# Event-driven Programming with Logical Execution Times[*]

Arkadeb Ghosal, Thomas A. Henzinger, Christoph M. Kirsch, and Marco A.A. Sanvido

University of California, Berkeley
{arkadeb,tah,cm,msanvido}@eecs.berkeley.edu

**Abstract.** We present a new high-level programming language, called xGIOTTO, for programming applications with hard real-time constraints. Like its predecessor, xGIOTTO is based on the LET (logical execution time) assumption: the programmer specifies when the outputs of a task become available, and the compiler checks if the specification can be implemented on a given platform. However, while the predecessor language GIOTTO was purely time-triggered, xGIOTTO accommodates also asynchronous events. Indeed, through a mechanism called event scoping, events are the main structuring principle of the new language. The xGIOTTO compiler and run-time system implement event scoping through a tree-based event filter. The compiler also checks programs for determinism (absence of race conditions).

## 1 Introduction

One of the key issues in real-time software is the development of high-level programming languages. On one hand, a real-time programming language should be sufficiently abstract as to support the automatic verification of programs against mathematical models such as hybrid automata or Simulink, which are commonly used in practice. On the other hand, the language should be sufficiently concrete as to support the automatic compilation of a program into efficient code. While some tools generate code directly from mathematical models (Real-Time Workshop, dSpace), the resulting code is useful for rapid prototyping but, especially on distributed platforms, it is neither sufficiently efficient nor sufficiently reliable to be used in safety-critical products. By contrast, we envision a future where hard real-time properties are guaranteed by a compiler that produces code of sufficient quality such that, as with current optimizing compilers for conventional (non-real-time) programming languages, manual code tweaking is rarely (if ever) necessary or desirable.

Previous attempts to define real-time languages fall mostly into two categories. The first approach uses *priorities* to specify (indirectly) the relative deadlines of software tasks [1]. This approach supports efficient code generation based on scheduling theory [2]. The resulting run-time behavior of a program, however, is highly nondeterministic; for example, varying execution times of tasks cause race conditions. This makes

[*] This research is supported by the AFOSR MURI grant F49620-00-1-0327, the DARPA SEC grant F33615-C-98-3614, the MARCO GSRC grant 98-DT-660, and the NSF grants CCR-0208875 and CCR-0225610.

program verification difficult. The second approach is based on the *synchrony* assumption [3], which postulates that the execution platform is sufficiently fast as to complete all computation before the next environment event arrives. This approach leads to deterministic behavior and supports formal verification. It is, however, nontrivial to compile synchronous programs if either non-negligible execution times or distributed execution platforms are involved [4, 5]. We submit that the priority-based approach, which sacrifices determinacy in order to accommodate varying physical task execution times, is insufficiently abstract; and that the strictly synchronous approach, which sacrifices non-negligible task execution times in order to recover determinacy, is insufficiently realistic about the physical platform.

We propose an intermediate approach, which is based on the LET (logical execution time) assumption. Using LET, the programmer specifies with every task invocation the logical execution time of the task, that is, the time (or event) at which the task provides its outputs. The compiler makes sure that, on a specified platform, the outputs are computed in time. If the outputs are ready early, then they are made visible only when the specified logical execution time expires. This buffering of outputs achieves determinacy in both timing (no jitter) and functionality (no race conditions). LET programming, therefore, trades code efficiency in favor of code predictability when compared with traditional task scheduling, which makes all outputs visible as soon as they become available. We have demonstrated, however, that the loss in efficiency is insignificant even in high-performance control applications, such as helicopter flight control [6]. When compared with the synchrony assumption, LET programming trades mathematical expressiveness in favor of computational realities: it accommodates tasks with varying execution times, but in order to avoid fixpoint issues, all logical execution times are assumed to be strictly positive.

Previously, we have proposed and implemented a LET-based language for time-triggered programming, called GIOTTO [7]. In this paper we generalize GIOTTO to accommodate also asynchronous events. Indeed, through a mechanism called event scoping, events are the main structuring principle of the new language, which is called xGIOTTO. Event scoping admits a variety of ways for handling events within a hierarchical block structure: an out-of-scope event may either be ignored, or it may be postponed until the event comes back into scope, or it may cause the current scope to terminate as soon as all currently active tasks are terminated. The xGIOTTO compiler and run-time system implement event scoping through a tree-based event filter. The xGIOTTO compiler also checks programs for determinism (absence of race conditions caused by multiple tasks terminating at the same time). Finally, we show how the compiler could check for time safety (schedulability within logical execution times). This has not yet been implemented in the current compiler prototype.

## 2 The xGiotto Language

xGIOTTO is an event-driven real-time programming language that is built around the notion of software tasks with logical execution times. A *LET task* is sequential code operating on memory that is assigned to the task upon its release and which is not accessible to any other tasks. The memory holds the input and output as well as possible

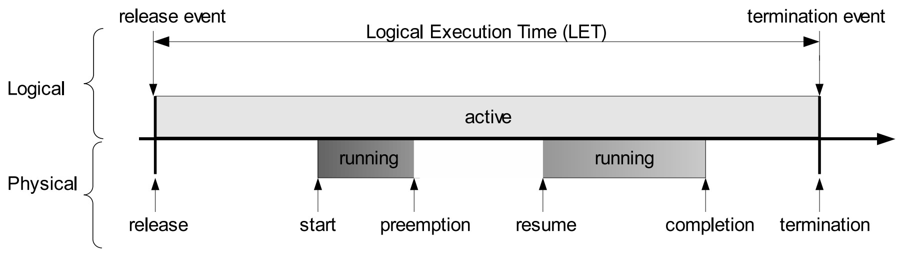

**Fig. 1.** Logical and physical execution of a task

state information of the task. In xGIOTTO, the same task may be instantiated on different memory at the same time. This task model is more general than GIOTTO tasks [7], which consist of both code and memory (i.e., fi xed input and output ports).

Figure 1 shows that the logical execution of a LET task begins with the *release* of the task and ends with the *termination* of the task. The release as well as the termination of a LET task are triggered by events such as clock ticks or sensor interrupts. This is a generalization of GIOTTO tasks, which can be released and terminated only at clock ticks. From the release event to the termination event, the task is called *active*. A LET task is *time-safe* on some given hardware if the task *completes* execution on that hardware before the termination event occurs. The output of a time-safe LET task is made accessible to other tasks and to actuators only when the termination event occurs, even if the task completes its physical execution earlier. Similarly, the input of a LET task is written into its assigned memory when the release event occurs, not when the task actually *starts* executing. As a consequence, a LET task always exhibits the same behavior in the value and time domain on different hardware as long as the task is time-safe. Note that proving time safety is usually impossible if unconstrained events are used as release or termination events for LET tasks. For example, an unconstrained event may occur immediately after a LET task is released. This problem could be solved by an *explicit* environment assumption on the minimal inter-arrival time between events. With xGIOTTO, however, we propose a new approach based on the notion of *event scoping*, which allows us to encode an *implicit* environment assumption in xGIOTTO programs. Event scoping temporarily disables event monitoring for a subset of the observed events. In this way, the environment assumption is reflected by the control structure of the program itself.

xGIOTTO has three important programming constructs. The statement `react {b} until [e]` is a *reaction block*, with a body `b` of xGIOTTO statements and an *until event* `e`, which determines when the reaction block, and the tasks that are released within the block, are terminated. The statements in `b` are executed logically in zero time but possibly at different time instants, if the reaction block contains nested reaction blocks. Even if all statements in `b` have been executed, the reaction block waits for the until event `e` to occur. During the wait, released tasks may be executed by the system scheduler. The second type of statement, `release t(in)(out)` fi rst allocates memory for a new instance of the task `t`, then loads data stored in the list `in` of input ports into the allocated memory, and fi nally releases `t` to the system scheduler for execution. When the enclosing reaction terminates, the task is assumed to be complete and its output is made available in the list `out` of output ports, which can be used as input for other tasks. Third, the statement `when [e] r` enables the *reaction* `r`, which is a parallel composition of reaction blocks, to be invoked when the *when event* `e` occurs. If two sequential `when` statements share the same event `e`, then the corresponding reactions are invoked by two different occurrences of `e`. If, however, the two `when [e]` statements are nested, then the corresponding reactions are invoked by the same occurrence of `e`.

A reaction block in XGIOTTO defines the event scope for the statements in its body. An *event scope* consists of the until event of the reaction block and the when events of the `when` statements in the body of the reaction block. Upon invoking the reaction of one of the `when` statements, the current event scope is pushed onto a stack (i.e., it becomes *passive*) and a new event scope is created and becomes the *active* scope. In general, a reaction is a parallel composition of reaction blocks. If two or more reaction blocks are invoked in parallel, then the scope of the parent block is pushed onto the stack and the scopes of all parallel blocks become active. Therefore we have a tree of scopes with the root of the tree being the initial scope, and the leaves of the tree being the active scopes. There are two ways for parallel reaction blocks to terminate. If the parallel reaction blocks are invoked with `wait`-parallelism, then the until event of one of the blocks will close the corresponding leaf of the tree. Consequently, the entire reaction consisting of `wait`-parallel reaction blocks terminates once all until events have occurred, and then the parent scope is resumed. In contrast, if the parallel reaction blocks are invoked with `asap`-parallelism, then the until event of any one of the reaction blocks disables the sibling blocks. Disabling a reaction block does not cause its immediate termination, but it implies that no new activity (e.g., task releases) will happen until the until event of the reaction block occurs: all when events are erased from the disabled event scope, leaving only its until events active.

An event of an active scope either, in the case of a when event, invokes a reaction, or in the case of an until event, terminates the corresponding scope. If an event `e` of an active scope can both invoke a reaction as well as terminate the scope, then the termination action has precedence. An event of a passive scope can be handled in the following three ways: it may be ignored (keyword `forget`); or it may be postponed until its scope becomes active again, once all descendent blocks have terminated (keyword `remember`); or it may disable all descendent blocks, thus speeding up their termination (keyword `asap`). Note that only active until events can terminate active tasks; in particular, active tasks cannot be prematurely terminated, neither by the termination of `asap`-parallel reaction blocks nor by passive `asap` events.

Since XGIOTTO is a generalization of the GIOTTO language [7], consider first the two GIOTTO program fragments on the left of Figure 2. A GIOTTO mode specifies a set of periodic tasks. The mode `m` shown here contains a task `t1` with a period of 20 ms and a task `t2` with a period of 10 ms. The LET of a GIOTTO task is equal to its period. At 0 ms, both tasks load input (code not shown here) into their memory and are then released to execute concurrently. At 10 ms, the result of task `t2` is made accessible to actuators and to other tasks. However, task `t1` may load new input only at 20 ms, even if `t1` has not had the chance to start before 10 ms. On the other hand, `t2` now loads

```
mode m() period 20 {
  taskfreq 1 do t1();
  taskfreq 2 do t2();
}

mode n() period 60 {
  taskfreq 3 do t1();
  taskfreq 2 do t2();
}
```

```
react {
  release t1()();
  begin
    react {
      release t2()();
    } until [10];
    react {
      release t2()();
    } until [10];
  end
} until [20];
```

```
react {
  whenever [20]
    react {
      release t1()();
    } until [20];
} until [60] ||
react {
  whenever [30]
    react {
      release t2()();
    } until [30];
} until [60];
```

**Fig. 2.** Giotto and xGiotto code fragements

new input such as sensor data, but does not yet have access to the output of `t1`, even if `t1` has already completed execution. For now, `t2` is released to execute a second time, logically in parallel with `t1`. At 20 ms, the results of both tasks are made accessible, possibly to each other, and a new round of mode `m` begins. The xGiotto fragment in the middle of Figure 2 implements exactly the behavior of one round of mode `m`. For simplicity, we have omitted the input and output ports of tasks. The code is a sequence of two reaction blocks. Initially the code releases task `t1` and executes the first inner block, which releases task `t2`. We write `10` (resp. `20`) for the event that recurs every 10 ms (resp. 20 ms). The termination of a task is defined by the until event of the surrounding reaction block, and therefore `t1` and `t2` terminate at 20 ms and 10 ms, respectively. At 10 ms, the second inner block is entered. Now, task `t2` is released a second time and terminated at 20 ms.

The right column of Figure 2 implements, in xGiotto, one round of the Giotto mode `n` with two nonharmonic tasks: task `t1` has again a period of 20 ms but task `t2` has now a period of 30 ms. For this, the xGiotto program uses a reaction that consists of two parallel reaction blocks. The first reaction block releases the task `t1` every 20 ms: the `whenever [20]` statement invokes its reaction every 20 ms, i.e., at 0 ms, 20 ms, and 40 ms, respectively. The second reaction block releases the `t2` task every 30 ms. Both parallel blocks terminate at 60 ms, implementing one round of the nonharmonic Giotto mode `n`.

In the middle code fragment of Figure 2, we add an asynchronous event `async`, which instantiates a task `ta` with a LET of 1 ms. This cannot be done in Giotto. We use the hierarchical structure of xGiotto to constrain the times at which the asynchronous event may cause the release of a new task instance. First consider the left column of Figure 3. Since the event `async` is remembered, if it occurs between 0 ms and 10 ms, during the first inner reaction block, then task `ta` is released at 10 ms and terminated at 11 ms. Since the event `10` is also remembered (by default), the second inner reaction block is invoked at 11 ms, releasing task `t2` a second time, and terminated at 20 ms. If `async` occurs, instead, between 10 ms and 20 ms, then task `ta` is never released, because until events —i.e., block termination (at 20 ms)— have precedence over when events. If the event `async` would have been specified as `forget` instead of `remember`, then task `ta` would never be released, because we assume that no two unrelated events can happen at exactly the same time (e.g., `async` cannot happen at exactly 10 ms). Now consider the middle column of Figure 3. Here, the event `async`

```
react {
  when remember [async]
    react {
      release ta()();
    } until [1];
  release t1()();
  begin
    react {
      release t2()();
    } until [10];
    react {
      release t2()();
    } until [10];
  end
} until [20];
```

```
react {
  release t1()();
  begin
    react {
      when [async] react {
        release ta()();
      } until [1];
      release t2()();
    } until [10];
    react {
      when [async] react {
        release ta()();
      } until [1];
      release t2()();
    } until [10]
  end
} until [20];
```

```
react {
  when asap [async]
    react {
      release ta()();
    } until [1];
  release t1()();
  begin
    react {
      when [asyncb] react {
        release tb()();
      } until [1];
      release t2()();
    } until [10];
  end
} until [20];
```

**Fig. 3.** xGIOTTO code fragments with asynchronous event handling

may be serviced twice, once between 10 ms and 20 ms, and a second time between 10 ms and 20 ms. While our default specification of an event is `remember`, note that in this case, it does not matter if `async` is specified as `forget` or `remember`. The third column of Figure 3 shows the use of an `asap` event. In this example an occurrence of the `async` event between 0 ms and 10 ms disables the reaction to the `asyncb` event and releases task `ta` at 10 ms.

## 3 Syntax

We refer to the language manual [8] for a full definition of the language. Here we introduce only the syntax necessary for understanding the most important xGIOTTO concepts:

```
Program  = "program" Ident ’{’ [ConstDecl] [TypeDecl] [PortDecl] [EventDecl]
           {ReactionDecl | TaskDecl} ReactionBody ’}’.

ConstDecl = "const" {Ident "=" Number ";"}.
TypeDecl = "type" {TypeId (("array" Number "of" TypeId) |
           ("record" ’{’ {TypeId Ident ’;’ } ’}’))’;’}.
PortDecl =  "port"  {TypeId Ident [InitPort] ’;’}.
EventDecl = "event" {TypeId Ident ["at" Ident] ’;’ }.

TaskDecl = "task" Ident Pars "output" Pars ["var" Pars] Body.
Body = ’{’ StatSeq ’}’.
StatSeq = Statement {";" Statement }.
Pars = ’(’ [TypeId Ident {’,’ TypeId Ident }] ’)’.

ReactionDecl = "react" Ident ReactionBody "until" ’[’ TypeId Ident ’]’.
ReactionBody = ’{’ Triggers Releases [("begin" | "loop") RStatSeq "end" ";"] ’}’.
Triggers = { [Condition] ("when" | "whenever") Event Reaction ";" }.
Releases = { [Condition] "release" Ident ParsRef ParsRef ";"}.
RStatSeq = Reaction {";" Reaction }.
Reaction = ReactionBlock { (’||’ | ’&&’ ) ReactionBlock }.
ReactionBlock = "react" ((Ident) | ReactionBody) "until" Event.
ParsRef = ’(’ [Ident {’,’ Ident }] ’)’.
Event = ["asap" | "remember" | "forget"] ’[’ [Number] Ident ’]’.
Condition = ’(’ BoolExpression ’)’.
```

**Constant, type, port, and event declarations.** Constant declarations allow to associate a name with a value. Type declarations associate a name with a structured data type. Each port has a fixed type and can be initialized, if an initial value different from a type-dependent default value is desired. Similarly, each event has as a fixed type, the value being assigned by the interrupt generating the event. The events `time` and `now` are predefined. The integer event `time` is bound to the system clock. The event `now` is a placeholder for the current event and can be used in `when` statements only (not in `whenever` and `until` statements). Events are structured hierarchically in a tree, e.g., the event `20` occurs at every other occurrence of the event `10`. Logically, no two *unrelated* events (i.e., neither one is a descendent of the other in the event tree) can happen simultaneously, as they are sequenced by the interrupt handler.

**Task declarations.** A task header specifies a task name, formal input parameters, formal output parameters, and local variables. The task body is a standard sequential program without reference to events (we omit the exact syntax). The input parameters are passed by value, i.e., they are local ports to which the actual parameters are assigned as initial values upon release of a task instance. The output parameters are passed by value-reference, i.e., they are local ports with the actual parameters as initial values, but their values are instantaneously copied back to the actual parameters at termination of the task instance.

**Reaction block declarations.** A header specifies the name of the reaction block and a formal until-event parameter. The body of the reaction block contains three parts: (1) conditional `when` and `whenever` statements called *trigger* statements, (2) conditional `release` statements, and (3) sequential reaction statements. The trigger statements whose condition is true specify the active events of the reaction block (in addition to the until event, which is also active) and the corresponding reactions. The occurrence of an active event is processed in the order in which the trigger statements are declared. The events can be specified as `forget`, `remember`, or `asap`. A `whenever` statement corresponds to a `when` statement that reenables itself immediately after its event occurs, until the surrounding reaction block is terminated. The reenabled active event will be processed after all the other previously enabled events are processed. The `release` statements whose condition is true hand task instances to the system scheduler. The sequential reaction statements can be declared either as a one-time sequence or as a loop of reaction statements. Each reaction statement is an `asap`-parallel (defined by `&&`) or a `wait`-parallel (defined by `||`) composition of reaction blocks. Each reaction invocation renders the active events passive. When the until event of the reaction block is active and arrives, the scope and all tasks released in its scope are terminated, and control is returned to the invoking reaction block, reenabling its active events. The trigger and `release` statements are executed instantaneously (in logical zero time), but time passes between events; in particular, time passes during the execution of a `react` statement, between the trigger and `release` statements of the reaction block, and the until event of the block.

**Core-xGiotto.** The syntax of xGiotto given by the above grammar is used in all program examples in this paper. However, the corresponding formal semantics is provided only for a fully expressive fragment called core-xGiotto. Each xGiotto program can be transformed into a core-xGIOTTO by replacing each call to a named reaction block with the code of the reaction block, and by removing from each reaction block all sequential reaction statements as follows: each one-time sequence of reaction statements is replaced by a set of `when` trigger statements; each loop, by a set of `whenever` trigger statements. xGIOTTO programs with recursive (cyclic) calls of reaction blocks are considered illegal, because they represent infinite core-xGIOTTO programs. Note that in core-xGIOTTO, a reaction block consists of a sequence of trigger statements, a set of `release` statements, and an until event.

**Example of a control program.** Figure 4 shows a program for controlling a one-dimensional system with an actuator `u`, a position sensor `p`, and a velocity sensor `v`. The controller is a cascaded controller combining a proportional velocity controller in the inner loop, and a proportional position controller in the outer loop. The proportional controllers are used for simplicity; more advanced controller algorithms can easily replace the task code of `Pos` and `Vel`. The velocity controller updates the actuator every 2 time units, whereas the position controller updates the target velocity every 3 time units. The target of the position controller is a position point stored in the position array `p`. When the position is reached, the next target position is chosen from `p`. If the last position in `p` is reached, the system stabilizes at the actual position. The system will follow the trajectory stored in `p` until the last point of the trajectory is reached. In addition to the two periodic tasks, we introduce an asynchronous task, which computes an array of target position points for a given set of way-points. The asynchronous task is triggered by an external event, such as an operator input. We limit the number of asynchronous way-point updates to one every 6 time units.

## 4 Semantics

The execution of an xGIOTTO program yields a possibly infinite sequence of configurations. Each configuration consists of the values of all program variables (*ports*) and a tree of scopes. Each *scope* corresponds to a reaction block of the program; it contains a termination event, a trigger queue, and a ready set. The active scopes are the leaves. The *trigger queue* contains the enabled reactions, each associated with an invocation event: if the invocation event for an enabled reaction of an active scope arrives, then the first such reaction is invoked, and for each of its parallel reaction blocks, a new scope is added as a child to the present scope, rendering that scope passive. The *ready set* of a scope contains the tasks that have been released in the scope; their termination event is the termination event of the scope. Each `when` and `whenever` statement of a reaction block adds an event-reaction pair to the trigger queue; each `release` statement adds a task to the ready set. The termination event of an active scope removes that scope.

In the following, we make this formal by defining a state-transition graph whose states are the program configurations, and whose transitions correspond to the occurrence of a new event, the termination of a scope, and the invocation of a core-xGIOTTO reaction. When a new event arrives, first an *event transition* records the event occurrence in all scopes, then a sequence of *termination transitions* removes (possibly nested) scopes that have terminated, and finally a sequence of *reaction transitions* adds (possibly nested) new scopes by invoking enabled reaction blocks. If no more reaction blocks

```
program CascadedAsyncController {

 const Kp = 0.8;

 type
  waypoint array 10  of real;
  points   array 100 of real;

 port
  real p;  real v; /*sensor values*/
  /* destination values */
  points dp; int ip; real dv;
  real u; /* actuator */

 event
  waypoint A;
  bool start; bool stop;

 task Waypoints2Pos (waypoint wp)
   output (points dp, int ip) {
   ip = 0; /* reset start point*/
   /* compute a spline */
   dp = spline(wp);
 }

 task Pos(points dp, int ip, real p)
   output (int newip, real u)
   var (real error) {
   if ((error > 1) | (ip == 9))
    newip = ip;
   else
    newip = ip+1;
   error = dp[newip] - p;
   u = Kp*error;
 }
```

```
 task Vel(real dv, real v)
   output (real u) var (real error) {
   error = dv - v;
   u = Kp*error;
 }

 react computePos {
   release Waypoints2Pos(A)(dp, ip);
 } until [int c]

 react mode {
   whenever remember [6time]
     react {
       release Vel(dv, v)(u);
       whenever [2time]
         react {
           release Vel(dv, v)(u);
         } until [2time];
     } until [6time]
     || react {
          release Pos(dp,ip,p)(ip,dv);
          whenever [3time]
            react {
              release Pos(dp,ip,p)(ip,dv);
            } until [3time];
     } until [6time]
     || react {
          when [A]
            react computePos until [1time];
     } until [6time];
 } until [bool b]

 { whenever [start]
      react mode until asap [stop];
 }
}
```

**Fig. 4.** xGIOTTO program for a cascaded controller

can be invoked, the configuration is called *waiting*, and the arrival of the next event is awaited. All transitions take place in logical zero time; time advances only in waiting configurations. No two unrelated events arrive at the same time.

**Configurations.** Consider an xGIOTTO program with ports $P$, events $E$, task addresses $T$, and reaction addresses $R$. A *configuration* is a pair $(\Sigma, \Delta)$, where $\Sigma$ is a function from the port set $P$ to values, and $\Delta$ is a labeled tree —each node is labeled by a scope. A *scope* is a tuple $(U, Q, S, \alpha)$, where $U$ specifies the termination event instance, $Q$ is a queue of triggers, $S$ is a set of task instances, and $\alpha \in \{\mathtt{asap}, \mathtt{wait}\}$ specifies the parallelism for (the siblings of) the scope. An *event instance* is a tuple $(e, n, \beta)$, where $e \in E$, $n \in \mathbb{N}$, and $\beta \in \{\mathtt{A}, \mathtt{R}, \mathtt{F}\}$, denoting `asap`, `remember`, and `forget`, respectively; the tuple implies that the required action (terminating a scope, or invoking a reaction) happens when the event $e$ occurs $n$ number of times. A *trigger* is a tuple $(I, m, r)$, where $I$ specifies the invoking event instance, $m \in \mathbb{N} \cup \{\perp\}$ records if the trigger is registered by a `when` ($m = \perp$) or `whenever` ($m \in \mathbb{N}$) statement, and $r \in R$ is the invoked reaction. A *task instance* is a tuple $(t, p_i, s_i, p_o)$, where $t \in T$ and $p_i, p_o \subseteq P$, and $s_i$ is a function from $p_i$ to values. At task termination the output ports

$p_o$ are updated to the values computed by the task $t$, given that the input ports $p_i$ had the values $s_i$ when the task was released.

In the *initial* confi guration, all ports have their initial values, and the scope tree has a single node labeled by the scope of the `main` reaction block. A scope is *terminating* if it is a leaf scope and its terminating event instance has the form $(e, 0, \beta)$. A scope is *reacting* if it is a leaf scope and its trigger queue contains a trigger with an invoking event instance of the form $(e, 0, \beta)$; this is called an *invoked* trigger. A confi guration is *waiting* if all its scopes are neither terminating nor reacting.

**Event transitions.** For each waiting confi guration $(\Sigma, \Delta)$ and event $e' \in E$, an event successor is obtained by replacing each event instance $(e, n, \beta)$ in $\Delta$ with $(e, n-1, \beta)$ if $n > 0$ and $e = e'$ and either (1) $\beta \in \{\mathtt{A}, \mathtt{R}\}$ or (2) $\beta = \mathtt{F}$ and the event instance occurs in a leaf scope. Moreover, if the terminating event instance of a scope is replaced by $(e', 0, \mathtt{A})$, then the trigger queues of all descendent scopes are emptied.

**Termination transitions.** For each confi guration $(\Sigma, \Delta)$ which has a terminating scope, a termination successor is obtained by removing the leaf with the terminating scope. Second, for each task instance $(t, p_i, s_i, p_o)$ of the removed scope, the port values of $p_o$ in $\Sigma$ are updated by applying task $t$ to the port values $s_i$ of $p_i$. Third, if the removed scope is `asap`-parallel, then the trigger queues of all sibling scopes and their descendents are emptied. If the program is free of race conditions (see next section), then each sequence of termination transitions leads, independent of their order, to a unique confi guration without terminating scopes.

**Reaction transitions.** For each confi guration $(\Sigma, \Delta)$ which has no terminating scope but a reacting scope, if the fi rst invoked trigger in the queue is $g = ((e, 0, \beta), m, r)$, then a reaction successor is obtained by adding to the node with the reacting scope a set of children —one for each reaction block of $r$. Moreover, the trigger $g$ is removed from the queue, and if $m \neq \perp$, then the new trigger $((e, m, \beta), m, r)$ is appended at the end of the queue. The scope of each new node is computed by executing the corresponding reaction block: the termination event instance of the new scope is determined by the until event of the reaction block; the trigger queue of the new scope contains one trigger for each `when` and `whenever` statement whose condition is true in $\Sigma$, in the order of the statements; the ready set of the new scope contains one task instance for each `release` statement whose condition is true in $\Sigma$, where the values of the task input ports are taken from $\Sigma$; and the parallelism of the new scope is determined by whether the reaction block is composed with `asap`- or `wait`-parallelism. It is not diffi cult to see that each sequence of reaction transitions leads, independent of their order, to a unique waiting confi guration.

## 5 Program Analysis

The xGIOTTO compiler performs several program analyses. First, it does a conservative check and rejects programs whose execution may encounter a race condition. A race occurs when two tasks that are terminated by the same event write to the same port; in this case, the port value is not predictable. By contrast, for a given event sequence, programs without race conditions are executed deterministically. Second, since the memory of embedded systems is often constrained, the xGIOTTO compiler computes a conservative estimate for the memory requirements of a program. Third, we show how the compiler could check for time safety (schedulability) of a program on a given platform. The platform is specifi ed through WCETs (worst-case execution times) for all ¡¡¡¡¡¡¡ hscc04-fi nal.tex tasks. ======= tasks. ¿¿¿¿¿¿¿ 1.67

**Race detection.** A program trace is a sequence of transitions starting from the initial confi guration. A trace contains a *race* if it has two termination transitions that update the same port without an interspersed event transition. The absence of races can be checked precisely by a traversal of the exponential confi guration graph. The compiler performs a less precise, but conservative polynomial-time check on the program text. It associates with every reaction block $b$ a set $T(b)$ of *potential termination events*: the set $T(b)$ contains the until event of $b$, and if the until event of $b$ has type `remember` or `asap`, then $T(b)$ contains also all potential termination events of the immediate subblocks of $b$. If for any two distinct `release` statements $s$ and $s'$ that have an output port in common, the potential termination events of the reaction blocks containing $s$ and $s'$ are disjoint, then all program traces are race-free. A less conservative analysis might consider the confi guration graph, but with all port values abstracted.

**Resource requirements.** In order to allocate suffi cient memory, the compiler computes from conservative bounds on the size of the scope tree, trigger queues, and ready sets of an xGIOTTO program (the computation of exact bounds would again require a traversal of the confi guration graph). As the reaction block structure of a program is nonrecursive, the size of the scope tree is bounded. An upper bound on the tree size for a reaction block is 1 plus the maximum of the tree sizes for the contained reactions, and for each reaction, it is the sum of the tree sizes for the contained reaction blocks. The length of the trigger queue of a reaction block is bounded by the number of `when` and `whenever` statements of the block, and the size of the ready set of a reaction block is bounded by the number of `release` statements.

**Time-safety (schedulability) analysis.** The execution of an xGIOTTO program is *time-safe* if each active task instance completes before its termination event. Time safety, of course, depends not only on the program but also on the execution platform. In particular, for time-safety analysis, the xGIOTTO compiler needs WCET information. For example, if there is a single task instance, then the program is time-safe if the WCET is less than the LET. In general, there may be concurrent active task instances and time-safety checking requires a schedulability analysis. The xGIOTTO compiler uses discrete time. The WCET for each task is assumed to be a positive integer, and scheduling decisions (i.e., task preemptions) are taken only at integer times. For this purpose, we assume there is a periodic event called `tick`. In every waiting confi guration, the scheduler assigns to the CPU one of the tasks that have been released but not completed. Scheduling decisions take effect only if the subsequent event is a `tick` event. Then, an integer counter that keeps the task execution time is decremented. At task release the counter is initialized to the WCET, and if the termination event arrives before the counter is 0, then a *time-safety violation* occurs.

Formally, the schedulability of an xGIOTTO program (on a single CPU) is defi ned as a two-player safety game. The game graph is an extended confi guration graph, where

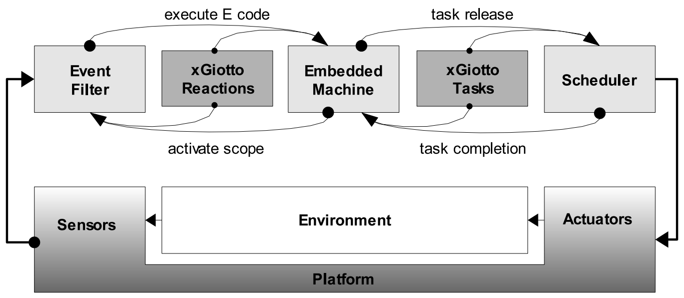

**Fig. 5.** The system architecture

each configuration is extended with execution-time counters for all active tasks, and in addition to event, termination, and reaction transitions, there are *scheduling transitions*. An *extended configuration* is a triple $(c, \nu, b)$, where $c = (\Sigma, \Delta)$ is a configuration, $\nu$ is a function that assigns a positive integer to each task instance in the ready set for each node of $\Delta$, and $b$ is a bit. The function $\nu$ indicates for each task instance the remaining (worst-case) execution time. The bit $b$ indicates which player moves next in the scheduling game: if $b = 0$, then the environment chooses an event or the system performs a termination or reaction transition; if $b = 1$, then the scheduler chooses a task to be executed. The transitions between extended configurations are: if $(c, c')$ is a transition other than an event transition on `tick`, then $((c, \nu, 0), (c', \nu, 0))$ is the corresponding extended transition; if $(c, c')$ is an event transition on `tick`, then $((c, \nu, 0), (c', \nu, 1))$ is the corresponding extended transition; and $((c, \nu, 1), (c, \nu', 0))$ is a *scheduling transition* if either (1) the scheduler does not schedule any task and $\nu' = \nu$, or (2) the scheduler schedules a task instance $\mathtt{i}$ of $c$ and $\nu'(\mathtt{i}) = \nu(\mathtt{i}) - 1$, and $\nu'(\mathtt{i}') = \nu(\mathtt{i}')$ for all task instances $\mathtt{i}'$ different from $\mathtt{i}$.

Player 1 is the system and its environment; they choose event, termination, and reaction transitions. Player 2 is the scheduler; it chooses scheduling transitions, i.e., it determines the task whose execution-time counter is to be decremented (while the execution-time counters of all other tasks stay unchanged). The program is *time-safe* with respect to a given WCET mapping (which maps each task to a WCET) if in this game player 2 —the scheduler— has a strategy to avoid time-safety violations forever. Safety games can be solved in linear time in the size of the game graph. Since the extended configuration graph is exponential (even if port values are abstracted), in theory the schedulability problem for XGIOTTO is complete for `EXPTIME`. It is an interesting question to look for restrictions on the program structure which make the problem tractable in practice, at least if ports values are abstracted. For example, in GIOTTO (which is a special case of XGIOTTO) the schedulability check can be done by a simple utilization test [10].

## 6 Implementation

The prototype implementation of the xGIOTTO system consists of a compiler and a run-time environment. The run-time environment is shown in the upper part of Figure 5 and executes the code generated by the xGIOTTO compiler. The generated code is divided into two parts, *reaction code* and *task code*. Reaction code is essentially E code (the instruction set of the E Machine [9]), whereas task code is similar to Java byte-code. This can be any platform-native code, but we chose to interpret the task code in our prototype and generate ¡¡¡¡¡¡¡ hscc04-fi nal.tex native code in a later stage of the project. The compiler checks for race conditions and determines upper bounds on the resource ======= native code in a later stage of the project. The compiler checks race conditions and determines upper bounds on the resource ¿¿¿¿¿¿¿ 1.67 requirements of the program. We are currently implementing a ¡¡¡¡¡¡¡ hscc04-fi nal.tex time-safety check with respect to given task WCETs. The run-time system also performs checks that raise exceptions when a time-safety violation is detected at run-time. This may happen if WCET data is wrong. The user can specify, again ======= time-safety check with respect to given task WCETs. The run-time system also performs checks that raise exceptions when a time-safety violation is detected at run time. Even if a static time-safety check is performed a time-safety violation may happen at run time if the WCET data is wrong. The user can specify, again ¿¿¿¿¿¿¿ 1.67 in xGIOTTO, how exceptions are handled.

The xGIOTTO run-time environment consists of three interacting components: the event fi lter, the E Machine, and the scheduler. The original E Machine is insuffi cient to implement event scoping; therefore we have augmented the architecture presented in [11] with an event fi lter. The *event filter* implements the event-scoping mechanism and presents the fi ltered events to the E Machine. The implementation of the event fi lter is tree-based, where each node of the tree is the event scope of a reaction block. The leaves of the tree are the active event scopes. An event scope is composed of the trigger events (from the `when` and `whenever` statements), the until event of the reaction block, and the set of released tasks. At run-time, the occurrence of an event is processed by the event fi lter. The event fi lter computes the event transition and the termination transitions on the tree of event scopes and gives to the E Machine a set of E code addresses, which correspond to the invoked reaction blocks. The E Machine interprets the E code, thus performing the reaction transitions. The E code instructions may release new tasks to the scheduler and enable new triggers. When all invoked reactions are processed by the E Machine, the system scheduler chooses a task to execute from the ready set of the active event scopes, and whenever such a task completes, the E Machine is notifi ed. In addition, the E Machine monitors the running tasks by detecting task overruns (time-safety violations). If a task overrun is detected (i.e., if a task termination event arrives before the task completes), a run-time exception is generated.

The lower part of Figure 5 shows the execution environment. The platform interacts with the environment through actuators and sensors. The actuators are driven by the task outputs and the sensors generate *raw* events (interrupts), which are handled by the event fi lter. The prototype system is implemented in Java and is able to run any xGIOTTO program on any Java virtual machine (JVM). The E Machine is available on several platforms, including JVM, POSIX, HelyOS, and KURT-Linux, and we are in the process of porting the xGIOTTO system to these platforms.

## 7 Related Work

xGIOTTO has been inspired by the GIOTTO [7] language. The GIOTTO programmer's model is restricted to time-triggered task release and termination, and therefore well-suited for control applications with a periodic task structure. The interest in investigating a LET-based programmer's model that can handle also asynchronous events and aperiodic tasks has been the main driver for the xGIOTTO language project. *Timed multitasking* (TM) [12] is based on a computational model similar to the LET assumption. However, in TM the execution time of each parallel task is logically fixed only by time, and not by general, dynamically scoped events as in xGIOTTO.

The zero-time execution of xGIOTTO statements is inspired by synchronous reactive languages, such as Esterel [13] and Lustre [14]. In synchronous reactive languages all computations are assumed to take zero logical time, as opposed to xGIOTTO, where all task computations have a strictly positive logical execution time. xGIOTTO, therefore, on one hand restricts the theoretical expressiveness of synchronous reactive languages, and on the other hand integrates them with scheduling theory. Esterel allows the parallel execution of tasks in a way similar to xGIOTTO. A parallel task can be started by the `exec` statement, and at its completion a signal is raised. While Esterel can stop the task execution, it cannot specify its termination point; it has no notion of LET. Moreover, event scoping would have to be explicitly coded into an Esterel program. Recent work in the synchronous language community has been aimed at relating logical (synchronous) time and physical (real) time. For example, Taxys [15] relaxes the zero-delay assumption with real-time constraints by merging the Esterel language and the Kronos real-time constraint verifier, and an extension to Lustre with a relaxed zero-delay assumption has been proposed as well [16].

nesC [17] is a programming language especially targeted to small, networked sensor devices. The goal of nesC is very similar to xGIOTTO. Both compilers check for race conditions. Interestingly, xGIOTTO task instantiation can be specified in nesC by using the `post` command, which releases a computation, but without explicitly giving a termination requirement. The main difference between nesC and xGIOTTO is the absence of the concept of time, and therefore no hard-real time constraints can be guaranteed by the nesC compiler. Also, the nesC programmer's model is platform-independent but not value-deterministic. In particular, the same program running on different platforms with the same input events may produce different results. Erlang [18] is a functional language for real-time embedded systems, specifically for the telecommunication domain. Erlang, like xGIOTTO, generates code for a virtual machine, and is therefore easily portable to different platforms. Erlang features the execution of parallel tasks but, like nesC, does not explicitly address real-time requirements apart from timeouts and the handling of run-time exceptions.

Real-Time Euclid [19] is a language designed specifically to address reliability and schedulability issues in time-constrained environments. The language definition forces every construct in the language to be time- and space-bounded. These restrictions make it easier to estimate the execution time of the program, and they facilitate scheduling to meet all deadlines. Therefore, RT-Euclid programs can always be analyzed for schedulability. However, RT-Euclid does not have any notion of event reaction and is therefore lacking an important aspect of embedded-systems programming. The programming language Flex [20] extends C++ by introducing explicit real-time constraints. In Flex, timing constraints can be specifi ed for each section of code. The run-time mechanism of Flex ensures that the timing constraints are satisfi ed, or else the block is aborted and an exception handler is invoked. In Flex timing constrains are guaranteed at run-time and no schedulability analysis is performed at compile-time. However, like xGIOTTO, both Flex and Real-Time Euclid defi ne a platform-independent logical execution model for real-time programs, which makes them predictable.

## References

1. Burns, A., Wellings, A.: Real-Time Systems and Programming Languages. Addison-Wesley (2001)
2. Buttazzo, G.: Hard Real-Time Computing Systems. Kluwer (1997)
3. Halbwachs, N.: Synchronous Programming of Reactive Systems. Kluwer (1993)
4. Girault, A., Ménier, C.: Automatic production of globally asynchronous, locally synchronous systems. In: Embedded Software. LNCS 2491. Springer (2002) 266–281
5. Girault, A., Nicollin, X.: Clock-driven automatic distribution of Lustre programs. In: Embedded Software. LNCS 2855. Springer (2003) 206–222
6. Henzinger, T.A., Kirsch, C.M., Sanvido, M.A.A., Pree, W.: From control models to real-time code using GIOTTO. IEEE Control Systems Magazine **23** (2003) 50–64
7. Henzinger, T.A., Horowitz, B., Kirsch, C.M.: GIOTTO: a time-triggered language for embedded programming. Proc. IEEE **91** (2003) 84–99
8. Sanvido, M.A.A., Ghosal, A., Henzinger, T.A.: xGIOTTO language report. Technical Report UCB//CSD-03-1261, UC Berkeley (2003)
9. Henzinger, T.A., Kirsch, C.M.: The Embedded Machine: predictable, portable real-time code. In: Proc. Programming Language Design and Implementation, ACM (2002) 315–326
10. Henzinger, T.A., Kirsch, C.M., Majumdar, R., Matic, S.: Time-safety checking for embedded programs. In: Embedded Software. LNCS 2491. Springer (2002) 76–92
11. Kirsch, C.M., Henzinger, T.A., Sanvido, M.A.A.: A programmable microkernel for real-time systems. Technical Report UCB/CSD-03-1250, UC Berkeley (2003)
12. Liu, J., Lee, E.A.: Timed multitasking for real-time embedded software. IEEE Control Systems Magazine **23** (2003) 65–75
13. Boussinot, F., de Simone, R.: The Esterel language. Proc. IEEE **79** (1991) 1293–1304
14. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data-fbw programming language Lustre. Proc. IEEE **79** (1991) 1305–1320
15. Bertin, V., Closse, E., Poize, M., Pulou, J., Sifakis, J., Venier, P., Weil, D., Yovine, S.: Taxys = Esterel + Kronos. A tool for verifying real-time properties of embedded systems. In: Proc. Decision and Control, IEEE **3** (2001) 2875–2880
16. Caspi, P., Curic, A., Maignan, A., Sofronis, C., Tripakis, S., Niebert, P.: From Simulink to Scade/Lustre to TTA: a layered approach for distributed embedded applications. In: Proc. Languages, Compilers, and Tools for Embedded Systems, ACM (2003) 153–162
17. Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., Culler, D.: The nesC language: a holistic approach to networked embedded systems. In: Proc. Programming Languages Design and Implementation, ACM (2003) 1–11
18. Armstrong, J., Virding, R., Wikström, C., Williams, M.: Concurrent Programming in Erlang. Prentice-Hall (1992)
19. Kligerman, E., Stoyenko, A.: Real-time Euclid: a language for reliable real-time systems. IEEE Trans. Software Engineering **12** (1986) 941–949
20. Kenny, K., Lin, K.J.: Building fexible real-time systems using the Flex language. IEEE Computer **24** (1991) 70–78