Time-Safety Checking for Embedded Programs*

Thomas A. Henzinger, Christoph M. Kirsch,
Rupak Majumdar, and Slobodan Matic

Department of Electrical Engineering and Computer Sciences
University of California, Berkeley, USA
{tah,cm,rupak,matic}@eecs.berkeley.edu

Abstract. Giotto is a platform-independent language for specifying software for high-performance control applications. In this paper we present a new approach to the compilation of Giotto. Following this approach, the Giotto compiler generates code for a virtual machine, called the E machine, which can be ported to different platforms. The Giotto compiler also checks if the generated E code is time safe for a given platform, that is, if the platform offers sufficient performance to ensure that the E code is executed in a timely fashion that conforms with the Giotto semantics. Time-safety checking requires a schedulability analysis. We show that while for arbitrary E code, the analysis is exponential, for E code generated from typical Giotto programs, the analysis is polynomial. This supports our claim that Giotto identifies a useful fragment of embedded programs.

1 Introduction

We have advocated a *platform-independent* approach to embedded programming [4, 5]: the programmer specifies the timing and functional aspects of the program, and the compiler checks if the program can be executed as intended on a particular platform in a particular environment. Besides providing the programmer with an application-level abstraction, and the obvious benefits of code reuse, this approach offers maximal flexibility in the implementation: failure to compile may be equally due to platform performance (CPUs too slow or too few) and platform utilization (scheduling scheme inadequate) and environment behavior (events too frequent), and therefore may be remedied by addressing any one or more of these factors. Perhaps the most important benefit of the platform-independent approach is that it permits a clean separation of timing and function. An embedded program written in this way, called an *E program*, consists of a timing part and a functional part. The *functional part* is a set of software processes, and the *timing part* is a control-flow skeleton that supervises the invocation of the software processes relative to events. Platform independence means that the programmer can think of each software process as an atomic

* Supported in part by the DARPA SEC grant F33615-C-98-3614, MARCO GSRC grant 98-DT-660, AFOSR MURI grant F49620-00-1-0327, NSF grant CCR-9988172, and a Microsoft Research Fellowship.

operation on a state; that is, as an infinitely fast, terminating program without internal synchronization points. The programmer's fiction of atomicity can be implemented by nonpreemption, as in the synchronous reactive languages [3]. We pursue maximal flexibility in the implementation and permit the system scheduler to preempt software processes as long as the compiler can ensure that the scheduler maintains logical atomicity. An execution of the program that maintains this fiction is called *time safe*. Thus a central task of the compiler is to check the time safety of all possible executions of a given E program on a given platform in a given environment. In other words, a compiler that checks time safety guarantees that the functional part of an E program can be executed as specified by the timing part.

Let us be more precise. The functional part of an E program consists of two kinds of software processes. Software processes with nonnegligible WCETs (worst-case execution times) need to be scheduled, and their execution can be preempted. These processes are called *tasks*. A typical example of a task is a control law computation. On the other hand, software processes with negligible WCETs —i.e., processes that can always be completed before the next environment event— can be executed synchronously. These processes are called *drivers*. A typical example of a driver is a sensor reading, or an actuator update. Both tasks (scheduled computation) and drivers (synchronous computation) are written in a conventional language, such as C. The timing part of an E program consists of a set of *E actions*. Each E action is triggered by an event, and may call a driver, which is executed immediately, or schedule a task, which is handed over to the scheduler of the operating system. It is important to note that the schedule action is independent of the scheduling scheme: between events that are separated in time, the OS allocates scheduled tasks to CPUs, but how this is done, is not specified by the E program. In this model, time safety —the logical atomicity of tasks— means that during an execution, the state of a scheduled task is not accessed (by a driver or another task) until the task completes. We have introduced two languages for specifying E actions. The *E machine* [5] is a virtual machine that executes *E code*, whose instructions can specify arbitrary sequences of E actions. *Giotto* [4] is a structured language for specifying limited combinations of E actions that occur in typical control applications, where a controller may switch between modes, and within each mode, periodically invoke a given set of tasks and drivers. From a Giotto source program we generate E code, similar to the way in which assembly code is generated from high-level programming languages. This offers portability, as Giotto programs can now be run on any implementation of the E machine.

Time safety is a property of an individual program execution. *Schedulability* is the existence of a scheduler that guarantees that all executions of a program are time safe. Here, we restrict ourselves to single-CPU platforms. Such a platform is specified by a WCET for each task. We solve two schedulability problems. First, we show that for arbitrary E code (defined in Section 2), schedulability checking is difficult: the problem corresponds to a game between the environment and the scheduler [1] on an exponential state space, and is therefore EXPTIME-complete (Section 3). Second, we show that for E code that is generated from a Giotto source program in a specific way (defined in Section 4), EDF (earliest-deadline-first) scheduling is well-defined and optimal. This fixes the strategy of the scheduler, and thus reduces schedulability checking to a reachability problem on an exponential state space, which is PSPACE-complete. Indeed, if we know that all modes of a Giotto program are reachable, then schedulability can be checked by solving a linear utilization equation for each mode independently, no matter how the program switches modes. This can be done in almost quadratic time (Section 5). These results give a technical justification for our intuition that Giotto captures a natural, widely applicable, and also easily schedulable fragment of E programs. They also provide the basis for the Giotto compiler we have implemented. The compiler generates E code following the algorithm of Section 4, and checks schedulability using the method of Section 5.

2 E Code

The E machine [5] is a virtual machine that mediates between the physical processes and the software processes of an embedded system through a control program written in *E code*. E code controls the execution of software processes in relation to physical events, such as clock ticks, and software events, such as task completion. E code is interpreted on the E machine in real time. In this paper, we restrict our attention to the *input-triggered* programs of [5]; they are *time-live*, that is, all synchronous computation is guaranteed to terminate.

Syntax. The E machine supervises the execution of *tasks* and *drivers* that communicate via *ports*. A task is application-level code that implements a computation activity. A driver is system-level code that facilitates a communication activity. A port is a typed variable. Given a set P of ports, a P *state* is a function that maps each port $p \in P$ to a value of the appropriate type. The set P is partitioned into three disjoint sets: a set P_E of *environment ports*, a set P_T of *task ports*, and a set P_D of *driver ports*, updated respectively by the physical environment, by tasks, and by drivers. The environment ports include p_c, a *discrete clock*. An *input event* is a change of value at an environment or task port, say, at a sensor p_s. An input event is observed by the E machine through an event interrupt that can be characterized by a predicate, namely, $p'_s \neq p_s$, where p'_s refers to the current sensor reading, and p_s refers to the most recent previous sensor reading.

All information between the environment and the tasks flows through drivers: environment ports cannot be read by tasks, and task ports cannot be read by the environment. Formally, a *driver* d consists of a set $P[d] \subseteq P_D$ of driver ports, a set $I[d] \subseteq P_E \cup P_T$ of read environment and task ports, and a function $f[d]$ from $P[d] \cup I[d]$ states to $P[d]$ states. A *task* t consists of a set $P[t] \subseteq P_T$ of task ports, a set $I[t] \subseteq P_D$ of read driver ports, and a function $f[t]$ from $P[t] \cup I[t]$ states to $P[t]$ states. The E machine handles event interrupts through triggers. A *trigger* g consists of a set $P[g] \subseteq P_E \cup P_T$ of monitored environment and task ports, and a predicate $f[g]$, which evaluates to true or false over each pair (s, s') of $P[g]$ states. We require that $f[g]$ evaluates to false if $s = s'$. The state s is the state of the ports at the time instant when the trigger is *activated*. The state s' is the state of the ports at the time instant when the trigger is *evaluated*. All active triggers are logically evaluated with each event interrupt. An active trigger that evaluates to true is *enabled*, and may cause the E machine to execute E code. The trigger g is a *time trigger* if $P[g] = \{p_c\}$ and $f[g]$ has the form $p'_c = p_c + \delta$, for some positive integer $\delta \in \mathbb{N}_{>0}$. A time trigger monitors only the clock and specifies an *enabling time* δ, which is the number of clock ticks after activation before the trigger is enabled.

The E machine has three non-control-flow instructions. An *E instruction* is either $\texttt{call}(d)$, for a driver d; or $\texttt{schedule}(t)$, for a task t; or $\texttt{future}(g, a)$, for a trigger g and an address a. The $\texttt{call}(d)$ instruction invokes the driver d. The $\texttt{schedule}(t)$ instruction schedules the task t for execution by inserting it into the ready queue of the OS. The $\texttt{future}(g, a)$ instruction marks the E code at address a for possible execution at a future time when the trigger g becomes enabled. The E machine also has two *control-flow instructions*: the conditional jump instruction $\texttt{if}(f, a)$, where f is a predicate over the driver ports P_D, and a is the target address of the jump if f is true; and the termination instruction $\texttt{return}$, which ends the execution of E code. Formally, an *E program* consists of a set P of ports, a set D of drivers, a set T of tasks, a set G of triggers, a set A of addresses, an initial address $a_0 \in A$, and for each address $a \in A$, an E or control-flow instruction $ins(a)$, and a successor address $next(a)$. All sets that are part of an E program are finite. We require that E code execution always terminates, i.e., for each address $a \in A$ and all branches of $\texttt{if}$ instructions, a $\texttt{return}$ instruction must be reached in a finite number of steps. The E program is *time-triggered* if all triggers $g \in G$ are time triggers.

Example. We illustrate the semantics of E code using a simple program with two tasks, t_1 and t_2. The task t_2 is executed every 10 ms; it reads sensor values using a driver d_s, processes them, and writes its result to an interconnect driver d_i. The task t_1 is executed every 20 ms; it obtains values from driver d_i (the result of t_2), computes actuator values, and writes to an actuator driver d_a. There are two environment ports (the discrete clock p_c and a sensor p_s), two task ports (for the results of the two tasks), and three driver ports (the destinations of the drivers). The following time-triggered E program implements the above behavior:

```
a0: call(d_a)                         a1: call(d_s)
    call(d_s)                             schedule(t_2)
    call(d_i)                             future(p'_c = p_c + 10, a0)
    schedule(t_1)                         return
    schedule(t_2)
    future(p'_c = p_c + 10, a1)
    return
```

There are two blocks of E code; the block at a_0 is executed initially. The E machine processes each instruction in logical zero time. First, it calls the driver d_a

and waits until the execution of d_a is finished (in logical zero time), and then proceeds immediately to the next instruction. Once d_s and d_i have been called, all driver ports are updated. Then the E machine schedules the task t_1 by adding it to the ready queue of the operating system (which is initially empty). As we assume no particular scheduling scheme, we do not know the organization of the ready queue, and maintain the scheduled tasks as a set, called the *task set*. After inserting t_1 into the task set, the E machine immediately processes the next instruction and adds t_2 to the task set. Next, it proceeds to the `future` instruction, which creates a *trigger binding* $(p_c' = p_c + 10, a_1, s)$, where s is the current value of p_c, and appends it to a queue, called *trigger queue*, of active trigger bindings (initially empty). The trigger queue ensures that the E machine will execute the E code block at a_1 as soon as the trigger $p_c' = p_c + 10$ is enabled. For now the E machine proceeds to the `return` instruction. Since no active triggers are enabled, the E machine relinquishes control to the scheduler of the OS, which takes over to schedule the tasks t_1 and t_2 in the task set. The E machine wakes up again when an input event occurs that enables an active trigger. In particular, at 10 ms the trigger binding $(p_c' = p_c + 10, a_1, s)$ is removed from the trigger queue, and the E code at address a_1 is executed. The execution of block a_1 is similar to that of block a_0. The whole process repeats every 20 ms.

The above scenario assumes that the execution of a task has completed before it is scheduled again, in other words, we need that $w(t_1) + 2 \cdot w(t_2) \leq 20$, where $w(t)$ is the WCET of task t. This requirement must be checked by the compiler. We will see in the next section how this requirement can be derived automatically and checked statically.

Semantics. The execution of an E program yields an infinite sequence of program configurations, called *trace*. Each configuration tracks the values of all ports, the program counter, the task set, and the trigger queue. Formally, a (*program*) *configuration* $c = (s', a', \mathit{Trigs}, \mathit{Tasks})$ consists of (1) a P state s', called *port state*; (2) an address $a' \in A \cup \{\bot\}$, called *program counter*, where the special symbol $\bot$ indicates termination; (3) a queue *Trigs* of trigger bindings (g, a, s), called *trigger queue*, where g is a trigger, a is an address, and s is a $P[g]$ state; and (4) a set *Tasks* of triples (t, s, Δ), called *task set*, where t is a task, s is a $P[t] \cup I[t]$ state, and $\Delta \in \mathbb{N}$ is the *CPU time*, i.e., the amount of time that the task has run. We assume that CPU time is given in discrete units of the clock p_c, which may represent CPU cycles. The *ready tasks* of configuration c are defined as $T_c = \{t \mid \exists s, \Delta.\, (t, s, \Delta) \in \mathit{Tasks}\}$. The configuration c is *initial* if the program counter is the initial address ($a' = a_0$), and the trigger queue and task set are empty ($\mathit{Trigs} = \mathit{Tasks} = \emptyset$). A trigger binding (g, a, s) is *enabled* at c if the trigger predicate $f[g]$ evaluates to true over the pair (s, s') of $P[g]$ states. The configuration c is *input-enabling* if $a' = \bot$ and *Trigs* contains no enabled trigger bindings; otherwise, c is *input-disabling*.

The E machine runs as long as the program configuration is input-disabling. If the program counter a' is different from $\bot$, then the instruction $\mathit{ins}(a')$ is executed. This updates the current configuration $c = (s', a', \mathit{Trigs}, \mathit{Tasks})$ to a new configuration $\mathit{succ}(c)$ as follows. We only specify the parts of the configuration $succ(c)$ that are different from c: if $ins(a') = \mathtt{call}(d)$, then the new $P[d]$ state is $f[d](s'(P[d] \cup I[d]))$, and the new program counter is $next(a')$; if $ins(a') = \mathtt{schedule}(t)$, then the new task set is $Tasks \cup \{(t, s'(P[t] \cup I[t]), 0)\}$, and the new program counter is $next(a')$; if $ins(a') = \mathtt{future}(g, a)$, then the new trigger queue is $Enqueue(Trigs, (g, a, s'(P[g])))$, and the new program counter is $next(a')$; if $ins(a') = \mathtt{if}(f, a)$ and f evaluates to true (respectively, false) over s', then the new program counter is a (respectively, $next(a')$); if $ins(a') = \mathtt{return}$, then the new program counter is $\perp$. Note that the `call` instruction updates the port state, the `schedule` instruction updates the task set, and the `future` instruction updates the trigger queue. When a `return` instruction is reached, the program counter becomes $\perp$. Consider the configuration $c = (s', \perp, Trigs, Tasks)$. If some trigger binding in $Trigs$ is enabled at s', then let (g, a, s) be the first such binding, and define $succ(c)$ to be the configuration that differs from c in that the new program counter is a, and the new trigger queue results from $Trigs$ by removing (g, a, s). This leads to the execution of more instructions. If no trigger binding in $Trigs$ is enabled at s' (i.e., c is input-enabling), then event interrupts are enabled and the E machine relinquishes control of the CPU to the scheduler until a new input event enables an active trigger binding. While the E machine waits for an input event, scheduled computation can be performed.

A trace is a sequence of configurations such that from one configuration to the next, there is either an environment event, an elapse of one time unit possibly followed by a software event (i.e., the completion of a task), or the execution of an instruction of E code. An environment event causes a nondeterministic change in the values of some environment ports; the choice is up to the environment. A time elapse causes a nondeterministic change in the CPU time of some task; the choice of task is up to the scheduler. If the chosen task completes, it also causes a deterministic change in the value of the task ports according to the task function. The execution of E code causes a deterministic change as specified by the function *succ* defined above. Formally, given an E program Π with task set T, a *WCET map* for Π is a map $w\colon T \to \mathbb{N}_{>0}$ that assigns to each task a positive integer. A *trace* of the pair (Π, w) is a finite or infinite sequence of program configurations such that (1) the first configuration is initial and (2) for any two adjacent configurations c and c', one of the following holds:

(*Environment event*) c is input-enabling, and c' differs from c at most in the values of environment ports other than p_c. In this case, we write $e\text{-}step(c, c')$.

(*Time elapse with idle CPU*) c is input-enabling, and c' results from c by incrementing the clock p_c. In this case, we write $t\text{-}step(c, \emptyset, c')$.

(*Time elapse with used CPU*) c is input-enabling, and c' results from c by incrementing the clock p_c. In addition, there is a task t such that the task set of c contains a triple of the form (t, s, Δ), and either $\Delta + 1 < w(t)$ and the task set of c' results from c by replacing (t, s, Δ) with $(t, s, \Delta + 1)$; or $\Delta + 1 = w(t)$ and the task set of c' results from c by removing (t, s, Δ), and the $P[t]$ state of c' is $f[t](s)$. In this case, we write $t\text{-}step(c, t, c')$.

(*E code instruction*) c is input-disabling, and $c' = succ(c)$.

Note that we consider only traces where all task invocations consume their full WCETs; this is a worst-case assumption necessary for schedulability analysis. A *trace with atomic task execution* of Π is a sequence of configurations such that (1) the first configuration is initial and (2) for any two adjacent configurations c and c', either (*Environment event*); or (*E code instruction*); or (*Time elapse*) c is input-enabling, the task set of c is empty, and c' results from c by incrementing the clock p_c; or (*Task completion*) c is input-enabling, the task set of c contains a triple of the form $(t, s, 0)$, the task set of c' results from c by removing $(t, s, 0)$, and the $P[t]$ state of c' is $f[t](s)$. In a trace with atomic task execution, all tasks are executed in zero time.

3 Time-Safety Checking for E Code

An E program executes as intended only if the platform offers sufficient performance so that the computation of a task t always finishes before drivers access task ports of t, and before another invocation of t is scheduled. A trace that satisfies these conditions is called time safe, because the outcomes of `if` instructions cannot be distinguished from a trace with atomic task execution. Formally, a configuration c with program counter a is *time safe* [5] if either $a = \perp$, or for every ready tast $t \in T_c$, the instruction $ins(a)$ that is executed at c obeys the following two conditions: if $ins(a) = \texttt{call}(d)$, then $P[d] \cap I[t] = \emptyset$ and $I[d] \cap P[t] = \emptyset$; and if $ins(a) = \texttt{schedule}(t')$, then $P[t'] \cap P[t] = \emptyset$. If one of these two conditions is violated, then we say that the configuration c *conflicts with* the task t. A trace is *time safe* if it contains only time-safe configurations.

Given a nonempty finite trace τ, let $last(\tau)$ be the final configuration of τ. A *scheduling strategy* is a function that maps every nonempty finite trace τ whose final configuration $last(\tau)$ is input-enabling, either to $\emptyset$ (meaning that no task is scheduled), or to some ready task $t \in T_{last(\tau)}$. An infinite trace $\tau = c_0c_1c_2\ldots$ is an *outcome* of the scheduling strategy σ if for all nonempty finite prefixes $\tau' = c_0 \ldots c_j$ of τ, if c_j is input-enabling, then either $e\text{-}step(c_j, c_{j+1})$ or $t\text{-}step(c_j, \sigma(\tau'), c_{j+1})$. The E program Π is *schedulable* for the WCET map w if there exists a scheduling strategy σ such that all infinite traces of (Π, w) that are outcomes of σ are time safe. The *schedulability problem* for E code asks, given an E program Π and a WCET map w for Π, if Π is schedulable for w.

To solve the schedulability problem we need to eliminate some possible sources of infinity. An E program Π is *propositional* if it satisfies the following conditions: (1) all ports of Π except the clock p_c are boolean; (2) for all drivers d of Π, we have $p_c \notin I[d]$; and (3) for all triggers g of Π, either $p_c \notin P[g]$, or g is a time trigger. As time safety implies that a task must finish before it can be invoked again, along a time-safe trace, the size of the task set is always bounded by the number of tasks. However, the size of the trigger queue may grow unbounded [5]. A configuration c is *k-bounded*, for a positive integer $k \in \mathbb{N}_{>0}$, if the trigger queue of c contains at most k trigger bindings. A trace is *k-bounded* if it contains only k-bounded configurations. The *bounded schedulability* problem asks, given an E program Π, a WCET map w for Π, and a bound $k \in \mathbb{N}_{>0}$, if there is a scheduling strategy σ such that all infinite traces of (Π, w) that are outcomes of σ are both time safe and k-bounded.

Two-player safety games. Schedulability can be solved as a safety game on the configuration graph [1]. A *two-player safety game* [2] $\mathcal{G} = (V, E, V_0, U)$ consists of a finite set V of vertices, a relation $E \subseteq V \times V$ of edges, a set $V_0 \subseteq V$ of initial vertices, and a set $U \subseteq V$ of safe vertices. The vertices V are partitioned into V_1 and V_2. The game is turn-based and proceeds in rounds. For $i = 1, 2$, when the game is in $v \in V_i$, player i moves to v' such that $E(v, v')$. The goal of player 1 is to stay inside the safe set U. A *source-v_0 run* is an infinite sequence $v_0 v_1 v_2 \ldots$ of vertices in V such that $E(v_j, v_{j+1})$ for all $j \geq 0$. A *strategy* for player i, with $i = 1, 2$, is a function $f_i \colon V^* \times V_i \to V$ such that for every finite sequence $r \in V^*$ and vertex $v \in V_i$, we have $E(v, f_i(r, v))$. For a player-1 strategy f_1, a player-2 strategy f_2, and a state $v_0 \in V$, the *outcome* $\rho_{f_1,f_2}(v_0) = v_0 v_1 v_2 \ldots$ is a source-v_0 run such that for all $j \geq 0$, for $i = 1, 2$, if $v_j \in V_i$, then $E(v_j, f_i(v_0 \ldots v_{j-1}, v_j))$. The outcome $\rho_{f_1,f_2}(v_0)$ is *winning* for player 1 if $v_j \in U$ for all $j \geq 0$. The strategy f_1 is *winning* for player 1 if for all initial vertices $v_0 \in V_0$ and all player-2 strategies f_2, the outcome $\rho_{f_1,f_2}(v_0)$ is winning for player 1.

The schedulability game. Let Π be a propositional E program with ports P, tasks T, triggers G, and addresses A. Let w be a WCET map for Π, and let $k \in \mathbb{N}_{>0}$. We define the *schedulability game* $\mathcal{G}_{\Pi,w,k}$ as follows. Player 1 is the scheduler; it chooses during time elapses which task to run on the CPU. The environment is player 2; it chooses environment events. The actions of the E machine when executing E code are deterministic, so it does not matter to which player we attribute them; we choose player 2. From each configuration $c = (s', a', \mathit{Trigs}, \mathit{Tasks})$ of Π we obtain its *clock abstraction* $[c]$ as follows: remove the value of p_c from the port state s', and for each time trigger g, replace each trigger binding (g, a, s) in *Trigs* by $(g, a, s'(p_c) - s)$. Hence, for each active time trigger, the clock abstraction $[c]$ tracks only the number of clock ticks since the trigger was activated. A configuration c of Π *conforms with* w if $\Delta \leq w(t)$ for each triple (t, s, Δ) in the task set of c. Let $C_{\Pi,w,k}$ be the set of clock abstractions for all k-bounded configurations of Π that conform with w. As Π is propositional, the set $C_{\Pi,w,k}$ is finite. Specifically, the size of $C_{\Pi,w,k}$ is bounded by $2^{|P|} \cdot |A| \cdot (|T| \cdot 2^{|P|} \cdot \alpha) \cdot (|G| \cdot |A| \cdot 2^{|P|} \cdot \beta \cdot k)$, where $\alpha = \max\{w(t) \mid t \in T\}$, and β is the greatest enabling time for all time triggers in G.

The vertices of the schedulability game $\mathcal{G}_{\Pi,w,k}$ are $V = C_{\Pi,w,k} \times \{1, 2\}$, where the second component indicates which player can choose the next move, that is, the second component defines the subsets V_1 and V_2. The initial vertices are those of the form $((\cdot, a_0, \emptyset, \emptyset), 2)$, where a_0 is the initial address of Π. There are three types of edges. The environment chooses new values for the environment ports (other than p_c): for all configurations c and c' with *e-step*(c, c'), there are two edges $(([c], 2), ([c'], 2))$ and $(([c], 2), ([c'], 1))$. The E machine executes E code in input-disabling configurations: if c is input-disabling, then there is an edge $(([c], 2), ([\mathit{succ}(c)], 2))$. The scheduler assigns the CPU to tasks: for all configurations c and c' with *t-step*$(c, \cdot, c')$, there is an edge $(([c], 1), ([c'], 2))$. A vertex $([c], i)$, for $i = 1, 2$, is in the safe set U if the configuration c is time safe. The objective of the scheduler is to ensure that the game always remains in U. A winning strategy of the scheduler prescribes a scheduling strategy for the program Π that guarantees time safety and k-boundedness under the WCET assumption w, no matter what the environment does. Conversely, if the scheduler does not win, then by the determinacy of the safety game $\mathcal{G}_{\Pi,w,k}$, there is an environment strategy that forces the game into a time-safety or k-boundedness violation, no matter what the scheduler does. Since safety games can be solved in linear time [7], this gives an exponential upper bound for checking bounded schedulability.

Hardness. The bounded schedulability problem is hard for EXPTIME by a reduction from alternating linear-space Turing machines. Let M be an alternating Turing machine that uses $\ell \cdot n$ tape cells for inputs of length n, and makes binary existential and universal choices. For an input x of length n, we construct a propositional E program $\Pi_{M,x}$ (with the WCETs of all tasks being 1) such that $\Pi_{M,x}$ is two-bounded schedulable iff M does not accept x. For each control state q of M, and each tape-head position $h \in \{1, \ldots, \ell \cdot n\}$, the program has an address (q, h) that begins a block of E code. We have $\ell \cdot n$ task ports to keep the tape contents. The initial block of E code writes x followed by blanks to these ports, and jumps to $(q_0, 1)$, where q_0 is the initial control state of M. We encode the existential moves of M as choices made by the environment, that is, if q is an existential control state, then the E program reads an environment port and goes to one or the other successor configuration of M. For universal moves, the program schedules two tasks, each with WCET 1. Each task triggers an event upon completion. This event writes the identity of the task to a special port p_u. The scheduler chooses which task to run first, and this choice determines the order in which the tasks finish, and thus the value of p_u after 2 time units. The E program reads the new value of p_u and goes to the corresponding successor configuration of M. Every step of the simulation gives rise to a solvable scheduling problem. Finally, as soon as the Turing machine reaches an accepting state, the E program goes to an address that sets up an unsolvable scheduling problem. The trigger queue contains at any time at most two trigger bindings, and all numbers are bounded by a small constant.

Theorem 1. *Checking bounded schedulability for propositional E programs is complete for EXPTIME (even if all numbers are coded in unary).*

4 E Code Generation from Giotto

4.1 The Giotto Language

Giotto [4] is a programming language for time-triggered applications. Figure 1 shows an example of a Giotto program. The Giotto compiler generates time-triggered E code from a Giotto program. We demonstrate code generation based

```
sensor gps uses dev[gps]; toggle uses dev[toggle];
actuator servo uses dev[servo];
output
  ctrlOut := init[ctrlOut] uses copy[ctrlOut];
  filterOut := init[filterOut] uses copy[filterOut];

task control(ctrlIn) output (ctrlOut) private () {
  schedule task[control](ctrlIn, ctrlOut); }
task filter(filterIn) output (filterOut) private (filterState := init[filterState]) {
  schedule task[filter](filterIn, filterOut, filterState); }
task adaptiveFilter(filterIn) output (filterOut) private (adaptiveState := init[adaptiveState]) {
  schedule task[adaptiveFilter](filterIn, filterOut, adaptiveState); }

driver inputCtrl(filterOut) output (ctrlIn) {
  call driver[inputCtrl](filterOut, ctrlIn); }
driver inputFilter(gps) output (filterIn) { call driver[inputFilter](gps, filterIn); }
driver updateServo(ctrlOut) output (servo) { call driver[updateServo](ctrlOut, servo); }
driver switchFilter(toggle) output (ctrlOut, filterOut) {
  if condition[switchFilter](toggle) call driver[switchFilter](ctrlOut, filterOut); }

start normal {
  mode normal(ctrlOut, filterOut) period 6 {
    actfreq 1 do servo(updateServo);
    exitfreq 2 do adaptive(switchFilter);
    taskfreq 1 do control(inputCtrl);
    taskfreq 2 do filter(inputFilter); }
  mode adaptive(ctrlOut, filterOut) period 12 {
    actfreq 2 do servo(updateServo);
    exitfreq 3 do normal(switchFilter);
    taskfreq 2 do control(inputCtrl);
    taskfreq 3 do adaptiveFilter(inputFilter); }}
```

Fig. 1. A Giotto program with two modes

on the abstract syntax of a program, rather than its concrete syntax, and we use the program from Figure 1 as a running example.

A *Giotto program* begins with declarations of a set *SensePorts* of sensor ports, a set *ActPorts* of actuator ports, and a set *OutPorts* of task output ports. The set *Ports* of all program ports also includes the set *InPorts* of task input ports and the set *PrivPorts* of task private ports, which are declared for each task separately. A sensor or actuator port p requires the declaration of a device driver $dev[p]$. For example, the sensor port *gps* uses the device driver $dev[gps]$ to read new sensor values. A task output port p requires the declaration of an initialization driver $init[p]$ and a copy driver $copy[p]$. Each task output port is double-buffered, that is, it is implemented by two copies, a *local* copy that is used by the task only, and a *global* copy that is accessible to the rest of the program. The initialization driver initializes the local copy; the copy driver copies data from the local copy to the global copy. The second part of a Giotto program are the task declarations. A *Giotto task* t has a set $In[t] \subseteq InPorts$ of input ports, a set $Out[t] \subseteq OutPorts$ of output ports, a set $Priv[t] \subseteq PrivPorts$ of private ports, and a task function $task[t]$ from the input and private ports to the private and output ports. In the example, the task *filter* has an input port *filterIn*, an output port *filterOut*, a private port *filterState*, and the task function $task[filter]$. Private ports have initialization drivers similar to task output ports. Each task function is implemented as an E machine task. The third part of a Giotto program are the driver declarations. Giotto drivers transport data between ports and initiate mode changes. A *Giotto driver* d has a set $\mathit{Src}[d] \subseteq \mathit{Ports}$ of source ports, an optional driver guard $\mathit{condition}[d]$, which is evaluated on the source ports and returns a boolean, a set $\mathit{Dst}[d] \subseteq \mathit{Ports}$ of destination ports, and a driver function $\mathit{driver}[d]$ from the source to the destination ports. The driver guard is implemented as a branching condition of the E machine; the driver function, as an E machine driver. The driver *inputCtrl* is a *task driver* that transports data from the *filterOut* port to the input port *ctrlIn* of the *control* task. The driver *updateServo* is an *actuator driver* that updates the *servo* port with data from the *ctrlOut* port. The driver *switchFilter* is a *mode driver* that initiates a mode change whenever the driver guard $\mathit{condition}[\mathit{switchFilter}]$ evaluates to true. Moreover, the driver function $\mathit{driver}[\mathit{switchFilter}]$ may update the *ctrlOut* and *filterOut* ports when changing mode.

The final part of a Giotto program declares the set *Modes* of modes and the start mode $\mathit{start} \in \mathit{Modes}$. In the example, there are two modes, *normal* and *adaptive*, the former being the start mode. A *mode* m has a period $\pi[m] \in \mathbb{Q}$, a set $\mathit{ModePorts}[m] \subseteq \mathit{OutPorts}$ of mode ports, a set $\mathit{Invokes}[m]$ of task invocations, a set $\mathit{Updates}[m]$ of actuator updates, and a set $\mathit{Switches}[m]$ of mode switches. For example, the mode period $\pi[\mathit{normal}]$ of the *normal* mode is 6 ms, and its mode ports are *ctrlOut* and *filterOut*. For simplicity, we use milliseconds though another unit of time is possible. A *task invocation* $(\omega_{task}, t, d) \in \mathit{Invokes}[m]$ consists of a task frequency $\omega_{task} \in \mathbb{N}$ relative to the mode period, a task t, and a task driver d, which loads the task inputs. For example, the *normal* mode invokes the *control* task with the task driver *inputCtrl* and a period of 6 ms (i.e., once per mode period) as well as the *filter* task with the task driver *inputFilter* and a period of 3 ms. An *actuator update* $(\omega_{act}, d) \in \mathit{Updates}[m]$ consists of an actuator frequency $\omega_{act} \in \mathbb{N}$, and an actuator driver d. For example, the *normal* mode updates the *servo* port every 6 ms using the *updateServo* driver. A *mode switch* $(\omega_{switch}, m', d) \in \mathit{Switches}[m]$ consists of a mode-switch frequency $\omega_{switch} \in \mathbb{N}$, a target mode $m' \in \mathit{Modes}$, and a mode driver d, which governs the mode change. For example, the *normal* mode may change to the *adaptive* mode every 3 ms using the *switchFilter* driver.

A Giotto program is *well-timed* [4] if for all modes $m \in \mathit{Modes}$, all task invocations $(\omega_{task}, t, \cdot) \in \mathit{Invokes}[m]$, and all mode switches $(\omega_{switch}, m', \cdot) \in \mathit{Switches}[m]$, if $\omega_{task}/\omega_{switch} \notin \mathbb{N}$, then there exists a task invocation $(\omega'_{task}, t, \cdot) \in \mathit{Invokes}[m']$ with $\pi[m]/\omega_{task} = \pi[m']/\omega'_{task}$. Well-timedness ensures that mode changes do not terminate tasks: if a mode change occurs when a task may not be completed, then the same task must be present also in the target mode. The Giotto program in Figure 1 is well-timed, because the *control* task, which may be preempted by a mode change, runs in both modes with the same period. All non-well-timed Giotto programs are rejected as ill-formed. A complete list of syntactic criteria for the well-formedness of a Giotto program, such as the condition that the ports of different tasks be disjoint, is given in [4].

For a mode m, the least common multiple of the task, actuator, and mode-switch frequencies of m is called the number of *units* of m, and is denoted

$\omega_{max}[m]$. For example, $\omega_{max}[normal]$ is 2, and $\omega_{max}[adaptive]$ is 6. Thus a unit in the *normal* mode is equivalent to 3 ms, and a unit in the *adaptive* mode is equivalent to 2 ms. We use an integer $u \in \{0, \ldots, \omega_{max}[m] - 1\}$ as the *unit counter* for a mode m. Given a mode $m \in \mathit{Modes}$ and a unit u with $0 \leq u < \omega_{max}[m]$, we need the following auxiliary operators for E code generation:

$$
\begin{array}{ll}
\mathit{taskInvocations}(m,u) := & \{(\omega_{task}, t, d) \in \mathit{Invokes}[m] \mid u \cdot \omega_{task}/\omega_{max}[m] \in \mathbb{N}\} \\
\mathit{tasks}(m,u) := & \{t \mid (\cdot, t, \cdot) \in \mathit{taskInvocations}(m,u)\} \\
\mathit{taskDrivers}(m,u) := & \{d \mid (\cdot, \cdot, d) \in \mathit{taskInvocations}(m,u)\} \\
\mathit{taskOutputPorts}(m,u) := & \{p \mid t \in \mathit{tasks}(m,u) \wedge p \in \mathit{Out}[t]\} \\
\mathit{taskSensorPorts}(m,u) := & \{p \mid d \in \mathit{taskDrivers}(m,u) \wedge p \in \mathit{Src}[d] \cap \mathit{SensePorts}\} \\
\multicolumn{2}{l}{\mathit{preemptedTaskPeriods}(m,u) :=} \\
\multicolumn{2}{l}{\quad \{\omega_{max}[m]/\omega_{task} \mid (\omega_{task}, \cdot, \cdot) \in \mathit{Invokes}[m] \setminus \mathit{taskInvocations}(m,u)\}} \\
\mathit{actuatorDrivers}(m,u) := & \{d \mid (\omega_{act}, d) \in \mathit{Updates}[m] \wedge u \cdot \omega_{act}/\omega_{max}[m] \in \mathbb{N}\} \\
\mathit{actuatorPorts}(m,u) := & \{p \mid d \in \mathit{actuatorDrivers}(m,u) \wedge p \in \mathit{Dst}[d]\} \\
\multicolumn{2}{l}{\mathit{modeSwitches}(m,u) :=} \\
\multicolumn{2}{l}{\quad \{(m', d) \mid (\omega_{switch}, m', d) \in \mathit{Switches}[m] \wedge u \cdot \omega_{switch}/\omega_{max}[m] \in \mathbb{N}\}} \\
\multicolumn{2}{l}{\mathit{modeSensorPorts}(m,u) :=} \\
\multicolumn{2}{l}{\quad \{p \mid (\cdot, d) \in \mathit{modeSwitches}(m,u) \wedge p \in \mathit{Src}[d] \cap \mathit{SensePorts}\}}
\end{array}
$$

Consider the *taskInvocations* operator. For example, $\mathit{taskInvocations}(normal, 1)$ returns $\{(2, \mathit{filter}, \mathit{inputFilter})\}$, because the *filter* task is the only task that is invoked at unit 1 in the *normal* mode. At unit 0 the operator returns the invocations of both the *control* and the *filter* task. All other operators work in a similar way, except the *preemptedTaskPeriods* operator, which returns the periods of the tasks that are preempted, not invoked, at the specified unit.

4.2 From Giotto to E Code

Algorithm 1 generates E code that implements the logical semantics of a well-timed Giotto program, as it is specified in [4]. The command *emit* generates E code instructions. The key programmer's abstraction in Giotto is that the computation of a task takes exactly as long as its period. Thus the outputs of a task are logically made available at the end of its period, not at the end of its computation. The compiler begins generating E code by emitting `call` instructions to the initialization drivers of all task output and private ports. Then an absolute `jump` is emitted to the first instruction of the start mode.[1] Since this instruction is unknown at this point, we use a symbolic reference $\mathit{mode\_address}[m, u]$. The symbolic reference will be linked to the first instruction of the E code that implements mode m at unit u. Finally, Algorithm 2 is called to generate code for all modes and units. For the Giotto program from Figure 1, the following E code is generated by Algorithm 1:

$$
\begin{array}{l}
\texttt{call}(\mathit{init}[\mathit{ctrlOut}]) \\
\texttt{call}(\mathit{init}[\mathit{filterOut}]) \\
\texttt{call}(\mathit{init}[\mathit{filterState}]) \\
\texttt{call}(\mathit{init}[\mathit{adaptiveState}]) \\
\texttt{jump}(\mathit{mode\_address}[normal, 0])
\end{array}
$$

[1] The instruction $\texttt{jump}(a)$ is shorthand for $\texttt{if}(\mathit{true}, a)$.

```
∀p ∈ OutPorts ∪ PrivPorts: emit(call(init[p]));
emit(jump(mode_address[start, 0]));
∀m ∈ Modes: invoke Algorithm 2 for mode m;
```

Algorithm 1: The Giotto program compiler

```
u := 0; γ := π[m]/ω_max[m];
while u < ω_max[m] do
  link mode_address[m, u] to the address of the next free instruction cell;
  ∀p ∈ taskOutputPorts(m, u): emit(call(copy[p]));
  ∀d ∈ actuatorDrivers(m, u): emit(call(driver[d]));
  ∀p ∈ actuatorPorts(m, u): emit(call(dev[p]));
  ∀p ∈ modeSensorPorts(m, u): emit(call(dev[p]));
  ∀(m′, d) ∈ modeSwitches(m, u): emit(if(condition[d], switch_address[m, u, m′, d]));
  emit(jump(task_address[m, u]));

  ∀(m′, d) ∈ modeSwitches(m, u):
    link switch_address[m, u, m′, d] to the address of the next free instruction cell;
    // compute the unit u′ to which to jump in the target mode m′ and
    // compute the time δ′ before new tasks in the target mode m′ can be scheduled
    if preemptedTaskPeriods(m, u) = ∅ then
      // jump to the beginning of m′ if all tasks in mode m are completed
      δ′ := 0; u′ := 0;
    else
      // compute the hyperperiod h of the preempted tasks in units of mode m
      h := lcm(preemptedTaskPeriods(m, u));
      // compute the time δ to finish the hyperperiod h
      δ := (h − u mod h) ∗ π[m]/ω_max[m];
      // compute the time δ′ to wait for the unit u′ in the target mode m′ to begin
      δ′ := δ mod (π[m′]/ω_max[m′]);
      // compute the closest unit u′ to the end of the mode period in m′ after δ′
      u′ := (ω_max[m′] − (δ − δ′) ∗ ω_max[m′]/π[m′]) mod ω_max[m′];
    end if
    emit(call(driver[d]));
    if δ′ > 0 then
      emit(future(timer[δ′], mode_address[m′, u′]));
      emit(return);
    else
      emit(jump(task_address[m′, u′]));
    end if

  link task_address[m, u] to the address of the next free instruction cell;
  ∀p ∈ taskSensorPorts(m, u): emit(call(dev[p]));
  ∀d ∈ taskDrivers(m, u): emit(call(driver[d]));
  ∀t ∈ tasks(m, u): emit(schedule(task[t]));
  emit(future(timer[γ], mode_address[m, u + 1 mod ω_max[m]]));
  emit(return);
  u := u + 1;
end while
```

Algorithm 2: The Giotto mode compiler

Algorithm 2 generates three types of E code blocks for each unit u of a mode m. The duration of a unit is denoted by γ. The first type of E code block, labeled $mode\_address[m, u]$, takes care of updating task output ports, updating actuators, reading sensors, and checking mode switches. The compiler generates `call` instructions to the appropriate drivers and an `if` instruction for each mode switch. The block is terminated by a `jump` instruction to a block that deals with task invocations; see below. The `jump` is only reached if none of the mode switches is enabled. The second type of E code block implements the mode change to a target mode m' with a given mode driver d. We use the symbolic reference $switch\_address[m, u, m', d]$ to label this type of E code block. Upon a mode change, the mode driver is called and then control is transfered to the appropriate E code block of the target mode. The compiler computes the destination unit u' as close as possible to the end of the target mode's period. We distinguish the two cases of whether the duration of either u or u' is a multiple of the other, or not. If so, then the compiler generates a `jump` instruction to the E code of the target mode for u'. If not, then the time δ' to wait for u' is computed and generated as part of a `future` instruction. The trigger $timer[\delta']$ is a time trigger with enabling time δ'; that is, it specifies the trigger predicate $p'_c = p_c + \delta'$, which evaluates to true after δ' ms elapse. The third type of E code blocks handles the invocation of tasks and the future invocation of the E machine for the next unit. The label for these blocks is $task\_address[m, u]$. Before scheduling the tasks, the task drivers are called in order to load the task input ports with new data. The final `future` instruction makes the E machine wait for the duration γ of u and then execute the E code for the next unit. Note that the resulting E code is time-triggered. If, along a trace, the E code generated for m and u is executed consecutively without mode change for all units u with $0 \leq u < \omega_{max}[m]$, then the trace contains a *full period* of mode m.

Figure 2 shows the E code generated for the *normal* mode of the Giotto program from Figure 1. The E code for the *adaptive* mode is not shown. The *normal* mode has a period of 6 ms and two units of 3 ms length each. At the 0 ms unit, the *control* and *filter* tasks are invoked; at the 3 ms unit, only the *filter* task is invoked. The mode switch to the *adaptive* mode is checked at both units. Consider the mode switch at unit 1 when the *control* task is preempted after 3 ms of logical computation time. In Algorithm 2 the number h of units in the *normal* mode until the task completes is 1. Thus the time δ to complete the task is 3 ms. The time δ' to wait for the next unit u' in the *adaptive* mode is 1 ms, because the duration of a unit in the *adaptive* mode is 2 ms. The closest unit u' to the end of the *adaptive* mode's period is 5. Within one more unit of 2 ms the end of the period will be reached. Thus the *control* task has exactly 3 ms time to complete even when the mode is changed to the *adaptive* mode.

5 Time-Safety Checking for Giotto

For special classes of E programs, schedulability can be checked efficiently. For example, a set T of *periodic* tasks can be scheduled iff it satisfies the utilization

```
mode_address[normal, 0]:
call(copy[ctrlOut])
call(copy[filterOut])
call(driver[updateServo])
call(dev[servo])
call(dev[toggle])
if(condition[switchFilter], switch_address[
    normal, 0, adaptive, switchFilter])
jump(task_address[normal, 0])
```

```
mode_address[normal, 1]:
call(copy[filterOut])
call(dev[toggle])
if(condition[switchFilter], switch_address[
    normal, 1, adaptive, switchFilter])
jump(task_address[normal, 1])
```

```
switch_address[normal, 0, adaptive, switchFilter]:
call(driver[switchFilter])
jump(task_address[adaptive, 0])
```

```
switch_address[normal, 1, adaptive, switchFilter]:
call(driver[switchFilter])
future(timer[1], mode_address[adaptive, 5])
return
```

```
task_address[normal, 0]:
call(dev[gps])
call(driver[inputCtrl])
call(driver[inputFilter])
schedule(task[control])
schedule(task[filter])
future(timer[3], mode_address[normal, 1])
return
```

```
task_address[normal, 1]:
call(dev[gps])
call(driver[inputFilter])
schedule(task[filter])
future(timer[3], mode_address[normal, 0])
return
```

Fig. 2. E code generated for the *normal* mode of the Giotto program from Figure 1

test, i.e., the processor utilization $\sum_{t \in T} w(t)/\pi(t)$, where $w(t)$ is the WCET time and $\pi(t)$ is the period of task t, is less than or equal to 1 [6]. For E programs that are generated from typical Giotto source programs —namely, Giotto programs where each mode may be executed for a full period— we have a similarly simple schedulability test. More precisely, a mode m of a Giotto program G is *fully reachable* if there exists a trace with atomic task execution of Π_G that contains a full period of mode m. Note that the definition of full reachability does not depend on time safety, as it assumes atomic task execution. Given a (well-timed) Giotto program G, we write Π_G for the E program that is generated from G by Algorithm 1. The Giotto program G is *E schedulable* for the WCET map w if the E program Π_G is schedulable for w. The *E schedulability problem* for Giotto asks, given a Giotto program G and a WCET map w for Π_G, if G is E schedulable for w.

Theorem 2. *A Giotto program G is E schedulable for the WCET map w if for each mode m of G, the utilization equation $\sum_{(\omega_{task}, t, \cdot) \in Invokes[m]} w(t)/\pi(t) \leq 1$ holds, where $\pi(t) = \pi[m]/\omega_{task}$ is the period of task t. This condition is necessary if each mode of G is fully reachable.*

Assuming that all modes are fully reachable, Theorem 2 gives a polynomial-time algorithm for checking the E schedulability of a Giotto program: the utilization equation can be checked in time no more than $O(n^2 polylog(n))$, where n is the size of the Giotto program. This test shows, for example, that the Giotto program from Figure 1 is E schedulable for the WCETs $w(control) = 3$, $w(filter) = 1.5$, and $w(adaptiveFilter) = 2$. Note that E code schedulability is not the most general notion of schedulability for Giotto programs. While E code schedulability refers to the particular E code generation scheme specified by Algorithm 1, for any given Giotto program, there may be other, more flexible code generation strategies that are faithful to the Giotto semantics. For example, a task could be started as soon as all arguments are available, which may be before its logical release time.

Proof sketch. Suppose that each mode satisfies the utilization test. This means that each mode is schedulable by itself. We show that mode changes do not cause the system to become non-schedulable, by using the schedule that assigns to each ready task t a time slice equal to $w(t)/\pi(t)$ in every time unit. This is possible by the utilization equation. Now consider a mode change. By the semantics of a mode change in Giotto, in particular by the well-timedness condition, at the mode change, all tasks that are not present in the target mode have finished executing, and all tasks that are present in the target mode retain their deadline after the mode change. This implies that the common tasks have executed for exactly the same amount of time in the source mode up to the mode change as they would have, following the chosen schedule, in an execution that started in the target mode. Since the target mode can finish executing all tasks by their deadlines, it can do so even after the mode change. In fact, the only difference of an execution in the target mode after the mode change from an execution starting in the target mode is that some tasks in the target mode may not run in the former case. Since the target mode is individually schedulable, it remains schedulable after a mode change into it.

For the converse, suppose that the utilization test fails for a fully reachable mode m. Consider a trace τ with atomic task execution that contains a full period of m. Choose any scheduling strategy σ. If the environment plays against σ using the same behavior as used along τ in a game with WCET task execution, then either a time-safety violation occurs, or a full period of m is reached. In the latter case, time safety must be violated also, because of the failed utilization test. It follows that the program is not E schedulable.

EDF scheduling. For each configuration c and ready task $t \in T_c$, we define the *deadline* $D(c,t)$ as the weight of the shortest path in the (unbounded) game graph $\mathcal{G}_{\Pi,w,\infty}$ from $[c]$ to the clock abstraction of a configuration that conflicts with t, where each player 1 (scheduler) move has weight 1, and all other moves have weight 0; if no such path exists, then $D(c,t) = \infty$. An *EDF scheduling strategy* is a function that maps every nonempty finite trace τ to $\emptyset$ if the task set of the final configuration $last(\tau)$ is empty, and otherwise to a task $t \in T_{last(\tau)}$ such that for all ready tasks $t' \in T_{last(\tau)}$, we have $D(c,t) \leq D(c,t')$. The E programs that are generated from Giotto source programs using Algorithm 1 have the property that with the execution of a `schedule`(t) instruction, the deadline of t is known and does not change in subsequent configurations until t completes. In particular, all deadlines are independent of the chosen scheduling strategy. It follows by a standard argument that EDF is optimal for Giotto.

Proposition 1. *Let σ be an EDF scheduling strategy. A Giotto program G is E schedulable for the WCET map w iff all infinite traces of (Π_G, w) that are outcomes of σ are time safe.*

The deadlines for an EDF scheduler can be computed directly on the Giotto source and may be passed to the scheduler as E code annotations [5]. The Giotto compiler we have implemented proceeds in two steps. First, it computes the relative deadline for each task in each mode and checks the E schedulability of the Giotto program by performing the utilization test of Theorem 2 for each mode. Second, it generates E code using Algorithm 1 and annotates each `schedule`(t) instruction with the deadline of t. Then, for an EDF scheduler that uses the deadlines, time safety is guaranteed for all possible environment behaviors.

Checking E schedulability. If there are modes that are not fully reachable, then the utilization test is only a sufficient condition for E schedulability. In general, however, it is PSPACE-hard to check if a mode is fully reachable, or if a Giotto program is E schedulable. We can reduce succinct boolean reachability to the reachability of a boolean port state, say s_f, in the start mode (using a driver to encode the transition relation). The start mode is schedulable, but if s_f is reached, the program switches to a non-schedulable mode. Then the Giotto program is E schedulable iff s_f is reachable. This gives half of the theorem below. For *propositional* Giotto programs, which have only boolean ports, inclusion in PSPACE follows from the following scheduling algorithm. First observe that if G is propositional, then so is Π_G. From Algorithm 1 it follows also that all traces of Π_G are one-bounded. Moreover, the number of E code instructions of Π_G is exponential in the size of the Giotto program G. By Proposition 1, we can fix an EDF scheduling strategy and check if a non-time-safe configuration can be reached on the game graph $\mathcal{G}_{\Pi_G,w,1}$. This is a reachability, rather than a game problem, on an exponential graph, and therefore in PSPACE.

Theorem 3. *Checking E schedulability for propositional Giotto programs is complete for PSPACE (even if all numbers are coded in unary).*

References

1. K. Altisen, G. Gössler, A. Pnueli, J. Sifakis, S. Tripakis, and S. Yovine. A framework for scheduler synthesis. In *Proc. Real-Time Systems Symp.*, pp. 154–163. IEEE Computer Society, 1999.
2. J.R. Büchi and L.H. Landweber. Solving sequential conditions by finite-state strategies. *Trans. AMS*, 138:295–311, 1969.
3. N. Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer, 1993.
4. T.A. Henzinger, B. Horowitz, and C.M. Kirsch. Giotto: a time-triggered language for embedded programming. In *Proc. Embedded Software*, LNCS 2211, pp. 166–184. Springer, 2001.
5. T.A. Henzinger and C.M. Kirsch. The embedded machine: predictable, portable real-time code. In *Proc. Conf. Programming Languages Design and Implementation*, pp. 315–326. ACM, 2002.
6. C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20:46–61, 1973.
7. J.W. Thatcher and J.B. Wright. Generalized finite-automata theory with an application to a decision problem in second-order logic. *Mathematical Systems Theory*, 2:57–81, 1968.