task instance within the period of that mode. Each task τ has a number of invocations within a mode m equal with its relative frequency ω_τ within that mode (tasks can have different periods in different modes). TDL regards the tasks as *scheduled* elements with logical execution time [11]. $\mathsf{Tasks}[m] \subset \mathsf{Tasks}[M]$ represents the set of tasks invoked in a mode m of M.

Sensors and Actuators. In TDL, the *sensors* provide information from the environment to the entities of a module, i.e. tasks or guards, while the *actuators* provide the feedback from tasks back to the environment. The underlying assumption is that the external functional implementation of the sensor getters and actuator setters executes in *logical zero time*, i.e. orders of magnitude faster than the smallest task computation. Practical implementations

```
module M1 {
  public const
    c1 = 50; c2 = 200;
    basep = 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);
  }
  public task dec [wcet=1ms] {
    output int o := c2;
    uses decImpl(o);
  }
  start mode f11 [basep] {
    task
      [1] inc(); // LET=basep/1
      [1] dec(); // 10ms period
    actuator
      [1] a1 := inc.o;
      [1] a2 := dec.o;
    mode
      [1] if switch2f12(s, inc.o)
            then f12;
  }
  mode f12 [period=basep] {
    task
      [1] inc(); // 10ms period
      [2] dec(); // LET=5ms
    actuator
      [1] a1 := inc.o; // 10ms
      [2] a2 := dec.o; // 5ms
    mode
      [1] if switch2f11(s, inc.o)
            then f11;
  }
}
```

```
module M2 {
  // get access to M1 and
  // all its public entities
  import M1;

  // initialize actuator a with
  // constant defined in M1
  actuator
    int a := M1.c2 uses setA;

  task sum [wcet=1ms] {
    input
        // input for inc.o of M1
        int i1;
        // input for dec.o of M1
        int i2;

    output
        // initialize task out port o
        // with constant from M1
        int o := M1.c2;

    // external C function o=i1+i2
    uses sumImpl(i1, i2, o);
  }

  // use public constant from M1
  // as reference period
  start mode main [M1.basep] {
    task
      // import task ports from M1
    // Note: this is not RPC
      [1] sum(M1.inc.o, M1.dec.o);

      // update actuator with the
      // sum of inc and dec of M1
    actuator
      [1] a := sum.o; // 10ms
  }
}
```

Figure 2. Modules M1 and M2

may simply read or write to dedicated memory locations or I/O ports, and additional overhead of complex computations may be included into the corresponding task's *worst-case completion time* (wcct).

Guards. The *guards* are external user-defined Boolean functions, which take as input the sensor or task output values, compute in logical zero time, and depending on their result at runtime, they condition the execution of corresponding tasks, actuator updates, or mode switches. Guards$[m]$ represents the set of guards defined in m.

Ports and Drivers. The *ports* are the fundamental interfacing options between the TDL entities of a module, e.g. tasks, sensors, actuators. There can be input, output, or state ports, each with a distinct type, e.g. int, byte, float. Only tasks can have state ports, which we regard as private ports. The output ports of tasks declared as *public* may provide inputs to tasks or actuators in other modules.

The *drivers* as introduced by Giotto are no longer syntactically explicit in TDL. The improved syntax of TDL allows the TDL compiler to automatically perform the type checking between ports and then generate the drivers which transport the port values between the interconnected TDL entities. $d[e] \in$ Drivers$[M]$ represents the driver of a TDL entity e defined in M.

As example consider a simple application from Figure 2, consisting of two modules. M1 has two operating modes f11 and f12, the only difference being that in the second mode, the dec task is invoked twice as fast as inc task. The implementations of both tasks (incImpl and decImpl) have a worst case execution time of 1ms on the test platform. A push button connected as sensor via the external function getS may be used to trigger the mode switches via the guards switch2f11 and switch2f12. The module M2 has only one operational mode, one sum task and no sensors. It imports the module M1, hence, its sum task gets as input the outputs of the two tasks of M1. The actuators provide the tasks' results to the physical environment.

3. Embedded Code

The E-Code concept, a novel way of encapsulating the real-time behavior of an application, was first introduced as a compilation target for the Giotto language [9]. Naturally, we reuse this flexible concept for capturing the timing information of TDL. We consider the E-Code as *reactive* code [11] that we execute at run-time on a virtual machine (E-Machine [10]), assuming that there are sufficient run-time resources, e.g. CPU and memory, in order to meet the real-time behavior requirements expressed by the E-Code and the accompanying functionality code.

The TDL compiler verifies this assumption using the *wcct* of all tasks from all modules in conjunction with the scheduling algorithm and an estimation of the overhead of the TDL runtime on the given platform. This is necessary because the TDL model is compositional by itself, but in practice the overhead of multiple modules' E-Code might add up and break the compositionality. Hence, only if the application set is schedulable on that platform, the TDL compiler generates a table-driven schedule using an algorithm such as EDF, which accounts for the E-code execution overhead and the computed number of preemptions. At run-time, the TDL scheduler monitors the tasks' execution by waking-up at precomputed dispatch or task completion times, and may either throw a LET violation exception or reschedule with a low priority the tasks exceeding their wcct.

3.1. E-Code Instructions for TDL

We define a TDL E-Code instruction as the pair (c, $[args]$), composed of the command c and its set of arguments $[args]$. All E-Code instructions are synchronous, i.e. execute in *logical zero time*. We can express the behavior of a TDL mode m with one or more E-Code instruction blocks. The reunion of all E-Code blocks of all modes of the module M, represents the E-Code of the module M, with the size $\|E[M]\|$.

The CALL(d,[$flag$]) instruction has the functional purpose of encapsulating the interaction between different TDL entities via drivers. Its first argument represents a

driver $d \in \mathsf{Drivers}[M]$, which performs the actual port copying operation required by LET semantics. An optional second argument indicates that the driver corresponds to a task termination event.

RELEASE(τ, $deadline$) prepares an instance of τ for execution; however, it does not imply an immediate *dispatch* operation. The runtime *scheduler* maintains one or more queues of released task instances, and depending on their deadlines and a specific scheduling policy decides at runtime which task instance runs on the CPU [4].

The FUTURE(a, t) instructions plans a later execution of the E-Code block starting at the address $a \in [0 \dots \|E[M]\|)$. t represents the time interval until the E-Machine starts executing the instructions from address a.

The IF(g, a_{true}, a_{false}) instruction introduces branches in the execution of the E-Code, in order to implement the functionality of TDL guard g. The E-machine, depending on the Boolean result of the guard condition $u[g]$, continues the execution from the a_{true} or a_{false} addresses.

The SWITCH(η) instruction triggers a *synchronous* change in the state of the current module, such that the execution continues with the E-Code of the mode m_η. The context in which this instruction appears has to retain the TDL semantics regarding harmonic mode switches.

The JUMP(a) instruction instructs the run-time environment to continue the execution from address a.

The RETURN instruction terminates a block of E-Code. In order to continue the execution of its E-Code instructions, a TDL module has to provide a pair of future address and time offset via a FUTURE instruction before the RETURN instruction. Except for the initialization phase, this instruction stops the execution of the module.

The E-Code instructions have well defined syntax and semantics; however, they are not strictly limited to TDL or its current semantics. As E-Code is synchronous, reactive code, we could also use it to express the behavior of software written in a synchronous language.

3.2. Encoding TDL Semantics into E-Code

Encoding the semantics of a TDL module into E-Code improves portability and paves the way for real-time applications. For example, a low power microcontroller executing the E-Code enables better power management of a more powerful processor executing the user tasks and deep sleeping when idle.

The TDL compiler creates one E-Code file per module using the Algorithm 1. The function emit(c, [args]) produces the E-Code instruction (c, $[args]$) and increments the address counter a. We first generate the initialization E-Code for module M ending with a RETURN instruction. This enables a synchronized startup of an application composed of parallel modules. From a developer perspective, common resources used by multiple modules should only be initialized by a module, and the modules' initialization order should not matter.

For each mode m of the module M we retain the starting address $start_m$ of its E-Code sequence. We later reuse

Algorithm 1. E-Code generation

$a \leftarrow 0$ *// initialization of the address counter*
$\forall p_o \in \mathsf{Ports}_{out}[M]$: **emit**(CALL($d[p_o]$)) *// init. of output ports*
emit(RETURN) *// module initialization is complete*

$\forall m \in \mathsf{Modes}[M]$
  $start_m \leftarrow a$ *// start address of mode m*
  Activ $\leftarrow \emptyset$ *// set of actions indexed by time*
  $\forall \tau \in \mathsf{Tasks}[m] : \forall t \in [0, \omega_\tau),\ t \in \mathbb{N}$ *// release & terminate of τ*
    $\mathsf{Activ}\{t \cdot (\Pi_m/\omega_\tau)\} \leftarrow \mathsf{Activ}\{t \cdot (\Pi_m/\omega_\tau)\} \cup \tau^r$
    $\mathsf{Activ}\{(t+1) \cdot (\Pi_m/\omega_\tau)\} \leftarrow \mathsf{Activ}\{(t+1) \cdot (\Pi_m/\omega_\tau)\} \cup \tau^t$
  $\forall \alpha \in \mathsf{Actuators}[m] : \forall t \in [0, \omega_\alpha),\ t \in \mathbb{N}$ *// actuator updates*
    $\mathsf{Activ}\{t \cdot (\Pi_m/\omega_\alpha)\} \leftarrow \mathsf{Activ}\{t \cdot (\Pi_m/\omega_\alpha)\} \cup \alpha$
  $\forall \eta \in \mathsf{ModeSWs}[m] : \forall t \in (0, \omega_\eta],\ t \in \mathbb{N}$ *// mode switches*
    $\mathsf{Activ}\{t \cdot (\Pi_m/\omega_\eta)\} \leftarrow \mathsf{Activ}\{t \cdot (\Pi_m/\omega_\eta)\} \cup \eta$
  // the set Activ*[ ] contains a timeline of mode m activities*

  for $t \in$ Activ *// at each time instant t do*
    $\forall \tau^t \in \mathsf{Activ}\{t\}$: **emit**(CALL($d[\tau^t]$, true)) *// terminate τ driver*
    $\forall \alpha \in \mathsf{Activ}\{t\}$: *// ACTUATORS*
      if ($\exists g[\alpha]$) *// guards for actuator updates*
        emit(IF($g[\alpha], a+1, a+2$))
      end if
      emit(CALL($d[\alpha^u]$)) *// update actuator driver*
      emit(CALL($d[\alpha]$)) *// set actuator driver*
    $\forall \sigma \in \mathsf{Sensors}[m]$: *// SENSORS read in mode m*
      emit(CALL($d[\sigma]$))
    $\forall \eta \in \mathsf{Activ}\{t\}$: *// MODE SWITCHES*
      emit(IF($g[\eta], a+1, a+3$))
      emit(CALL($d[\eta]$))
      emit(SWITCH(η))
    $\forall \tau^r \in \mathsf{Activ}\{t\}$: *// TASKS*
      if ($\exists g[\tau]$) *// guards for task releases*
        emit(IF($g[\tau], a+1, a+3$))
      end if
      emit(CALL($d[\tau^r]$)) *// update task input ports*
      emit(RELEASE(τ)) *// release task τ*
    if ($\exists t'$ = successor(t), $t' \in$ Activ[ ]) **then**
      $\delta \leftarrow t' - t$ *// time interval until next moment t'*
      emit(FUTURE(δ, a + 2))
      emit(RETURN)
    else *// $t = \Pi_m$ loop to the beginning of the mode m*
      emit(JUMP($start_m$)) || **emit**(SWITCH(m)) *// alternatives*
    end if
  end for *// loop to the next t in the list of activities* Activ

this information to perform mode switches into this mode or to loop back to the beginning of the mode's period. We create first the set of activities Activ[ ], which encapsulates the semantics of the TDL module M. The subset $\mathsf{Activ}\{t\}$ represents the set of activities to perform at time t. For each task τ in the task set $\mathsf{Tasks}[m]$ of the mode m, we add the release and terminate markers τ^r, respectively τ^t to the activities set $\mathsf{Activ}\{t\}$ related to its release and terminate events, i.e. when t is a multiple of its period π_τ. We continue by adding the actuator updates. Assuming the mode switches defined in m are harmonic, we add them to the set $\mathsf{Activ}\{t\}$ (when $t \mid \Pi_m/\omega_\eta$). The resulted set Activ[ ] contains the logical behavior of m.

Using the information encoded in the set Activ[ ], we create a timeline ordered from 0 to Π_m out of the moments $t \in [0, \Pi_m], \exists \mathsf{Activ}\{t\}$. For each t in the timeline, we emit the E-Code sequence that encodes the actions we have to perform at that time instant. We process first the task termination markers by CALL-ing their appropriate drivers to commit their computation results. A driver in the form

$d[\tau^t]$ updates the task output port set $\mathsf{Ports}_{out}[\tau]$ with the results of the user-defined calculation $u[\tau]$. We continue by issuing the CALL instructions for actuators and sensors.

The drivers $d[\alpha^u]$ update the actuator input port values, with the values from the output ports of the driving tasks, and then execute the actuator setter functions, which provide the reaction of the computational system to the environment. We only update the sensor output ports that are required as inputs into new task instances or guard conditions in the mode m. For each mode-switch defined in the mode, we emit a sequence of three E-Code instructions: an IF with the guard function $u[g]$, a CALL to the optional driver $d[\eta]$ that performs the update of output ports in the target mode m', and the actual SWITCH instruction.

For task releases, we optionally emit an IF instruction for the guard of each task, followed by the CALL to the release driver $d[\tau^r]$ that copies the output ports of other tasks or sensors to the input ports of the task τ, and the actual RELEASE instruction passing τ to TDL scheduler.

If we have a successor of the time instance t on the timeline, we compute the time interval δ between t and its successor. We issue a FUTURE instruction planning the execution of the following instructions after the time δ is elapsed at run-time. In the case when we have no successor for t (i.e., we are at the end of the mode $t = \Pi_m$), after all tasks are terminated and we completed all other activities, we loop back to the beginning of the E-Code block of the mode m via a JUMP or SWITCH instruction.

The usage of a SWITCH instruction to complete the mode is useful in distributed systems, or in systems with cyclic dependencies between modules. This final instruction bounds the E-Code sequence of the mode m.

The E-Code is an abstract and compact way of expressing the TDL semantics of a module (see Figure 3). By assigning an integer number to each driver, guard and task, and a binary *opcode* to each command from the E-Code instruction set, we obtain a binary representation[14] of the E-Code program for a TDL module.

```
—— Module M1 ——
// initialization
00: CALL(setA1(a1))
01: CALL(setA2(a2))
02: CALL(getS(s))
03: RETURN()

// Start Mode: f11
04: CALL(read_inputs(inc))
05: RELEASE(inc,10ms)
06: CALL(read_inputs(dec))
07: RELEASE(dec,10ms)
08: FUTURE(10,10ms)
09: RETURN()

10: CALL(terminate(inc),true)
11: CALL(terminate(dec),true)
12: CALL(update(a1))
13: CALL(setA1(a1))
14: CALL(update(a2))
15: CALL(setA2(a2))
16: CALL(getS(s))
17: IF(switch2f12(s),18,20)
18: CALL(switch_driver_f12)
19: SWITCH(f12)
20: SWITCH(f11)

// Mode: f12
21: CALL(read_inputs(inc))
22: RELEASE(inc,10ms)
23: CALL(read_inputs(dec))
24: RELEASE(dec,5ms)
25: FUTURE(27,5ms)
26: RETURN()
27: CALL(terminate(dec),true)
28: CALL(update(a2))
29: CALL(setA2(a2))
30: CALL(read_inputs(dec))
31: RELEASE(dec,5ms)
32: FUTURE(34,5ms)
33: RETURN()

34: CALL(terminate(inc),true)
35: CALL(terminate(dec),true)
36: CALL(update(a1))
37: CALL(setA1(a1))
38: CALL(update(a2))
39: CALL(setA2(a2))
40: CALL(getS(s))
41: IF(switch2f11(s),42,44)
42: CALL(switch_driver_f11)
43: SWITCH(f11)
44: SWITCH(f12)

—— Module M2 ——
00: CALL(setA(a))
01: RETURN()

// Start Mode: main
02: CALL(read_inputs(sum))
03: RELEASE(sum,10ms)
04: FUTURE(6,10ms)
05: RETURN()

06: CALL(terminate(sum),true)
07: CALL(update(a))
08: CALL(setA(a))
09: SWITCH(main)
```

Figure 3. E-code of M1 and M2

4. Virtual Machine

We retain the original naming of the virtual machine in Giotto, i.e. Embedded machine or for short E-Machine [10], which interprets an E-Code block. However, we adapt the E-Machine to the improved semantics of TDL, parallel modules, and the concept of stub modules used for transparent distribution [6].

We introduce the Algorithm 2 for executing an arbitrary E-Code block. It requires that the E-Machine can preempt the execution of TDL tasks in order to execute the E-Code instructions. In the case of an application consisting of a single module, it may be sufficient for the execution of that application on a resource limited system. For a TDL module M, we define its E-Code program configuration at a time t as the tuple $(E[M], i_p, f_a, f_t, \tau_a[t], t)$. The instruction pointer i_p denotes the address from which the E-Machine executes the next E-Code instruction of the module M. The future address f_a represents the next i_p, after the future time offset f_t has elapsed.

The set $\tau_a[t]$ denotes the tasks *active* at the moment t. We call a task τ_a as *active* at a moment t if the task was released at or before t and not completed until t. However, an *active* task τ_a at the moment t, does not mean that it is actually running at that moment, as it could have been preempted by some other task. A task τ, which terminates at a moment t, is automatically removed from $\tau_a[t]$. The function FetchInstruction, returns the next E-Code instruction e from the address i_p, and then increments i_p.

In most applications, the functionality is distributed into several modules that run logically in parallel, effectively sharing the processor and memory resources of the platform. TDL module may have data dependencies through the import relationship. To maintain the application's behavioral determinism, at the end of any tasks's LET from any module, its output ports must be updated before any other entity reads them. In this sense, a simplified version of the Algorithm 2 executes the CALL instruction for the task termination drivers.

For parallel modules, Algorithm 3 executes each module only at the logical time when it needs to run according to the LET semantics. Its complexity is $O(n)$, where n is the number of modules executed in parallel. As a difference to the Algorithm 2, where the logical time t was tightly correlated with the time of a single module, in the Algorithm 3 we have to deal with multiple parallel FU-

Algorithm 2. Execution of an E-Code block

$f_t \leftarrow \infty$ *// reset future time*
$f_a \leftarrow \perp$ *// reset future address*
while ($i_p \neq \perp$) *// until termination*
  $e \leftarrow$ **FetchInstruction**($E[M]$, i_p)
  if (e = CALL(d)) **then**
    $f[d]$ *// executes the driver d functionality code*
  else if (e = RELEASE(τ)) **then**
    $\tau_a[t] \leftarrow \tau_a[t] \cup \tau$ *// add the task τ to the active tasks set $\tau_a[t]$*
  else if (e = FUTURE(a', t')) **then**
    $f_a \leftarrow a'$ *// store the planned E−Code address a'*
    $f_t \leftarrow t'$ *// store the future relative logical time t'*
  else if (e = IF(g, a_{true}, a_{false})) **then**
    if ($u[g]$) **then** *// evaluate the guard condition $u[g]$*
      $i_p \leftarrow a_{true}$ *// jump to the address a_{true}*
    else
      $i_p \leftarrow a_{false}$ *// jump to the address a_{false}*
    end if
  else if (e = SWITCH(η) and $\tau_a[t] = \emptyset$) **then**
    $i_p \leftarrow start_{m'}$ *// only harmonic mode switches*
  else if (e = JUMP(a')) **then**
    $i_p \leftarrow a'$ *// unconditional jump to the address a'*
  else if (e = RETURN) **then**
    $i_p \leftarrow \perp$ *// terminate the execution of this E−Code block*
  end if
end while
// return the planned address f_a and time f_t

Algorithm 3. Parallel execution of modules

// initialization phase with Algorithm 2
for $i \leftarrow 1$ **to** n *// where n is the number of modules*
  execute(E[M_i], 0, 0) *// run the E−Code of M_i*
end for
for $i \leftarrow 1$ **to** n *// prepare modules for execution*
  // initialize f_a of each module with corresponding start mode address
  // assuming $\exists m_s \in m[M_i]$
  $f_a[M_i] \leftarrow start_{m_s}[M_i]$
  $f_t[M_i] \leftarrow 0$ *// simulate a future instruction at time 0*
end for
$t \leftarrow 0$ *// logical time t starts at 0*
$\delta_{min} \leftarrow 0$ *// initialization of time offset until first E−action*
while ($\neg$ shutdown) *// main loop*
  for $i \leftarrow 1$ **to** n
    if ($f_t[M_i] = \delta_{min}$) *// we have to execute this module*
      $i_p[M_i] \leftarrow f_a[M_i]$ *// set E−Code entry point*
      commit(E[M_i], $f_a[M_i]$, $f_t[M_i]$) *// simplified Alg. 2 from $f_a[M_i]$*
    else *// we don't have to run the E−Code of this module yet*
      $f_t[M_i] \leftarrow f_t[M_i] - \delta_{min}$ *// update elapsed*
    end if
  end for *// at this point task outputs are ready*
  for $i \leftarrow 1$ **to** n
    if ($f_t[M_i] = 0$) *// continue the E−Code of this module*
      $i_p[M_i] \leftarrow f_a[M_i]$ *// set E−Code entry point*
      execute(E[M_i],$f_a[M_i]$, $f_t[M_i]$) *// Alg. 2 after last termin. drv.*
    end if
  end for
  // get the minimum time offset till next E−action from all modules
  $\delta_{min} \leftarrow min(f_t[M_i]),\ \forall M_i, i \in [1, n]$

  pass_to_scheduler($\tau_a[\,]$) *// list of all active tasks*
  // we abstract here the scheduling of tasks and the logical time update
  sleep(δ_{min}) *// free CPU for user tasks until next E−action*
  $t \leftarrow t + \delta_{min}$ *// update logical time*
end while

TURE instructions with unrelated timings.

At the moment $t = 0$, we execute for all runable modules their E-Code and update their future time and address parameters (f_t and f_a). We then compute the minimum waiting interval δ_{min} until we have to execute the E-Code of one or more modules. This interval is given by the smallest argument f_t of a FUTURE instruction from all modules. We then pass to the TDL Scheduler the set of active tasks from all modules $\tau_a[\,]$, and idle the virtual machine until $t+\delta_{min}$. We begin a new cycle and execute the modules that have the $f_t = \delta_{min}$ from the previous cycle and update f_t of all other modules with δ_{min}.

For the E-Code from Figure 3, after executing their initialization drivers of both modules, we set the logical time $t = 0$ and the entry points $f_a[M1] = 4$ and $f_a[M2] = 2$ to the beginning of the start modes f11, respectively main.

For the module M1, we run with Algorithm 2 the drivers that update the input ports of the tasks inc and dec and then release the two tasks by placing them in the set of active tasks. We similarly release the sum task and update δ_{min}.

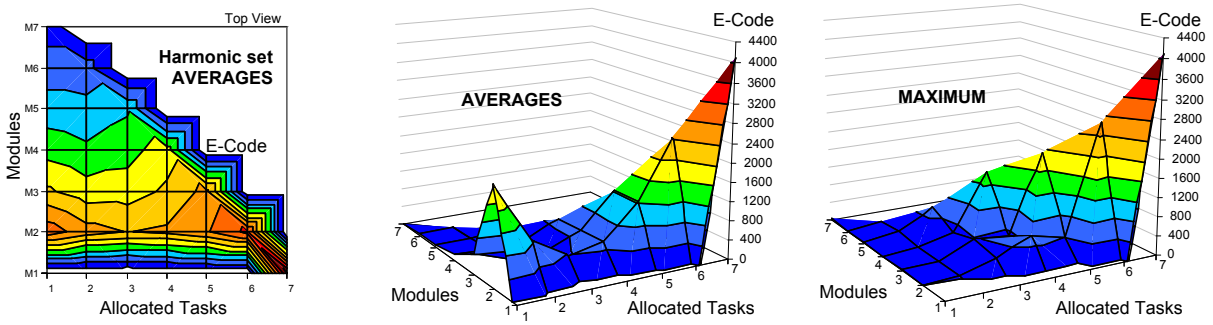
After 10ms, we start a new cycle where modules have updated task outputs. We run the termination drivers of inc and dec, followed by sum. We assume that s has changed its value triggering a mode change in M1 into the new mode f12. E-Machine idles 5ms, whereas TDL scheduler dispatches the active-tasks. At t = 15ms, M2 has no logical action to perform as sum is still logically running. However, dec has completed its LET; thus, we update its output ports. Afterward, we start a new mode f12 cycle, release a new instance of dec, and set a new $f_t[M1]$ = 5ms. During the execution of the E-Code, the E-Machine preempted sum and inc that remained in the list of active tasks.

When t reaches 20ms, all tasks have completed their LETs, and after updating their output ports, we begin a new mode cycle in both modules. The process continues deterministically, as at any logical moment t only the module that has logical activities assigned with that moment gets executed, whereas the others are preempted.

5. Evaluation

The component model of TDL introduces determinism and flexibility in the development of embedded systems. Real-time applications can be built out of individual modules developed independently, and later integrated in various ways without perturbing the behavior of each individual component. An interesting aspect is the allocation of functionality, i.e. tasks, into modules according to the application logic. Depending on the amount of data exchanged between tasks and platform specifics such as network topology or the availability of specialized I/O, the developer may have the option of placing tasks into one or more modules. This decision has a great impact on the resulted E-Code of each module, the CPU utilization and the overall system's performance.

In this sense, the classic Bell polynomials [2] from combinatorial mathematics provide the upper bound on the number of possible allocations of tasks to modules. This number grows exponentially making automated optimal allocations difficult. We analyzed in Figure 4 the case of allocating 7 tasks, with a total number of 877 possible



a) Harmonic task set (T=2,4,8,16,32,64,128ms) b), c) Non-harmonic task set (T=10,20,30,40,50,60,70ms)

Figure 4. E-Code size in relation with the Bell distribution of tasks into modules

allocations into 1-7 modules running on the same CPU. For each of these allocations, the E-Code size of the entire application depends greatly on the task periods. The worst and most uncommon case is when the tasks have prime-number periods, as the size of the E-Code would be proportional with their LCM. The simplest case is when the task set is harmonic. Figure 4 (a) presents this distribution, where allocating all tasks to one module produces the largest E-Code. The most common case is when the tasks have periods derived from a common reference period (in our case 10ms). The two graphs (b) and (c) illustrate the average and maximum E-Code size vs the tasks to module allocation. The leftmost peak on the averages graph comes from the case when the 1st module uses the task with the smallest period, whereas the 2nd module contains all other tasks. The maximum E-Code size is important to determine the necessary ROM space for the application (functionality + TDL runtime + E-Code [+ RTOS]).

We identify two interesting extremes: allocating all tasks to one module, and the opposite of creating one module per task. We analyzed these two cases on an ATmega128 AVR microcontroller for sets of 1-7 tasks using the same task periods from Figure 4 (b). To correlate our results with previous work for Giotto [12], we used the Avrora simulator [15]. We evaluated the impact of the task allocation on the application's memory footprint. We also analyzed the overhead of the TDL runtime, without focusing on the performance of a particular implementation of the user functionality (e.g.,hand-coded or generated). While supporting parallel modules, the measured avg/max E-Machine execution times for TDL are similar with Giotto although TDL offers a component model with improved semantics for structuring complex applications. The results from Figure 5 illustrate a typical tradeoff between memory and CPU utilization: one module requires more memory but less CPU, whereas a module per task has the opposite requirements.

6. Related work

The initial step towards platform independence using the LET concept was laid out with Giotto [9] and E-Code [10]. However, there are significant changes at the level of E-Code and the TDL mode-change semantics.

Giotto. Without a component model and limited support for distribution Giotto can hardly cope with complex applications. TDL addresses these problems and allows data dependencies between components through *import* relationships, synchronized initializations, and improved semantics for the CALL and FUTURE instructions.

The Giotto the mode-change protocol specifies that the mode-switches happen after a time interval depending on the state of the program (number of running tasks) and the program logic, whereas TDL mode changes are regarded as instantaneous. This gives more flexibility to the developer and improves the schedulability analysis. Also, original Giotto E-Code performs mode switches through JUMP instructions hiding the logical control flow and the current execution mode. TDL introduces the SWITCH instruction as it provides additional support for distribution by clearly marking the moment when a mode switch occurs.

For its E-Machine execution, Giotto and its latest derivate HTL [7] has *triggers* to preempt the execution of user tasks at any t multiple of the GCD of all activity periods from the current mode. This is necessary to allow the E-machine to check for task releases, sensor updates or mode switches, but induces a high number of context switches [12]. TDL semantics make the timing of these events known in advance; thus, we can eliminate the queuing of triggers at run-time and skip the unnecessary context switches. Our algorithm compacts the E-Code by reusing periodically executing blocks through the FUTURE and SWITCH instructions. It also enables the parallel execution of multiple modules.

In summary, the key advantages of TDL over Giotto come from its component model, the improved semantics (including the mode-change protocol), the extension of the E-Code, and a leaner and more capable E-Machine.

RT-CORBA. is an optional extension of CORBA for real-time systems. The multithreading capabilities via thread-pools and the priority-mappings between CORBA priority levels and native RTOS priorities are particularly interesting for portable embedded systems. [8] introduces a dynamic scheduling proposal as improvement over the general restriction regarding the fixed-priority scheduling. The popular component models such as CCM, EJB, COM+, and .NET require too many resources for real-time systems. As they mature, more lightweight ap-

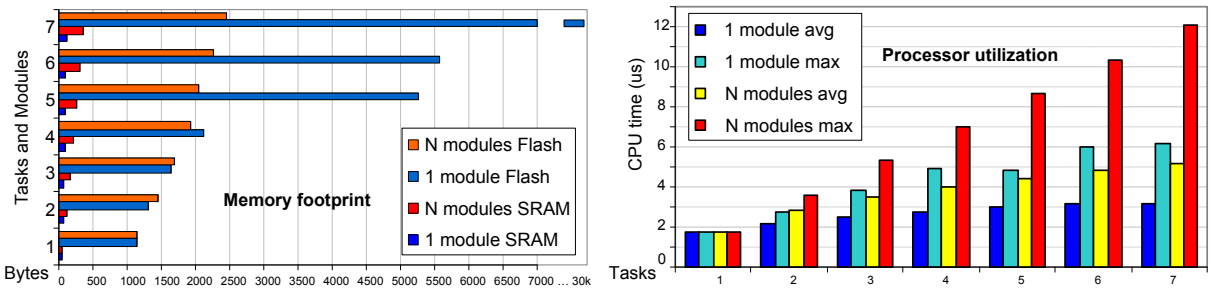


Figure 5. Virtual machine performance with the non-harmonic task set

proaches such as OpenCOM have the potential to become more popular for embedded design. However, RT-CORBA just ensures the end-to-end relations between components, and more research effort is required for proving that it can be extended without perturbing existing functionality of an embedded system.

RT-Java. The Java platform is attractive to embedded systems development for its component architecture, OOP, platform independence, and multithreading. However, Java is not deterministic, does not provide bounded resource usage, and the garbage collector introduces random delays. For real-time systems, the *Real-Time Specification for Java* (RTSJ) [3] offers real-time threads with synchronization, memory management, shared resources with bounded priority inversion, and explicit timing constraints. For an efficient real-time garbage collection [1] proposes a mostly non-copying incremental collector, which achieves low space and time overhead. However, in safety-critical systems, it is a common practice to forbid dynamic allocation. The Ravenscar-Java profile [13] eliminates complex semantics and features with high overheads, such as garbage collection and asynchronous transfer of control. The result is a less flexible programming environment, which pretty much reduces Java capabilities to the level of standard C language.

7. Conclusions

With a clear separation of timing, functionality, execution flow, and platform, we have achieved promising results towards portability of real-time software components. Using a virtual machine for the logical behavior maintains the observable timing and functional behavior of TDL software components on various platforms and enables advanced real-time systems with improved energy efficiency. In addition, the real-time components can be independently developed and later integrated, without changing their behavior or implementation. The determinism, composability, and platform independence properties gained by using TDL, and its runtime environment can reduce the development, testing and integration costs, and speed-up the development of complex real-time systems. As future work, more experimentation and a more thorough evaluation is necessary to provide more evidence about the applicability and superiority of our approach in comparison with other model-driven design approaches.

References

[1] D. F. Bacon, P. Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *POPL '03: Proc. of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, 2003. ACM Press.

[2] E. T. Bell. Partition Polynomials. *Annals of Mathematics*, 29:38–46, 1927.

[3] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull. The real-time specification for Java, 2000.

[4] C. Farcas. *Towards Portable Real-Time Software Components*. PhD thesis, Department of Computer Science, University of Salzburg, Austria, July 2006.

[5] E. Farcas. *Scheduling Multi-Mode Real-Time Distributed Components*. PhD thesis, Department of Computer Science, University of Salzburg, Austria, July 2006.

[6] E. Farcas, C. Farcas, W. Pree, and J. Templ. Transparent distribution of real-time components based on logical execution time. In *Proc. of LCTES*. ACM Press, 2005.

[7] A. Ghosal, A. Sangiovanni-Vincentelli, C. M. Kirsch, T. A. Henzinger, and D. Iercan. A hierarchical coordination language for interacting real-time tasks. In *Proc. of EMSOFT*, pages 132–141. ACM Press, 2006.

[8] O. M. Group. The COmmon Object Request Broker: Architecture and specification, Feb. 2001.

[9] T. Henzinger, B. Horowitz, and C. Kirsch. Giotto: A time-triggered language for embedded programming. In *Proc. of EMSOFT*, LNCS 2211, pages 166–184. Springer, 2001.

[10] T. Henzinger and C. Kirsch. The Embedded Machine: Predictable, portable real-time code. In *Proc. of the PLDI*, pages 315–326. ACM Press, 2002.

[11] C. Kirsch. Principles of real-time programming. In *Proc. International Workshop on Embedded Software (EMSOFT)*, LNCS 2491, pages 61–75. Springer, 2002.

[12] C. Kirsch, M. Sanvido, and T. A. Henzinger. A programmable microkernel for real-time systems. *Proc. of VEE*, 2005.

[13] J. Kwon, A. J. Wellings, and S. King. Ravenscar-Java: A high integrity profile for Real-Time Java. In *Proc. of the Joint ACM Java Grande - ISCOPE Conference*, 2002.

[14] J. Templ. TDL Specification and Report. Technical report, University of Salzburg, Austria, http://www.softwareresearch.net/site/publications/C059.pdf, March 2004.

[15] B. L. Titzer, D. K. Lee, and J. Palsberg. *Avrora: scalable sensor network simulation with precise timing*. IEEE Press, Los Angeles, CA, 2005.