The Logical Execution Time Paradigm

Christoph M. Kirsch and Ana Sokolova

Abstract Since its introduction in 2000 in the time-triggered programming language Giotto, the Logical Execution Time (LET) paradigm has evolved from a highly controversial idea to a well-understood principle of real-time programming. This chapter provides an easy-to-read overview of LET programming languages and runtime systems as well as some LET-inspired models of computation. The presentation is intuitive, by example, citing the relevant literature including more formal treatment of the material for reference.

1 LET Overview

Logical Execution Time (LET) is a real-time programming abstraction that was introduced with the time-triggered programming language Giotto [20, 21, 22]. LET abstracts from the actual execution time of a real-time program. LET determines the time it takes from reading program input to writing program output regardless of the time it takes to execute the program. LET is motivated by the observation that the relevant behavior of real-time programs is determined by when input is read and output is written and not when programs just execute any code.

Before the introduction of LET two other rather different real-time programming abstractions had been around for quite some time that originated from two largely disjoint communities: the Zero Execution Time (ZET) abstraction [31, 35] as the foundation of synchronous reactive programming [18] and the Bounded Execution Time (BET) abstraction [31, 35] as the foundation of real-time scheduling theory [7]. Figure 1 shows the three abstractions.

Christoph M. Kirsch
Department of Computer Sciences, University of Salzburg, Austria, e-mail: ck@cs.uni-salzburg.at

Ana Sokolova
Department of Computer Sciences, University of Salzburg, Austria, e-mail: anas@cs.uni-salzburg.at

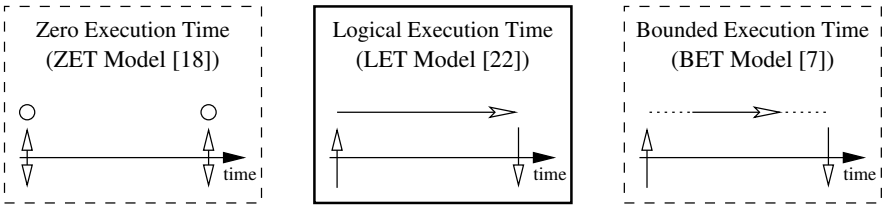

Fig. 1 Fundamental real-time programming abstractions [31, 35]

Similar to the LET abstraction, ZET also abstracts from the actual execution time of a real-time program yet even more than LET. With ZET the execution time of a program including reading input and writing output is assumed to be zero, or equivalently, the execution platform of a program is assumed to be infinitely fast. ZET is the key abstraction of synchronous reactive programming. ZET programs are reactive, i.e., always react to input with some output, and synchronous, i.e., do so in zero time. The execution of ZET programs is correct if the program always writes output before new input becomes available. Establishing correctness typically involves fixed-point analysis since ZET programs written in synchronous reactive programming languages such as Lustre [19] and Esterel [5] may contain cyclic dependencies.

Unlike the ZET and LET abstractions, BET does not abstract execution times but instead bounds them using deadlines. Strictly speaking, BET is therefore a model for temporal constraint programming rather than a programming abstraction. With BET a real-time program has a deadline, which constrains correct program execution to the instances when the program completes execution before or at the deadline. In the BET model, an execution of the program is incorrect if the program does not complete within the deadline, even if the program eventually completes with a functionally correct result. Correct execution of concurrent real-time programs with multiple, possibly different and recurring deadlines requires real-time scheduling. Rate-monotonic (RM) and earliest-deadline-first (EDF) scheduling [36] are prominent examples of scheduling strategies in the BET model.

LET is inspired by both the ZET abstraction and the BET model. LET program execution is correct, i.e., time-safe [21, 22], if the program reads input, in zero time, then executes, and finally writes output, again in zero time, exactly when the LET has elapsed since reading input. The end of the LET thus corresponds to a deadline in the BET model but only for program execution without reading input and writing output. In other words, if the program completes execution before the deadline, writing output is delayed until the deadline, i.e., until the LET has elapsed. The deadline is therefore not only an upper bound, like in the BET model, but also a lower bound, at least, logically. In the LET model, using a faster machine does therefore not result in (logically) faster program execution but only in decreased machine utilization, which makes room for more concurrency. Conversely, more concurrency on the same machine has no effect on input and output times as long as the machine is sufficiently fast to accommodate all deadlines.

1.1 Giotto

LET programs may be event- or time-triggered (or both) in the sense that the time when input is read and thus when the LET begins is determined by the occurrence of an event or the progress of time, respectively. Giotto programs, for example, are time-triggered LET programs while xGiotto [16] programs may be both event- and time-triggered LET programs, and even the LET itself in xGiotto programs may be determined by events rather than time.

There are two key results on Giotto programs. Checking time safety of Giotto programs is easy [26] and time-safe Giotto programs are time-deterministic [21, 22], i.e., the relevant input and output (I/O) behavior of time-safe Giotto programs does not change across varying hardware platforms and software workloads. Time-safe execution of Giotto programs requires real-time scheduling, similar to BET programs, but no fixed-point analysis, unlike ZET programs, since reading input and writing output is cycle-free. Note that all real-time programming paradigms have yet in common that they require some form of hardware-dependent, worst-case execution time (WCET) analysis [14] for establishing correctness.

Giotto programs may specify multiple modes and mode switching logic but can only be in one mode at a time during execution. Checking time safety of Giotto programs is easy, i.e., fast, because time safety of the individual modes of a Giotto program implies time safety of the whole program regardless of mode switching [26]. The converse is not true since the mode switching logic may prevent modes that are not time-safe from ever being executed. Checking time safety of a mode is linear in the size of its description using a standard, utilization-based schedulability test based on EDF.

LET programs may be distributed across multiple machines, just like ZET and BET programs. However, the key difference is that LET, unlike ZET and BET, is a natural, temporally robust placeholder not just for program execution but also for program communication [29]. Distributing LET programs is thus easy, in particular, onto architectures that provide time synchronization such as the Time-Triggered Architecture (TTA) [34]. Even more importantly, the relevant behavior of a time-safe, distributed version of a time-safe, non-distributed LET program is equivalent to the relevant behavior of the non-distributed program.

However, Giotto has a scalability issue in the sense that each mode of a Giotto program needs to specify the whole behavior of the program while being in that mode. For example, if a Giotto programmer would like to maintain some behavior of a mode when switching to another mode, both modes need to specify their common behavior. In other words, a Giotto program is a flat, non-hierarchical description of a real-time program. Giotto programs may therefore become large for non-trivial, in particular distributed applications. Giotto programming by example is discussed in more detail in Section 2.

1.2 Hierarchical Timing Language

The Hierarchical Timing Language (HTL) [15, 27] is a Giotto successor that aims at improving succinctness while keeping time safety checking easy. Modes in HTL are partial specifications of LET programs that may be hierarchical in the sense that some modes may be abstract placeholders reserving computation time for refining, more concrete modes that specify actual program behavior. Intuitively, a concrete mode refines an abstract mode if the LETs in the concrete mode start later and end earlier than the LETs in the abstract mode. The key result is that time safety of an abstract HTL program implies time safety of any concrete HTL program that refines the abstract program [15]. The converse is again not true. There may be time-safe concrete HTL programs that refine abstract HTL programs that are not time-safe. Note that checking refinement is easier than checking time safety [27].

Refinement in HTL enables modular real-time programming [27]. A time-safe HTL program may be changed locally without the need for re-checking time safety globally, with the exception of the top level of the program. Modifications to the most abstract portion of an HTL program may require re-checking time safety for the whole program. Since correctness of modifications below top level can be checked fast, i.e., independently of the size of the whole program, HTL programs may even be modified at runtime through a process called runtime patching while preserving the real-time behavior of the unmodified parts [32]. Runtime patching is a semantically robust method for introducing flexibility into real-time programming. An HTL programming example is discussed in more detail in Section 2.

1.3 Model-driven Development

LET programming is part of a larger, model-driven development (MDD) methodology [30] depicted in Figure 2. LET programs may be modeled and validated in a simulation environment such as Simulink [10] and then translated to executable code. Here, the key idea is that LET model, program, and code are equivalent with respect to their relevant real-time behavior [30]. Changes on one level have therefore a well-understood effect on the other levels enabling compositional implementation and validation with LET-based toolkits such as FTOS [6] and TDL [13]. LET-oriented runtime systems are discussed next.

1.4 The Embedded Machine

LET code generators may target general purpose programming languages such as C or virtual machines that have been specifically designed for LET semantics such as the Embedded Machine [23, 25], or E machine, for short, which is an interpreter for E code. Similar to Giotto and HTL programs, time-safe E code is time-deterministic.

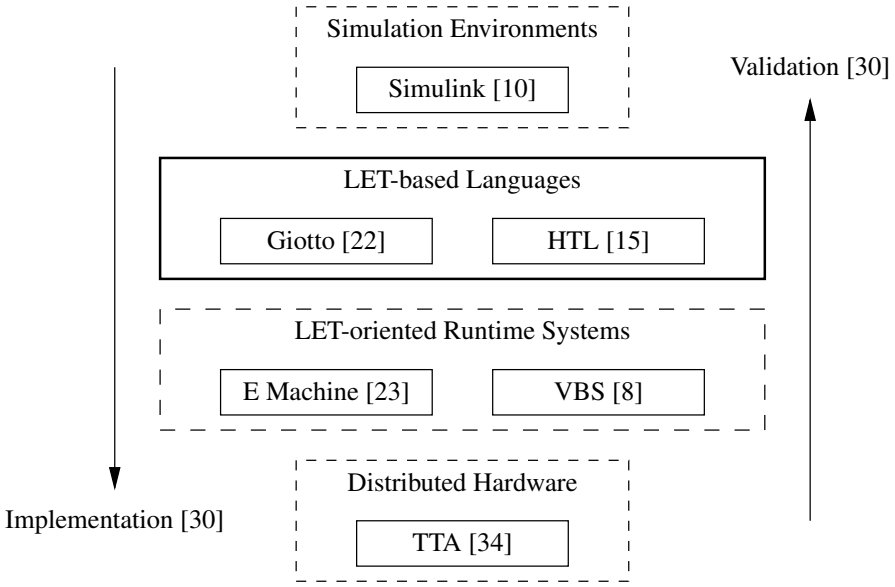

Fig. 2 Context of LET-based languages and LET-oriented runtime systems

We also say that time-safe E code is time-portable [2, 3] to any platform for which an E machine implementation exists. E code may also be executed on distributed systems such as TTA by running an E machine on each host [29].

Checking time safety of arbitrary E code may be difficult but remains easy for non-trivial classes of E code [24] such as E code generated from Giotto and HTL [26]. Executing E code requires an E machine as well as an EDF-scheduler. However, scheduling decisions may also be computed at compile time and represented by Schedule-Carrying Code (SCC) [28], which is E code extended by specific scheduling instructions. SCC is an executable witness of time safety. Checking whether SCC is time-safe is in general easier than generating SCC. An E machine extended for SCC support may therefore verify time safety prior to execution and does not require an EDF-scheduler [33].

E code generated from a Giotto program requires pseudo-polynomial space in the size of the program, i.e., the numerically represented program periods [26]. E code execution time is linear in the size of the program in this case. E code generated from an HTL program may even be exponentially larger than the program regardless of the periods because any hierarchy in the program is flattened prior to code generation. Flattening can be avoided by targeting a hierarchical version of the E machine [17]. The resulting E code requires then again only pseudo-polynomial space in the size of the program and can be executed in linear time. The E machine is discussed in more detail in Section 2.

1.5 Variable-Bandwidth Servers

Giotto and HTL are languages in which LET programs are constructed around the notion of modes. However, a LET program may also be understood as a specification of a set of concurrent processes where each process performs I/O as fast as possible, i.e., logically in zero time, and then computes as predictable as possible, i.e., logically for a given amount of time, and then performs I/O again and so on. After performing I/O the process may decide, based on the previous compution and the new input, what to compute next, which is essentially another way of switching modes. The logical execution time of each computation phase may also change as long as it is determined by the process itself.

Variable-Bandwidth Servers (VBS) [8] may provide a natural scheduling platform for executing concurrent processes specified by a LET program. A VBS process is a sequence of actions. Each action is sequential code, which is executed in temporal isolation of any other process in the system, i.e., lower and upper bounds on the time to execute the action are solely determined by the invoking process. The bounds may change from one action to the next to accommodate different types of actions such as latency-oriented as-fast-as-possible I/O actions and throughput-oriented yet as-predictable-as-possible computation actions. The bounds can be seen as a generalization of LET from an exact logical value of duration to a realistic interval of permitted values. Running LET programs on VBS remains to be a subject of future work.

1.6 Models of Computation

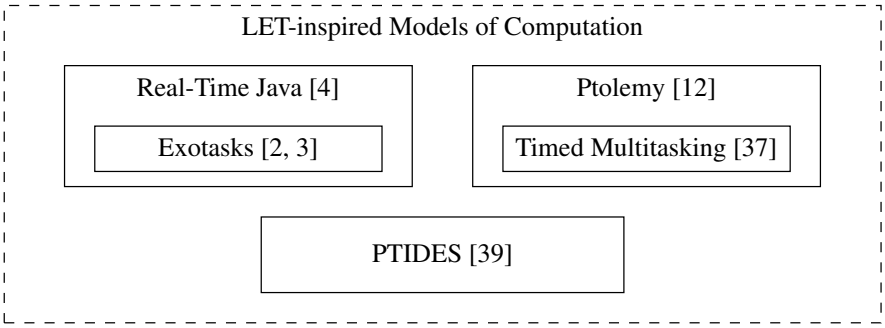

Fig. 3 LET-inspired models of computation

Figure 3 shows a selection of LET-inspired models of computation. Exotasks [2, 3] implement a real-time scheduling framework in Java using the real-time garbage collector Metronome [4] for real-time performance. The framework has been in-

stantiated to provide LET semantics in Java. A more general version called Flexotasks [1] provides even more freedom to implement and integrate temporal and spatial isolation policies.

The key motivation of LET programming is to develop systems that maintain their relevant real-time behavior across changing hardware platforms and software workloads. However, requiring all program parts to follow the LET regime may be unnecessarily restrictive. For example, program parts that are independent of I/O behavior may be scheduled in a more flexible manner without loosing guarantees on relevant behavior [11]. Timed multitasking [37] in Ptolemy [12] provides LET guarantees relative to the occurrences of events, similar to the previously mentioned xGiotto [16]. However, event scoping in xGiotto enables structured specification of event handling policies such as implicit assumptions on interarrival times to facilitate time safety analyses. Discrete-event models in PTIDES [39] provide another model of computation that enforces LET but only at communication boundaries, i.e., sensing, actuating, and other relevant I/O is performed at time instants that are independent of execution order and speed.

2 LET Programming by Examples

2.1 Giotto

A Giotto program consists of a functionality part and a timing part. The functionality part contains port, driver, and task declarations, which interface the Giotto program to a functionality implementation, written in C, for example. For the examples below, we show the timing part of a Giotto program.

Single-mode Giotto program

As a first example, we present a highly simplified version of the control program for a model helicopter such as the JAviator [9]. Consider the helicopter in hover mode m. There are two tasks, both given in native code, possibly autogenerated from Matlab/Simulink models [30]: the control task t_1, and the navigation task t_2. The navigation task processes GPS input every 10 ms and provides the processed data to the control task. The control task reads additional sensor data (not modeled here), computes a control law, and writes the result to actuators (reduced here to a single port). The control task is executed every 20 ms. The data communication requires three drivers: a sensor driver d_s, which provides the GPS data to the navigation task; a connection driver d_i, which provides the result of the navigation task to the control task; and an actuator driver d_a, which loads the result of the control task into the actuator. The drivers may process the data in simple ways (such as type conversion), as long as their WCETs are negligible. In general, since E code

execution is synchronous and can thus not be interrupted by other E code, we say that the WCET of an E code block (i.e., the sum of the WCETs of all drivers as well as all E code instructions called in that block) is negligible if it is shorter than the minimal time between any two events that can trigger the execution of E code. In the case of the helicopter software, the WCETs of all E code blocks are at least one order of magnitude shorter than 10 ms, which is the time between two consecutive invocations of E code in this example.

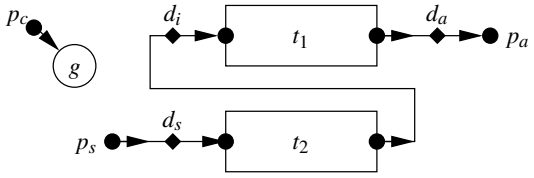

Fig. 4 The dataflow of the example with two periodic tasks [25]

There are two environment ports, namely, a clock p_c and the GPS sensor p_s; two task ports, one for the result of each task; and three driver ports —the destinations of the three drivers— including the actuator p_a. Figure 4 shows the dataflow of the program: we denote ports by bullets, tasks by rectangles, drivers by diamonds, and triggers by circles. It therefore presents an abstract functional implementation of the program. Here is a Giotto description of the program timing:

```
mode m() period 20 {
  actfreq 1 do pa(da);
  taskfreq 1 do t1(di);
  taskfreq 2 do t2(ds);
}
```

The "`actfreq` 1" statement causes the actuator to be updated once every 20 ms; the "`taskfreq` 2" statement causes the navigation task to be invoked twice every 20 ms; etc. Note that the LET of each task is specified by the ratio of the mode period over the task frequency (20 ms for t_1 and 10ms for t_2). Here is a simplified version of the E code generated by the Giotto compiler:

```
a1: call(da)          a2: call(ds)
    call(ds)              release(t2)
    call(di)              future(g,a1)
    release(t1)
    release(t2)
    future(g,a2)
```

The E code consists of two blocks. The block at address a_1 is executed at the beginning of a period, say, at 0 ms: it calls the three drivers, which provide data for

the tasks and the actuator, then hands the two tasks to the scheduler, and finally activates a trigger g with address a_2. When the block finishes, the trigger queue of the E machine contains the trigger g bound to address a_2, and the ready queue of the scheduler contains two tasks, t_1 and t_2. Now the E machine relinquishes control, only to wake up with the next input event that causes the trigger g to evaluate to true. In the meantime, the scheduler takes over and assigns CPU time to the tasks in the ready queue according to some scheduling scheme. When a task completes, the scheduler removes it from the ready queue.

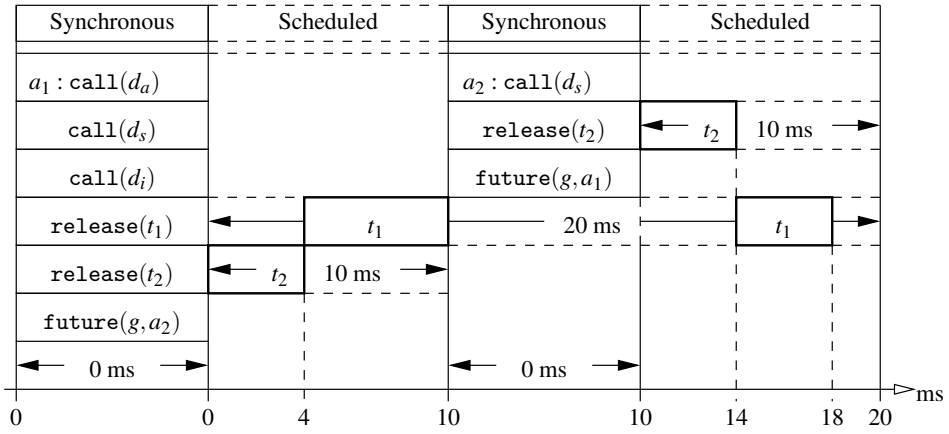

Fig. 5 Earliest-deadline-first (EDF) schedule of the two-task Giotto program (adapted from [25])

There are two kinds of input events, one for each environment port: clock ticks, and changes in the value of the sensor p_s. The implementation of the trigger g is part of the functional code. In the example, the trigger g: $p_c' = p_c + 10$ specifies that the E code at address a_2 will be executed after 10 clock ticks. Logically, the E machine wakes up at every input event to evaluate the trigger, finds it to be false, until at 10 ms, the trigger is true. An efficient implementation, of course, wakes up the E machine only when necessary, in this case at 10 ms. The trigger g is now removed from the trigger queue and the associated a_2 block is executed. It calls the sensor driver, which updates a port read by task t_2. There are two possible scenarios: the earlier invocation of task t_2 may already have completed, and is therefore no longer in the ready queue when the a_2 block is executed. In this case, the E code proceeds to put another invocation of t_2 into the ready queue, and to trigger the a_1 block in another 10 ms, at 20 ms. In this way, the entire process repeats every 20 ms. The other scenario at 10 ms has the earlier invocation of task t_2 still incomplete, i.e., in the ready queue. In this case, the attempt by the sensor driver to overwrite a port read by t_2 causes a runtime exception, called time-safety violation. At 20 ms, the end of the mode period, when ports read by both tasks t_1 and t_2 are updated and ports written by both t_1 and t_2 are read (via the drivers), a time-safety violation occurs unless both tasks have completed, i.e., the ready queue must be empty. In

other words, an execution of the program is time-safe if the scheduler ensures the following: (1) each invocation of task t_1 at $20n$ ms, for $n \geq 0$, completes by $20n+20$ ms; (2) each invocation of task t_2 at $20n$ ms completes by $20n+10$ ms; and (3) each invocation of task t_2 at $20n+10$ ms completes by $20n+20$ ms. Therefore, a necessary requirement for time safety is $\delta_1 + 2\delta_2 < 20$, where δ_1 is the WCET of task t_1, and δ_2 is the WCET of t_2. If this requirement is satisfied, then a scheduler that gives priority to t_2 over t_1 guarantees time safety. Figure 5 presents a time-safe EDF schedule of the two-task Giotto example, with $\delta_1 = 10$ ms and $\delta_2 = 4$ ms.

The E code implements the Giotto program correctly only if it is time-safe: during a time-safe execution, the navigation task is executed every 10 ms, the control task every 20 ms, and the dataflow follows Figure 4. Thus the Giotto compiler needs to ensure time safety when producing E code. In order to ensure this, the compiler needs to know the WCETs of all tasks and drivers (cf., for example, [14]), as well as the scheduling scheme used by the operating system. With this information, time safety for E code produced from Giotto can be checked [26]. However, for arbitrary E code and platforms, WCET analysis and time-safety checking may be difficult, and the programmer may have to rely on runtime exception handling, see [25] for more details. At the other extreme, if the compiler is given control of the system scheduler, it may synthesize a scheduling scheme that ensures time safety [28].

The time-safe executions of the E code example have an important property: assuming the two tasks compute deterministic results, for given sensor values that are read at the input port p_s at times 0, 10, 20, ... ms, the actuator values that are written at the output port p_a at times 0, 20, 40, ... ms are determined, i.e., independent of the scheduling scheme. This is a consequence of the LET paradigm, because each invocation of the control task t_1 at $20n$ ms operates on an argument provided by the invocation of the navigation task t_2 at $20n-10$ ms, whether or not the subsequent invocation of t_2, at $20n$ ms, has completed before the control task obtains the CPU. Time safety, therefore, ensures not only deterministic output timing, but also deterministic output values; it guarantees predictable, reproducible real-time code.

Multiple-mode Giotto program

The helicopter may change mode, say, from hover to descend, and in doing so, apply a different filter. In this case, the navigation task t_2 needs to be replaced by another task t_2'. We show how to implement different modes of operation using Giotto and the generated E code with control-flow instructions. Note that E code can also be changed dynamically, at runtime, still guaranteeing determinism if no time-safety violations occur. This capability enables the real-time programming of embedded devices that upload code on demand, of code that migrates between hosts, and of code patches [25].

Consider the following timing part of a Giotto program with two modes, m_a (representing the helicopter in hover mode) and m_b (descend mode):

```
start m_a {
  mode m_a() period 20 {
    actfreq 1 do p_a(d_a);
    exitfreq 2 do m_b(c_b);
    taskfreq 1 do t_1(d_i);
    taskfreq 2 do t_2(d_s);
  }
  mode m_b() period 20 {
    actfreq 1 do p_a(d_a);
    exitfreq 2 do m_a(c_a);
    taskfreq 1 do t_1(d_i);
    taskfreq 2 do t'_2(d_s);
  }
}
```

The program begins by executing mode m_a, which is equivalent to the (single) mode m of the Giotto program from the previous subsection except for the mode switch to mode m_b. A mode switch in Giotto has a frequency that determines at which rate an exit condition is evaluated. The exit condition c_b of mode m_a is evaluated once every 10 ms (the ratio of the period over the exit frequency). If c_b evaluates to true, then the program switches to mode m_b, which is similar to mode m_a except that task t_2' replaces task t_2. Task t_2' applies a different filter on the same ports as t_2. The mode switch back to m_a evaluates the exit condition c_a also once every 10 ms.

This example consists of two modes with equal periods. Programs with multiple nodes of different periods are also possible in Giotto but mode switching is restricted at the end of the node period, i.e., only exit frequency of 1 is allowed.

In order to express mode switching in E code, we use a conditional branch instruction $\texttt{if}(c,a)$. The first argument c is a condition, which is a predicate on some ports. The second argument a is an E code address. The $\texttt{if}(c,a)$ instruction evaluates the condition c synchronously (i.e., in logical zero time), similar to driver calls, and then either jumps to the E code at address a (if c evaluates to true), or proceeds to the next instruction (if c evaluates to false). Here is the E code that implements the above Giotto program:

a_1: $\texttt{call}(d_a)$	a_3: $\texttt{call}(d_a)$
$\texttt{if}(c_b, a_3')$	$\texttt{if}(c_a, a_1')$
a_1': $\texttt{call}(d_s)$	a_3': $\texttt{call}(d_s)$
$\texttt{call}(d_i)$	$\texttt{call}(d_i)$
$\texttt{release}(t_1)$	$\texttt{release}(t_1)$
$\texttt{release}(t_2)$	$\texttt{release}(t_2')$
$\texttt{future}(g, a_2)$	$\texttt{future}(g, a_4)$
a_2: $\texttt{if}(c_b, a_4')$	a_4: $\texttt{if}(c_a, a_2')$
a_2': $\texttt{call}(d_s)$	a_4': $\texttt{call}(d_s)$
$\texttt{release}(t_2)$	$\texttt{release}(t_2')$
$\texttt{future}(g, a_1)$	$\texttt{future}(g, a_3)$

The two E code blocks in the left column implement mode m_a; the two blocks on the right implement m_b. Just like in the single-mode example, the code is free of deadlines and exception handlers for the three tasks, see [25] for more details on exception handlers. Note that, no matter which conditional branches are taken, the execution of any block terminates within a finite number of E code instructions, i.e., the code corresponding to a mode is finite and therefore the execution of each mode instance is finite.

Generating E code, as in the above examples (with additional deadlines and exception handlers) is the result of the first, platform-independent phase of the Giotto compiler. The second, platform-dependent phase of the Giotto compiler performs a time-safety check for the generated E code and a given platform. For single-CPU platforms with WCET information and an EDF-based scheduling scheme, and for the simple code generation strategy illustrated in the example, the time-safety check is straightforward [26]. For distributed platforms, complex scheduling schemes, or complex code generation strategies, this, of course, may not be the case. The code generation strategy has to find the right balance between E code and E machine annotations, see [25] for details. An extreme choice is to generate E code that at all times maintains a singleton task set, which makes the scheduler's job trivial but E code generation difficult. The other extreme is to release tasks as early as possible, with precedence annotations that allow the scheduler to order task execution correctly. This moves all control over the timing of software events from the code generator to the scheduler. In other words, the compiler faces a trade-off between static (E machine) scheduling and dynamic (RTOS) scheduling. The strategy used in the examples and implemented in the Giotto compiler chooses a compromise that suggests itself for control applications. The generated code releases tasks and imposes deadlines according to the "logical semantics" of the Giotto source. To achieve controller stability and maximal performance, it is often necessary to minimize the jitter on sensor readings and actuator updates. This is accomplished by generating separate, time-triggered blocks of E code for calling drivers that interact with the physical environment. In this way, the time-sensitive parts of a program are executed separately [38], and for these parts, platform time is statically matched, at the E code level, to environment time as closely as possible. On the other hand, for the time-insensitive parts of a program, the scheduler is given maximal flexibility.

2.2 Hierarchical Timing Language

In this section we present HTL on one example, the multi-mode control program for a model helicopter of Section 2.1. In Giotto, the control task t_1 is part of both modes, since Giotto programs are flat. In contrast to that HTL, as the name suggests, allows for hierarchical models.

An HTL program is built out of four building blocks: program, module, mode, and task. A program is a set of concurrently running modules. A module consists of a set of modes and some mode-switching logic between them. Like in Giotto,

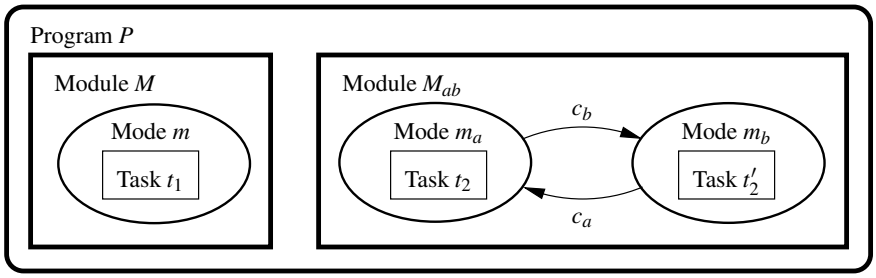

Fig. 6 HTL program for the multi-mode helicopter control example

each mode has a period and contains tasks. Unlike in Giotto, the periods of all tasks in a mode are equal to the mode period. Also different from Giotto, some of the tasks may be abstract tasks, schedulability placeholders for concrete tasks that may refine them. If a mode contains abstract tasks, then it also specifies a refinement program which refines the abstract tasks. Moreover, another difference to Giotto is that mode switching can only happen at the end of a mode (at a period instance). This is not a restriction, since the otherwise richer structure of HTL provides the same expressiveness, as we will see in the example of the multi-mode helicopter control program. The HTL program for our example has two modules: the module M contains a single mode with period 20 ms containing the concrete control task t_1; the module M_{ab} has two modes each with period 10 ms containing a single concrete task (t_2 and t_2', respectively). The program does not involve refinement since there are no abstract modes and tasks. Figure 6 depicts a graphical representation of the HTL program.

In HTL, input is read from and output is written to so-called communicators which are periodic global variables. A value can be read from or written to a communicator at period instances. Communicators have periods that divide the periods of tasks using them. LET in HTL is therefore specified by the time interval between the latest communicator period instance that a task reads, and the earliest communicator period instance that a task writes to (for distributed programs this needs to be slightly adjusted for modularity [27]). Tasks are linked to communicator period instances via ports. In the example, each of the three drivers corresponds to a communicator and the driver ports are the ports used for the link. Due to the simplicity of the example, there is no need to mention the ports in the code for the HTL program of Figure 6. Here is a simplified version of the HTL (pseudo) code:

```
program P {
  comm d_a(20), d_s(10), d_i(10)
  module M {
    mode m(20) {
      task t_1 in (d_i,1) out (d_a,2)
    }
  }
  module M_ab {
    start m_a
    if m_a ∧ c_b then m_b
    if m_b ∧ c_a then m_a
    mode m_a(10) {
      task t_2 in (d_s,1) out (d_i,2)
    }
    mode m_b(10) {
      task t'_2 in (d_s,1) out (d_i,2)
    }
  }
}
```

The mode switching rules are expressed with `if-then` statements. An instruction of the form `in` (d,i) within a task instruction specifies that the task reads from the i-th period instance of communicator d within the task period. In particular, $i = 1$ corresponds to the beginning of the task period. Similarly, an instruction `out` (d,i) specifies a communicator and its period instance when output is written. HTL also allows for specifying task precedences within a mode since an input port of a task may be linked to an output port of a preceding task, but our example does not illustrate this (as there are not even multiple tasks per mode).

The example program does not involve refinement so far. Therefore, for time safety one needs to check schedulability of all possible combinations of active modes in a program, which in this case amounts to two combinations: (1) mode m and mode m_a, and (2) mode m and mode m_b. Since both m_a and m_b have the same timing (apart from the maybe different WCETs) and I/O behavior, they can be seen as refining a single abstract mode m_A, as in the program presented in Figure 7.

The program still consists of two modules, the module M as before, and the module M_A containing a single abstract mode m_A with period 10 ms and a single abstract task t_A (with input, output, and LET equal to the ones of t_2 and t'_2 and WCET equal to the maximum of the WCETs of the two concrete tasks). The abstract mode has an associated refinement program P_R with a single module M_{ab} containing two modes m_a and m_b as it was the case with the original module M_{ab}. Both tasks, t_2 and t'_2 refine the abstract placeholder task t_A. Refining tasks need to have same or more LET, same or less WCET, and same or weaker task precedences [15, 27]. Now time safety is guaranteed if the tasks of m and m_A are schedulable, since m_a and m_b both refine the abstract mode m_A. Here is a simplified version of the HTL (pseudo) code for the example including refinement:

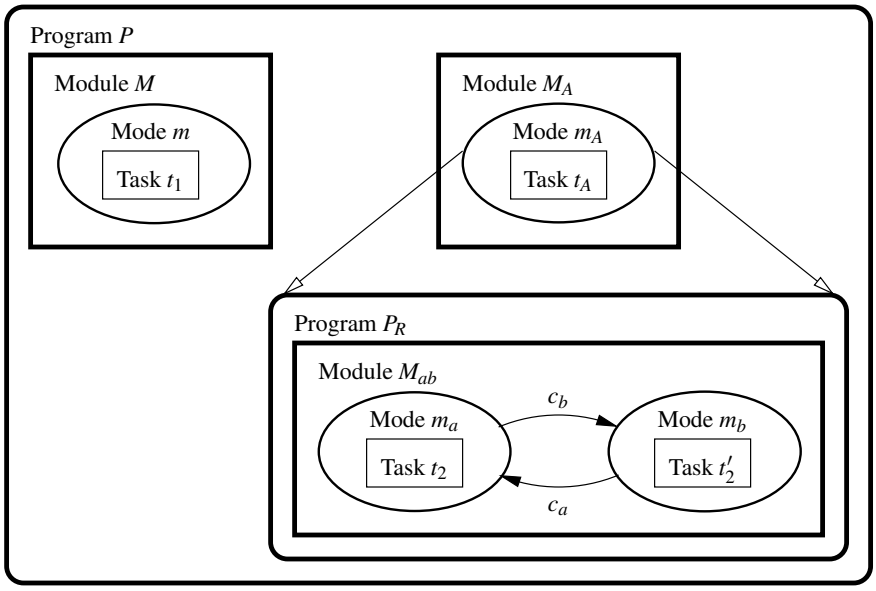

Fig. 7 HTL program with refinement for the multi-mode helicopter control example

```
program P {
  comm d_a(20), d_s(10), d_i(10)
  module M {
    mode m(20) {
      task t_1 in (d_i,1) out (d_a,2)
    }
  }
  module M_A {
    mode m_A(10) {
      abstract task t_A in (d_s,1) out (d_i,2)
      refinement program P_R {
        module M_2 {
          start m_a
          if m_a ∧ c_b then m_b
          if m_b ∧ c_a then m_a
          mode m_a(10) {
            task t_2 in (d_s,1) out (d_i,2)
          }
          mode m_b(10) {
            task t_2' in (d_s,1) out (d_i,2)
          }
        }
      }
    }
  }
}
```

For more details on HTL and its modular properties we refer the interested reader to [27]. E code can be generated for HTL programs in a flat way, as for equivalent Giotto programs, or in a hierarchical way. Details on HTL code generation can be found in [17].

3 Conclusion

We have discussed the notion of logical execution time (LET) and provided an overview of LET programming languages and runtime systems as well as some LET-inspired models of computation. We have also highlighted the key features of Giotto and its successor, the Hierarchical Timing Language (HTL), using program examples. The purpose of the rather informal presentation is to encourage the readers to study the LET paradigm further through original sources.

Acknowledgements The idea of logical execution time came up in 2000 in Thomas A. Henzinger's research group at UC Berkeley and has since been advanced by the hands of many people. Our bibliography lists quite some, but probably not all for which we apologize. We thank Eduardo R.B. Marques for his ongoing work on HTL and discussions related to it. The writing of this chapter has been supported by the EU ArtistDesign Network of Excellence on Embedded Systems Design and the Austrian Science Funds P18913-N15 and V00125.

References

1. J. Auerbach, D.F. Bacon, R. Guerraoui, J.H. Spring, and J. Vitek. Flexible task graphs: a unified restricted thread programming model for java. *SIGPLAN Not.*, 43:1–11, June 2008.
2. J. Auerbach, D.F. Bacon, D. Iercan, C.M. Kirsch, V.T. Rajan, H. Röck, and R. Trummer. Java takes flight: Time-portable real-time programming with Exotasks. In *Proc. ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. ACM, 2007.
3. J. Auerbach, D.F. Bacon, D. Iercan, C.M. Kirsch, V.T. Rajan, H. Röck, and R. Trummer. Low-latency time-portable real-time programming with Exotasks. *ACM Transactions on Embedded Computing Systems (TECS)*, 8(2):1–48, January 2009.
4. D.F. Bacon, P. Cheng, and V.T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proc. Symposium on Principles of Programming Languages (POPL)*, pages 285–298. ACM, 2003.
5. G. Berry. The foundations of Esterel. In C. Stirling G. Plotkin and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.
6. C. Buckl, D. Sojer, and A. Knoll. Ftos: Model-driven development of fault-tolerant automation systems. In *Proc. International Conference on Emerging Techonologies and Factory Automation (ETFA)*. IEEE, 2010.
7. G. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer, 1997.
8. S.S. Craciunas, C.M. Kirsch, H. Payer, H. Röck, and A. Sokolova. Programmable temporal isolation through variable-bandwidth servers. In *Proc. Symposium on Industrial Embedded Systems (SIES)*. IEEE, 2009.

9. S.S. Craciunas, C.M. Kirsch, H. Röck, and R. Trummer. The JAviator: A high-payload quadrotor UAV with high-level programming capabilities. In *Proc. AIAA Guidance, Navigation and Control Conference (GNC)*, 2008.
10. J. Dabney and T. Harmon. *Mastering Simulink*. Prentice Hall, 2003.
11. P. Derler and S. Resmerita. Flexible static scheduling of software with logical execution time constraints. In *Proc. International Conference on Embedded Systems and Software (ICESS)*. IEEE, 2010.
12. J. Eker, J.W. Janneck, E.A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity—the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, January 2003.
13. E. Farcas and W. Pree. Hyperperiod bus scheduling and optimizations for tdl components. In *Proc. International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2007.
14. C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *Proc. International Workshop on Embedded Software*, volume 2211 of *LNCS*, pages 469–485. Springer, 2001.
15. A. Ghosal, T.A. Henzinger, D. Iercan, C.M. Kirsch, and A.L. Sangiovanni-Vincentelli. A hierarchical coordination language for interacting real-time tasks. In *Proc. International Conference on Embedded Software (EMSOFT)*. ACM, 2006.
16. A. Ghosal, T.A. Henzinger, C.M. Kirsch, and M.A.A. Sanvido. Event-driven programming with logical execution times. In *Proc. International Workshop on Hybrid Systems: Computation and Control (HSCC)*, volume 2993 of *LNCS*, pages 357–371. Springer, 2004.
17. A. Ghosal, D. Iercan, C.M. Kirsch, T.A. Henzinger, and A. Sangiovanni-Vincentelli. Separate compilation of hierarchical real-time programs into linear-bounded Embedded Machine code. *Science of Computer Programming*, 2010.
18. N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer, 1993.
19. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9), 1991.
20. T.A. Henzinger, B. Horowitz, and C.M. Kirsch. Embedded control systems development with Giotto. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. ACM, 2001.
21. T.A. Henzinger, B. Horowitz, and C.M. Kirsch. Giotto: A time-triggered language for embedded programming. In *Proc. International Workshop on Embedded Software (EMSOFT)*, volume 2211 of *LNCS*, pages 166–184. Springer, 2001.
22. T.A. Henzinger, B. Horowitz, and C.M. Kirsch. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, January 2003.
23. T.A. Henzinger and C.M. Kirsch. The Embedded Machine: Predictable, portable real-time code. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 315–326. ACM, 2002.
24. T.A. Henzinger and C.M. Kirsch. A typed assembly language for real-time programs. In *Proc. International Conference on Embedded Software (EMSOFT)*, pages 104–113. ACM, 2004.
25. T.A. Henzinger and C.M. Kirsch. The Embedded Machine: Predictable, portable real-time code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(6):33–61, October 2007.
26. T.A. Henzinger, C.M. Kirsch, R. Majumdar, and S. Matic. Time safety checking for embedded programs. In *Proc. International Workshop on Embedded Software (EMSOFT)*, volume 2491 of *LNCS*, pages 76–92. Springer, 2002.
27. T.A. Henzinger, C.M. Kirsch, E.R.B. Marques, and A. Sokolova. Distributed, modular HTL. In *Proc. Real-Time Systems Symposium (RTSS)*. IEEE, 2009.
28. T.A. Henzinger, C.M. Kirsch, and S. Matic. Schedule-carrying code. In *Proc. International Conference on Embedded Software (EMSOFT)*, volume 2855 of *LNCS*, pages 241–256. Springer, 2003.

29. T.A. Henzinger, C.M. Kirsch, and S. Matic. Composable code generation for distributed Giotto. In *Proc. ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. ACM, 2005.
30. T.A. Henzinger, C.M. Kirsch, M.A.A. Sanvido, and W. Pree. From control models to real-time code using Giotto. *IEEE Control Systems Magazine (CSM)*, 23(1):50–64, February 2003.
31. C.M. Kirsch. Principles of real-time programming. In *Proc. International Workshop on Embedded Software (EMSOFT)*, volume 2491 of *LNCS*, pages 61–75. Springer, 2002.
32. C.M. Kirsch, L. Lopes, and E.R.B. Marques. Semantics-preserving and incremental runtime patching of real-time programs. In *Proc. Workshop on Adaptive and Reconfigurable Embedded Systems (APRES)*, 2008.
33. C.M. Kirsch, M.A.A. Sanvido, and T.A. Henzinger. A programmable microkernel for real-time systems. In *Proc. ACM/USENIX Conference on Virtual Execution Environments (VEE)*. ACM, 2005.
34. H. Kopetz. *Real-time Systems Design Principles For Distributed Embedded Applications*. Kluwer, 1997.
35. I. Lee, J. Leung, and S.H. Son, editors. *Handbook of Real-Time and Embedded Systems*, chapter The Evolution of Real-Time Programming. CRC Press, 2007.
36. C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.
37. J. Liu and E.A. Lee. Timed multitasking for real-time embedded software. *IEEE Control Systems Magazine (CSM)*, 23(1):65–75, February 2003.
38. N. Wirth. Towards a discipline of real-time programming. *Comm. ACM*, 20:577–583, 1977.
39. Y. Zhao, J. Liu, and E.A. Lee. A programming model for time-synchronized distributed real-time systems. In *Proc. Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 259–268. IEEE, 2007.