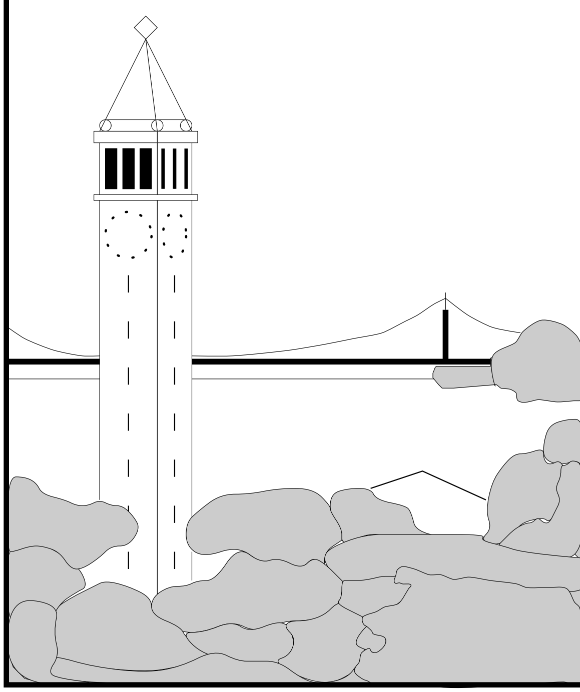
Schedule-Carrying Code

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



Report No. UCB//CSD-3-1230

February 2003

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Schedule-Carrying Code *

Thomas A. Henzinger, Christoph M. Kirsch, and Slobodan Matic
{tah,cm,matic}@eecs.berkeley.edu

February 2003

Abstract

The interactions of real-time tasks with each other and with the environment can be specified in a platform-independent machine language called E code. E code is *time safe* if it can be scheduled on a given platform so that all its timing constraints are met. For specifying static, dynamic, and conditional schedules, we propose again an executable machine language, called S code. A compiler for real-time programs, then, consists of a platform-independent and a platform-dependent part. The former produces E code; the latter generates S code that ensures the time-safe execution of the E code. The run-time system consists of an implementation of the E machine, which interprets E code that manages interrupts from the environment, and of the S machine, which interprets S code that manages task execution on the processors.

Generating nonpreemptive schedules for periodic tasks is NP-hard. However, for E code that specifies periodic tasks, and S code that specifies a corresponding nonpreemptive schedule, we show that time safety can be checked in linear time. This suggests the notion of *schedule-carrying code* (SCC), where E code is annotated with S code before being sent to an execution host. The host, if it does not trust the sender, can then check the time safety of the code at a cost that is far below the cost of generating a feasible schedule.

1 Introduction

Schedule-carrying code (SCC) is real-time code annotated with the description of a schedule that witnesses the *schedulability* of the real-time code. Schedulability of a real-time program is the existence of a scheduler that guarantees the *time safety* of all executions of the program. Intuitively, the execution of a real-time program is time-safe if all time-critical components of the program execute according to their timing constraints. The schedule in SCC is a witness of time safety. We will argue that, while generating SCC from real-time code may be difficult for non-trivial scheduling strategies such as non-preemptive or multiprocessor scheduling, checking time safety of SCC can be easy. As a consequence, SCC can be generated at compile-time when speed is not of primary concern while the validity of the result can later be verified at runtime prior to the program execution in order to obtain more confidence in the temporal correctness of the code.

We propose the following format for SCC: (1) the real-time code portion of SCC is *E code* of the *Embedded Machine* [2] and (2) the schedule portion of SCC is *S code* of the *Scheduling Machine*, which we will introduce here. E code is *timing code* that determines the invocation of (software) *tasks* with respect to the occurrences of events such as clock ticks. A task is a sequential program, e.g., a C procedure, without any internal synchronization mechanisms. A task is preemptable but

*This research was supported in part by the DARPA SEC grant F33615-C-98-3614, the AFOSR MURI grant F49620-00-1-0327, the California MICRO grant 01-037, and the NSF grants CCR-0208875 and CCR-0225610.

has its own memory space. Tasks always operate on mutually disjoint data. E code invokes *drivers* to transport data among tasks as well as from tasks to the environment of the system. Similar to a task, a driver is a sequential program but, unlike a task, is not preemptable and may operate on any memory. Intuitively, the execution of E code is time-safe if a driver that shares data with a task is never invoked when that task has already been invoked but not yet completed. S code is *scheduling code* that determines the order in which multiple tasks invoked by the Embedded Machine execute. The purpose of S code is to guarantee the time-safe execution of E code.

In Section 2, we describe the Embedded Machine and define the semantics of E code. The Scheduling Machine and the semantics of S code is introduced in Section 3. The semantics of Schedule-Carrying Code is defined in Section 4. In Section 5, we show that the schedulability problem for E code that describes an arbitrary set of priodic tasks is NP-hard when using a non-preemptive scheduling strategy. Then we show that the schedulability of SCC generated from a successful non-preemptive schedulability test of that E code can be checked in linear time in the size of the E code.

2 The Embedded Machine

The E machine [2] 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 *triggers* 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 [2]; they are *time-live*, that is, all synchronous computation is guaranteed to terminate.

E Code 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. A change of value of the discrete clock is also called a *clock tick*. We also say that a change of values at environment (task) ports constitutes an *environment* (*software*) *event*. 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 port reading, and p_s refers to the most recent previous port reading. $P_G \subseteq P_E \cup P_T$ denotes the environment and task ports that are observed by event interrupts. We call the ports in P_G *trigger ports*.

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 $P[g] \subseteq P_G$ 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 $\mathtt{call}(d)$, for a driver d; or $\mathtt{schedule}(t)$, for a task t; or $\mathtt{future}(g, a)$, for a trigger g and an E code address a. The $\mathtt{call}(d)$ instruction invokes the driver d. The $\mathtt{schedule}(t)$ instruction schedules the task t by handing t off to a *task scheduler* that dispatches the scheduled tasks to execute in some order after the E machine is finished. The task scheduler could be the scheduler of an operating system. The $\mathtt{future}(g, a)$ 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 $\mathtt{if}(f, a)$, 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 `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 E code addresses, an initial E code address $a_0 \in A$, and for each E code address $a \in A$, an E or control-flow instruction *Instruction*(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 E code address $a \in A$ and all branches of `if` instructions, a `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.

E Code 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:

a_0:	$\mathtt{call}(d_a)$	a_1:	$\mathtt{call}(d_s)$
	$\mathtt{call}(d_s)$		$\mathtt{schedule}(t_2)$
	$\mathtt{call}(d_i)$		$\mathtt{future}(p'_c = p_c + 10, a_0)$
	$\mathtt{schedule}(t_1)$		`return`
	$\mathtt{schedule}(t_2)$		
	$\mathtt{future}(p'_c = p_c + 10, a_1)$		
	`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 a *task invocation* N of the form $((t_1, a[t_1], 0), \perp)$ to the so-called *task set*, which is initially empty. $a[t_1]$ denotes the *initial program counter* of t_1. 0 is the amount of soft time for which t_1 already executed. The $\perp$ element is not important here and will be explained later. As we assume no particular scheduling scheme, we do not know the organization of the task set. If we were to use the scheduler of an operating system the task set could be represented by the ready queue of the operating system. 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* B of the form $(p_c' = p_c + 10, a_1, s)$, 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 task scheduler, 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 $wcet(t_1) + 2 \cdot wcet(t_2) \leq 20$, where $wcet(t)$ is the WCET of task t. This requirement must be checked by the compiler [3].

```
loop
  invoke task dispatcher (Algorithm 2)
  invoke E code interpreter (Algorithm 3)
  invoke task scheduler (Algorithm 4)
end loop
```

Algorithm 1: The Embedded Machine

E Code 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 trigger queue, the task set, and the running task. An *E program configuration* consists of (1) a P state s', called *port state*; (2) a queue of *trigger bindings* (g, a, s), called *trigger queue*, where g is a trigger, $a \in A$ is an E code address, and s is a $P[g]$ state; (3) a set of *task invocations* $((t, a_t, \delta), \perp)$, called *task set*, where t is a task, a_t is the program counter of t, and δ is the amount of soft time for which t already executed (we call (t, a_t, δ) a *task instance* of t); and (4) a *running task* R, where R is either $\perp$, or else of the form $(N, \perp)$ where N is either $\perp$ or a task instance. A trigger binding (g, a, s) is *enabled* if the trigger predicate $p[g]$ evaluates to true over the pair (s, s') of $P[g]$ states. The configuration c is *schedule-enabled* if the running task of c is $\perp$; otherwise, c is *schedule-disabled*. c is *input-enabling* if c is schedule-disabled and the trigger queue contains no enabled trigger bindings; otherwise, c is *input-disabling*. We call an input-enabling c *idling* if the running task of c is of the form $(\perp, \cdot)$.

We define the semantics of E code operationally using a pseudo-code description of the E machine. Algorithm 1 shows the main loop of the machine as it executes a given E program. After entering the main loop, the machine invokes the *task dispatcher* (Algorithm 2) to dispatch the task that has been chosen by the task scheduler to execute. Since no task is chosen initially, the task dispatcher enables the event interrupts and then waits for environment events. The occurrence of an environment event wakes up the machine, which immediately disables all event interrupts (thus it is still possible for low-level interrupts to preempt the machine, as long as they do not interfere with the triggering mechanism of the machine). Then the E machine interpreter (Algorithm 3) is invoked.

The E machine interpreter runs through the outer while loop of Algorithm 3 as long as there are enabled trigger bindings in the trigger queue, each time executing a block of E code that is bound to an enabled trigger. Initially, the trigger queue contains a single trigger binding $(true, a_0, \emptyset)$ where

a_0 is the initial E code address of the E program, and the task set is empty. In the inner while loop of Algorithm 3, the machine fetches the current instruction from the program, decodes and executes the instruction, and then determines the address of the next instruction.

After the interpreter is finished executing E code, the task scheduler (Algorithm 4) is invoked to choose a task from the so-called *ready set* to execute. Here the ready set is always equivalent to the task set. The task scheduler $Schedule(ReadySet)$ is free to choose any scheduling scheme including an idling scheme where no task is chosen although the task set is not empty. The chosen task becomes the *running task*, which is dispatched by the task dispatcher to execute until an input event occurs. Then the task is put back into the task set with its new program counter only when the task has not yet completed. However, if the task scheduler chooses to idle a running task of the form $(\perp, \cdot)$ is returned and no task is dispatched.

```
if RunningTask = (⊥, ·) then
  enable event interrupts
  (δ, Γ) := WaitForEnvironmentEvents()
  disable event interrupts
  if Γ = PortState(P_E \ p_c) then
    PortState(p_c) := PortState(p_c) + δ
    // Configuration: Idle Time Tick
  else
    PortState(P_E \ p_c) := Γ
    // Configuration: Environment Event
  end if
else
  ((t, a_t, δ), B) := RunningTask
  enable event interrupts
  (a'_t, δ', PortState(P[t])) := Dispatch(t, a_t, PortState(P[t] ∪ I[t]))
  disable event interrupts
  PortState(p_c) := PortState(p_c) + δ'
  if a'_t = ⊥ then
    if B = (g, a_s, s) then
      TaskSet := TaskSet ∪ {(⊥, (true, a_s, s))}
    end if
    // Configuration: Task Completion
  else
    TaskSet := TaskSet ∪ {((t, a'_t, δ + δ'), B)}
    // Configuration: Task Preemption
  end if
end if
```

Algorithm 2: The Task Dispatcher

An *initial configuration* of an E program Π consists of (1) a P state; (2) the trigger queue containing a single trigger binding $(true, a_0, \emptyset)$ where a_0 is the initial E code address of Π; (3) an empty task set; and (4) the running task set to $(\perp, \perp)$. A *trace* of Π 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 idling, and c' differs from c at most in the values of

while there is an enabled trigger in $\mathit{TriggerQueue}$ **do**
  $(g, a_e, s) := \mathit{GetFirstEnabledTriggerBinding}(\mathit{TriggerQueue})$
  $\mathit{TriggerQueue} := \mathit{RemoveFirstEnabledTriggerBinding}(\mathit{TriggerQueue})$
  $\mathit{ProgramCounter} := a_e$
  while $\mathit{ProgramCounter} \neq \perp$ **do**
    $i := \mathit{Instruction}(\mathit{ProgramCounter})$
    if $\texttt{call}(d) = i$ **then**
      $\mathit{PortState}(P[d]) := f[d](\mathit{PortState}(P[d] \cup I[d]))$
    else if $\texttt{schedule}(t) = i$ **then**
      $\mathit{TaskSet} := \mathit{TaskSet} \cup \{((t, a[t], 0), \perp)\}$
    else if $\texttt{future}(g, a) = i$ **then**
      $\mathit{TriggerQueue} := \mathit{TriggerQueue} \circ (g, a, \mathit{PortState}(P[g]))$
    end if
    $\mathit{ProgramCounter} := \mathit{Next}(\mathit{ProgramCounter})$
  end while
end while
$\mathit{RunningTask} := \perp$
// Configuration: E Code Execution

Algorithm 3: The E Code Interpreter

$\mathit{ReadySet} := \mathit{TaskSet} \cap \{(N, B) | \forall$ task instances $N \wedge \forall$ trigger bindings B with $B \neq \perp\}$
if $\mathit{ReadySet} = \emptyset$ **then**
  $\mathit{ReadySet} := \mathit{TaskSet}$
end if
$\mathit{RunningTask} := \mathit{Schedule}(\mathit{ReadySet})$
if $\mathit{RunningTask} \neq (\perp, \cdot)$ **then**
  $\mathit{TaskSet} := \mathit{TaskSet} \setminus \{\mathit{RunningTask}\}$
end if
// Configuration: Task Scheduling

Algorithm 4: The Task Scheduler

environment ports other than p_c. In this case, we write $e\text{-}step(c, c')$.

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

(*Task Completion*) c is input-enabling and a running task of the form $((t, a_t, \delta), \perp)$ is scheduled in c, and c' results from c by incrementing the clock p_c by some amount of time δ'. In addition, the execution of t results in the program counter $\perp$ (task completion) and in a new $P[t]$ state of c'. In this case, we write $t\text{-}step(c, t, c')$.

(*Task Preemption*) c is input-enabling and a running task of the form $((t, a_t, \delta), \perp)$ is scheduled in c, and c' results from c by incrementing the clock p_c by some amount of time δ'. In addition, the execution of t results in a new program counter a'_t of t that is different from $\perp$ and in a new $P[t]$ state of c', and the task set of c' results from c by adding $((t, a'_t, \delta + \delta'), \perp)$ to the task set. In this case, we write $t\text{-}step(c, t, c')$.

(*E code Execution*) c is input-disabling and schedule-disabled, and c' results from invoking the E machine interpreter (Algorithm 3) on c.

(*Task Scheduling*) c is schedule-enabled, and c' results from invoking the task scheduler (Algorithm 4) on c.

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 (*Idle Time Tick*); or (*Task Completion*); or (*E code Execution*); or (*Task Scheduling*). In a trace with atomic task execution, all tasks are executed in zero time without any task preemption.

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 *time safe* [2] if, for every task t in the task set of c and for every instruction $Instruction(a)$ that is executed at c, the following two conditions are obeyed: if $Instruction(a) = \texttt{call}(d)$, then $P[d] \cap I[t] = \emptyset$ and $I[d] \cap P[t] = \emptyset$; and if $Instruction(a) = \texttt{schedule}(t')$, 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 $\tau = c_0 c_1 c_2 \ldots$ 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 *wcet* if there exists a scheduling strategy σ such that all infinite traces of $(\Pi, wcet)$ that are outcomes of σ are time safe. The *schedulability problem* for E code asks, given an E program Π and a WCET map *wcet* for Π, if Π is schedulable for *wcet*.

3 The Scheduling Machine

The *Scheduling Machine* (S machine) is a virtual machine that schedules software processes to execute according to an *S code* program. S code consists of instructions to execute a task (or to idle) until a physical event, such as a clock tick, or a software event, such as the task completion, occurs.

S Code Syntax. The S machine schedules the execution of tasks. The S machine has five non-control-flow instructions. An *S instruction* is either `call`(d), for a driver d; or `schedule`(t), for a task t; or `dispatch`(t, g), for a task t and a trigger g; or `idle`(g), for a trigger g; or `fork`(a) for an S code address a. The `call` and `schedule` instructions are equivalent to the corresponding E instructions of the E machine. The `dispatch`(t, g) instruction dispatches the task t to execute until either t completes or the trigger g is enabled, whatever comes first. The S machine yields to t after executing the `dispatch`(t, g) instruction. The `idle`(g) instruction idles the S machine until the trigger g is enabled. The `fork`(a) instruction marks the S code at address a for execution in parallel to the current control flow. The S machine has only a single *control-flow instruction*: the termination instruction `return`, which ends the execution of S code. The S machine may have other control-flow instructions such as a conditional jump instruction in order to describe dynamic scheduling schemes. Without any additional control-flow instructions we call S code *static* since it can only describe static schedules. In this paper, we focus on static S code. Formally, an *S 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 S code addresses, an initial S code address $a_0 \in A$, and for each S code address $a \in A$, an S or control-flow instruction *Instruction*(a), and a successor address *Next*(a). All sets that are part of an S program are finite. We require that S code execution always eventually yields or terminates, i.e., for each S code address $a \in A$, either a `dispatch` or an `idle` instruction, or else a `return` instruction must be reached in a finite number of steps. The S program is *time-triggered* if all triggers $g \in G$ are time triggers and if all instructions that immediately precede a `fork` instruction are `idle` instructions.

S Code Example. We illustrate the semantics of S code using the program of Section 2 with the two tasks t_1 and t_2. Recall that the task t_2 is executed every 10 ms whereas the task t_1 is executed every 20 ms. The following time-triggered S program implements this behavior, which is equivalent to the behavior of the E program in Section 2:

$$
\begin{array}{ll}
a_0: & \texttt{call}(d_a) \\
 & \texttt{call}(d_s) \\
 & \texttt{call}(d_i) \\
 & \texttt{schedule}(t_1) \\
 & \texttt{schedule}(t_2) \\
 & \texttt{dispatch}(t_2) \\
a_1: & \texttt{dispatch}(t_1) \\
 & \texttt{idle}(p'_c = p_c + 10) \\
a_2: & \texttt{call}(d_s) \\
 & \texttt{schedule}(t_2) \\
 & \texttt{dispatch}(t_2) \\
 & \texttt{idle}(p'_c = p_c + 20) \\
 & \texttt{fork}(a_0) \\
 & \texttt{return}
\end{array}
$$

There is one block of S code with the initial S code address a_0. We also call a block of S code a *thread*. The S machine processes each instruction in logical zero time. The first five instructions are executed in the same way the E machine executes them. Then, the S machine proceeds to the first

$\texttt{dispatch}(t_2)$ instruction, which replaces the task invocation $((t_2, a[t_2], 0), \perp)$ created by the preceding $\texttt{schedule}(t_2)$ instruction by a so-called *thread instance* of the form $((t_2, a[t_2], 0), (\mathit{false}, a_1, s))$, where s is the value of p_c when the S machine began executing at a_0. Note that a $\texttt{dispatch}(t)$ instruction is an abbreviation for $\texttt{dispatch}(t, \mathit{false})$. The new thread instance of a_1 ensures that the S machine will execute the S code block at a_1 as soon as task t_2 completes. For now the S machine yields to t_2. When t_2 completes, the S machine wakes up, removes the thread instance from the task set, and then executes the $\texttt{dispatch}(t_1)$ instruction at a_1 in a similar way. After t_1 completes, the S machine proceeds to the $\texttt{idle}(p'_c = p_c + 10)$ instruction, which creates in the task set a thread instance of the form $(\perp, (p'_c = p_c + 10, a_2, s))$, which we also call an *idle phase*, where s is again the value of p_c when the S machine began executing at a_0. Now, the S machine idles until the trigger $p'_c = p_c + 10$ is enabled at 10 ms. Then the idle phase is removed from the task set and the S code at address a_2 is executed in a similar way than the previous S code. At 20 ms, the S machine executes the $\texttt{fork}(a_0)$ instruction, which creates in the task set a (zero) idle phase of the form $(\perp, (\mathit{true}, a_0, s'))$, where s' is the current port state of all trigger ports P_G of the S program. Thus the $\texttt{fork}(a_0)$ instruction starts a new instance of the thread at a_0. The following `return` instruction terminates the current thread. Then the S machine immediately removes the (zero) idle phase from the task set and starts executing the S code at a_0. The whole process repeats every 20 ms.

The above scenario assumes that the execution of both tasks completes within 10 ms. In other words, we need that $\mathit{wcet}(t_1) + \mathit{wcet}(t_2) \leq 10$, where $\mathit{wcet}(t)$ is the WCET of task t. We call S code in which task execution must neither be preempted by S code nor other tasks *synchronous*. The following time-triggered S program implements again the above behavior where, however, task t_1 may be preempted by S code and by task t_2:

$$
\begin{array}{ll}
a_0\text{:} & \texttt{call}(d_a) \\
 & \texttt{call}(d_s) \\
 & \texttt{call}(d_i) \\
 & \texttt{schedule}(t_1) \\
 & \texttt{schedule}(t_2) \\
 & \texttt{dispatch}(t_2) \\
a_1\text{:} & \texttt{dispatch}(t_1, p'_c = p_c + 10) \\
a_2\text{:} & \texttt{idle}(p'_c = p_c + 10) \\
 & \texttt{call}(d_s) \\
 & \texttt{schedule}(t_2) \\
 & \texttt{dispatch}(t_2) \\
 & \texttt{dispatch}(t_1) \\
 & \texttt{idle}(p'_c = p_c + 20) \\
 & \texttt{fork}(a_0) \\
 & \texttt{return}
\end{array}
$$

The `dispatch` instruction at address a_1 creates a thread instance of the form $((t_1, a[t_1]), (p'_c = p_c + 10, a_2, s))$, where s is again the value of p_c when the S machine began executing at a_0. If t_1 completes before 10 ms elapsed the S machine will proceed to the subsequent `idle` instruction and idle until the 10 ms elapsed. Technically, when t_1 completes, the task dispatcher replaces the above thread instance in the task set by an enabled thread instance of the form $((t_1, \perp), (\mathit{true}, a_2, s))$, which makes the S machine immediately proceed to the `idle` instruction at a_2. On the other hand, if t_1 does not complete before 10 ms elapsed t_1 gets preempted by the S machine. Then the above thread instance is removed from the task set and the `idle` instruction at a_2 is executed. Since 10 ms already elapsed the `idle` instruction creates an already enabled thread instance, which makes the S machine proceed immediately to the following `call` instruction. Thus the subsequent $\texttt{dispatch}(t_2)$ instruction may execute task t_2 before task t_1 completed. Then the following $\texttt{dispatch}(t_1)$ instruction executes t_1 or, if t_1 has already completed, proceeds immediately to the `idle` instruction to wait for the next 20 ms instant.

The above scenario assumes that the execution of a task has completed before it is scheduled again but not necessarily before other tasks are dispatched. In other words, we need that $wcet(t_1)+2\cdot wcet(t_2)\leq 20$, where $wcet(t)$ is the WCET of task t. We call S code in which task execution may be preempted by S code and other tasks *preemptive*. The following time-triggered S program implements again the above behavior where, however, task t_1 may be preempted by S code but not by task t_2:

$$
\begin{array}{ll}
a_0\text{:} & \texttt{call}(d_a) \\
 & \texttt{call}(d_s) \\
 & \texttt{call}(d_i) \\
 & \texttt{schedule}(t_1) \\
 & \texttt{schedule}(t_2) \\
 & \texttt{dispatch}(t_2) \\
a_1\text{:} & \texttt{dispatch}(t_1, p_c' = p_c + 10) \\
a_2\text{:} & \texttt{idle}(p_c' = p_c + 10) \\
 & \texttt{call}(d_s) \\
 & \texttt{schedule}(t_2) \\
 & \texttt{dispatch}(t_1) \\
 & \texttt{dispatch}(t_2) \\
 & \texttt{idle}(p_c' = p_c + 20) \\
 & \texttt{fork}(a_0) \\
 & \texttt{return}
\end{array}
$$

The only difference of this S code block to the previous block is the order of the last two `dispatch` instructions. Instead of dispatching t_2 even before t_1 may have completed, t_1 is dispatched again at the 10 ms instant. Then, after t_1 completes, t_2 is dispatched. Thus we assume again that the execution of a task has completed before it is scheduled again but in addition we assume that each task completes before another task is dispatched. In other words, we again need that $wcet(t_1)+2\cdot wcet(t_2)\leq 20$, where $wcet(t)$ is the WCET of task t, and that tasks do not preempt each other. We call S code in which task execution may be preempted by S code but not by other tasks *non-preemptive*.

S Code Semantics. The execution of an S program yields an infinite sequence of program configurations, called *trace*. Each configuration tracks the values of all ports, the task set, and the running task. An *S program configuration* consists of (1) a P state s', called *port state*; (2) a set of task invocations and *thread instances* $(N,(g,a,s))$, called *task set*, where N is either a task instance or $\perp$ and (g,a,s) is a trigger binding in which g is a trigger, $a\in A$ is an S code address, and s is a P_G state (if N is $\perp$ we call the thread instance an *idle phase*); and (3) a *running task* R, where R is either $\perp$, or else of the form $(N,\perp)$ where N is either $\perp$ or a task instance. A thread instance $(N,(g,a,s))$ is *enabled* if the trigger predicate $p[g]$ evaluates to true over the pair (s,s') of $P[g]$ states. The configuration c is *schedule-enabled* if the running task of c is $\perp$; otherwise, c is *schedule-disabled*. c is *input-enabling* if c is schedule-disabled and the task set contains no enabled thread instances; otherwise, c is *input-disabling*. We call an input-enabling c *idling* if the running task of c is of the form $(\perp, \cdot)$.

We define the semantics of S code operationally using a pseudo-code description of the S machine. Algorithm 5 shows the main loop of the machine as it executes a given S program. Algorithm 5 is similar to Algorithm 1. Instead of using the E machine interpreter we now use the S machine interpreter (Algorithm 6). The task dispatcher and task scheduler can be reused from the E machine. Similar to the E machine, after entering the main loop, the S machine invokes the task dispatcher to dispatch the task that has been chosen by the task scheduler to execute. Since no task is chosen initially, the task dispatcher enables the event interrupts and then waits for environment events. The occurrence of an environment event wakes up the machine, which immediately disables all event interrupts. Then the S machine interpreter is invoked.

The S machine interpreter runs through the outer while loop of Algorithm 6 as long as there are enabled thread instances in the task set, each time executing a thread of S code that is bound to an enabled thread instance. Initially, the task set contains a single thread instance $(\perp, (\mathit{true}, a_0, \mathit{PortState}(P_G)))$ where a_0 is the initial S code address of the S program. In the inner while loop of Algorithm 6, the machine fetches the current instruction from the program, decodes and executes the instruction, and then determines the address of the next instruction.

Similar to the E machine, after the interpreter is finished executing S code, the task scheduler is invoked to choose a task from the ready set to execute. Unlike in the E machine, the ready set contains only the thread instances of the task set, unless there are no thread instances in the task set. In this case, the ready set is equivalent to the task set. This gives priority to thread instances over task invocations. As before, the task scheduler $\mathit{Schedule}(\mathit{ReadySet})$ is free to choose any scheduling scheme including an idling scheme where no task is chosen although the task set is not empty. The chosen task becomes the running task, which is dispatched by the task dispatcher to execute until an input event occurs. Then the task is put back into the task set with its new program counter only when the task has not yet completed. If the task has completed and it was part of a thread instance, an enabled idle phase is inserted into the task set to make the S machine continue the thread. However, if the task scheduler chooses to idle a running task of the form $(\perp, \cdot)$ is returned and no task is dispatched.

```
loop
  invoke task dispatcher (Algorithm 2)
  invoke S code interpreter (Algorithm 6)
  invoke task scheduler (Algorithm 4)
end loop
```

Algorithm 5: The Scheduling Machine

An *initial configuration* of an S program Π consists of (1) a P state; (2) the task set containing a single thread instance $(\perp, (\mathit{true}, a_0, \mathit{PortState}(P_G)))$ where a_0 is the initial S code address of Π; and (3) the running task set to $(\perp, \perp)$. A *trace* of Π 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 idling, and c' differs from c at most in the values of environment ports other than p_c. In this case, we write $\mathit{e\text{-}step}(c, c')$.

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

```
while there is an enabled thread in TaskSet do
  (N, (g, a_s, s)) := ChooseEnabledThreadInstance(TaskSet)
  TaskSet := TaskSet \ (N, (g, a_s, s))
  if N ≠ ⊥ then
    TaskSet := TaskSet ∪ {(N, ⊥)}
  end if
  ProgramCounter := a_s; Yield := false
  while ProgramCounter ≠ ⊥ do
    i := Instruction(ProgramCounter)
    if call(d) = i then
      PortState(P[d]) := f[d](PortState(P[d] ∪ I[d]))
    else if schedule(t) = i then
      TaskSet := TaskSet ∪ {((t, a[t], 0), ⊥)}
    else if dispatch(t, g) = i then
      if there is a task invocation ((t, a_t, δ), ⊥) in TaskSet then
        TaskSet := (TaskSet \ {((t, a_t, δ), ⊥)}) ∪ {((t, a_t, δ), (g, Next(ProgramCounter), s))}
        Yield := true
      end if
    else if idle(g) = i then
      TaskSet := TaskSet ∪ {(⊥, (g, Next(ProgramCounter), s))}
      Yield := true
    else if fork(a) = i then
      TaskSet := TaskSet ∪ {(⊥, (true, a, PortState(P_G)))}
    end if
    if Yield then
      ProgramCounter := ⊥
    else
      ProgramCounter := Next(ProgramCounter)
    end if
  end while
end while
RunningTask := ⊥
// Configuration: S Code Execution
```

Algorithm 6: The S Code Interpreter

(*Task Completion*) c is input-enabling and a running task of the form $((t, a_t, \delta), B)$ is scheduled in c, and c' results from c by incrementing the clock p_c by some amount of time δ'. In addition, the execution of t results in the program counter $\perp$ (task completion) and in a new $P[t]$ state of c', and, if B is a trigger binding (g, a_s, s), the task set of c' results from c by adding $(\perp, (true, a_s, s))$ to the task set. In this case, we write $t\text{-}step(c, t, c')$.

(*Task Preemption*) c is input-enabling and a running task of the form $((t, a_t, \delta), B)$ is scheduled in c, and c' results from c by incrementing the clock p_c by some amount of time δ'. In addition, the execution of t results in a new program counter a'_t of t that is different from $\perp$ and in a new $P[t]$ state of c', and the task set of c' results from c by adding $((t, a'_t, \delta + \delta'), B)$ to the task set. In this case, we write $t\text{-}step(c, t, c')$.

(*S code Execution*) c is input-disabling and schedule-disabled, and c' results from invoking the S machine interpreter (Algorithm 6) on c.

(*Task Scheduling*) c is schedule-enabled, and c' results from invoking the task scheduler (Algorithm 4) on c.

4 Schedule-Carrying Code

In this section, we introduce the notion of schedule-carrying code. In general, the time-safe execution of an E program requires a system scheduler to determine the order in which tasks triggered by the E program are executed. However, a system scheduler can also be replaced or at least simplified when using the S machine. S code determines the order in which tasks execute. For a given E program and given platform constraints (e.g., WCETs), S code may be generated according to any scheduling strategy at compile time (static scheduling), at runtime (dynamic scheduling), or even partially at compile time and completed at runtime. S code may then serve as (1) a witness of time-safety of a given E program and (2) an executable representation of a schedule. If the S code dispatches at most a single task at any moment in time, a system scheduler will not be necessary anymore. Thus the S machine is a possible alternative to a system scheduler. We argue that generating S code is difficult in the presence of non-trivial platform constraints such as nonpreemptable tasks while checking time safety of E code annotated with S code is simple. We therefore call an E program annotated with an S program *schedule-carrying code* (SCC).

SCC Example. We illustrate the semantics of SCC by combining the E program of Section 2 with a version of the non-preemptive S program of Section 3 that does not contain any of the `call` and `schedule` instructions. The following time-triggered SCC program implements the triggering behavior of the E program and the non-preemptive scheduling behavior of the S program:

a_0: `call`(d_a)
`call`(d_s)
`call`(d_i)
`schedule`(t_1)
`schedule`(t_2)
`future`$(p'_c = p_c + 10, a_1)$
`return`

a_1: `call`(d_s)
`schedule`(t_2)
`future`$(p'_c = p_c + 10, a_0)$
`return`

a'_0: `dispatch`(t_2)
`dispatch`$(t_1, p'_c = p_c + 10)$
`idle`$(p'_c = p_c + 10)$
a'_1: `dispatch`(t_1)
`dispatch`(t_2)
`idle`$(p'_c = p_c + 20)$
`fork`(a'_0)
`return`

The two E code blocks at a_0 and a_1 are equivalent to the E code blocks in Section 2. The E code address a_0 is the initial E code address of the SCC program. In addition, there is one block of S code at a'_0, which is the initial S code address of the SCC program. The execution of the SCC program starts with the E code block at a_0. When the E machine is finished executing this block, the S machine starts executing the S code block at a'_0. After dispatching task t_2 and then t_1, the S machine gets preempted by the E machine at the 10 ms instant. Then the E machine executes the E code block at a_1. Subsequently, the S machine continues to execute the S code at a'_1. At the 20 ms instant, the E machine wakes up and starts a new round by executing the E code block at a_0.

SCC Semantics. An *SCC program* consists of an E program and an S program that may share ports, drivers, tasks, and triggers. An SCC program is *time-triggered* if the E program and the S program are time-triggered. The execution of an SCC program yields an infinite sequence of program configurations, called *trace*. Each configuration tracks the values of all ports, the trigger queue, the task set, and the running task. An *SCC program configuration* consists of (1) a P state s', called *port state*; (2) a queue of *trigger bindings* (g, a, s), called *trigger queue*, where g is a trigger, $a \in A$ is an E code address, and s is a $P[g]$ state; (3) a set of task invocations and *thread instances* $(N, (g, a, s))$, called *task set*, where N is either a task instance or $\perp$ and (g, a, s) is a trigger binding in which g is a trigger, $a \in A$ is an S code address, and s is a P_G state (if N is $\perp$ we call the thread instance an *idle phase*); and (4) a *running task* R, where R is either $\perp$, or else of the form $(N, \perp)$ where N is either $\perp$ or a task instance. A thread instance $(N, (g, a, s))$ is *enabled* if the trigger predicate $p[g]$ evaluates to true over the pair (s, s') of $P[g]$ states. The configuration c is *schedule-enabled* if the running task of c is $\perp$; otherwise, c is *schedule-disabled*. c is *input-enabling* if c is schedule-disabled, the trigger queue contains no enabled trigger bindings, and the task set contains no enabled thread instances; otherwise, c is *input-disabling*. We call an input-enabling c *idling* if the running task of c is of the form $(\perp, \cdot)$.

We define the semantics of SCC code operationally using a pseudo-code description of the SCC machine. Algorithm 7 shows the main loop of the machine as it executes a given SCC program. Similar to the E and S machines, after entering the main loop, the SCC machine invokes the task dispatcher to dispatch the task that has been chosen by the task scheduler to execute. Since no task is chosen initially, the task dispatcher enables the event interrupts and then waits for environment events. The occurrence of an environment event wakes up the machine, which immediately disables all event interrupts. Then the E machine interpreter is invoked followed by the S machine interpreter. Finally, the task scheduler chooses a task to execute.

```
loop
  invoke task dispatcher (Algorithm 2)
  invoke E code interpreter (Algorithm 3)
  invoke S code interpreter (Algorithm 6)
  invoke task scheduler (Algorithm 4)
end loop
```

Algorithm 7: The SCC Machine

An *initial configuration* of an SCC program Π consists of (1) a P state; (2) the trigger queue containing a single trigger binding $(\mathit{true}, a_0, \emptyset)$ where a_0 is the initial E code address of Π; (3) the task set containing a single thread instance $(\perp, (\mathit{true}, a'_0, \mathit{PortState}(P_G)))$ where a'_0 is the initial S code address of Π; and (4) the running task set to $(\perp, \perp)$. A *trace* of Π 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 idling, and c' differs from c at most in the values of environment ports other than p_c. In this case, we write $e\text{-}step(c, c')$.

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

(*Task Completion*) c is input-enabling and a running task of the form $((t, a_t, \delta), B)$ is scheduled in c, and c' results from c by incrementing the clock p_c by some amount of time δ'. In addition, the execution of t results in the program counter $\perp$ (task completion) and in a new $P[t]$ state of c', and, if B is a trigger binding (g, a_s, s), the task set of c' results from c by adding $(\perp, (true, a_s, s))$ to the task set. In this case, we write $t\text{-}step(c, t, c')$.

(*Task Preemption*) c is input-enabling and a running task of the form $((t, a_t, \delta), B)$ is scheduled in c, and c' results from c by incrementing the clock p_c by some amount of time δ'. In addition, the execution of t results in a new program counter a'_t of t that is different from $\perp$ and in a new $P[t]$ state of c', and the task set of c' results from c by adding $((t, a'_t, \delta + \delta'), B)$ to the task set. In this case, we write $t\text{-}step(c, t, c')$.

(*E code Execution*) c is input-disabling and schedule-disabled, there are enabled trigger bindings in c, and c' results from invoking the E machine interpreter (Algorithm 3) on c.

(*S code Execution*) c is input-disabling and schedule-disabled, there are no enabled trigger bindings but enabled thread instances in c, and c' results from invoking the S machine interpreter (Algorithm 6) on c.

(*Task Scheduling*) c is schedule-enabled, and c' results from invoking the task scheduler (Algorithm 4) on c.

5 Non-Preemptive Scheduling for E Code

In this section we discuss a potential application of the SCC on a real time system for which the execution of a requested task not only must be completed before its deadline, but is also required not to be preempted. Non-preemptive scheduling may be preferred solution for addressing the problem of shared resources and critical sections, it reduces task switching overhead and for some applications task preemption is even not allowed. Unfortunately, it is well known that generating non-preemptive code is computationally hard even for uniprocessor scheduling and is usually treated by branch-and-bound algorithms with exponential complexity in the worst case. Only recently, the researchers have shown that problem remains NP-hard even for some simple classes of task sets. The problem of testing the feasibility of a set of periodic tasks with arbitrary arrival times was shown to be NP-hard in the strong sense in [4]. The hardness result for the case where all tasks have the same arrival time was presented in [1]. Even the case when a period π_i of each task t_i from a task set is characterised by a relation $\pi_i = 2^j \pi_0$, where j is an integer and π_0 is the smallest period in the task set, turned out to be NP-hard in the strong sense [5]. In this section we argue that once the schedule has been generated and represented by S code part of the SCC it can be efficiently checked for deadline and non-preemption requirements. At the end we show that similar proposition holds if input task set is specified in the high level language such as Giotto. We start with some basic scheduling terminology and with the hardness result for a non-preemptive scheduling problem that motivates this section.

Non-preemptive Scheduling Problem. A *periodic* task t is given with a 3-tuple $t = (a, \pi, wcet)$, where a is the *arrival* time of t, π is the *period* of t and $wcet$ is the worst case execution time of t. The arrival time of the j-th request for processing of the task t is $a + (j-1)\pi$, and its *deadline* is $a + j\pi$. Given a set of periodic tasks *Tasks* to be executed on a single processor, a *schedule* is a function that maps processing time units to requests of the tasks in *Tasks*. For non-preemptive scheduling, a schedule is *valid* if processing of each request of each task from *Tasks*:

1. starts on or after the request arrival,

2. is not preempted for the task worst case execution time, and

3. terminates on or before the request deadline.

The task set *Tasks* is said to be *feasible* if there exists a valid schedule for it.

Cai and Kong [1] have shown the following scheduling problem to be NP-hard in the strong sense.

NSPT Problem Non-preemptive scheduling of a simply periodic task set.

Instance A set of non-preemptible periodic tasks $Tasks = \{t_1, t_2, ..., t_n\}$ to be executed on a single processor. The arrival of each task t_i is zero, i.e. $t_i = (0, \pi_i, wcet_i)$. The periods of the tasks satisfy relation $\pi_{i+1} = K_i \pi_i$ for $1 \le i \le n-1$ (*simply* periodic task set). The numbers $\pi_i, wcet_i$ $(1 \le i \le n)$, and K_i $(1 \le i \le n-1)$ are assumed to be positive integers.

Question Is the task set *Tasks* feasible?

It is clear that the same hardness result holds if we keep task arrival times at zero, but allow arbitrary task periods. In the next subsection we define a subclass of time-triggered E programs, the class of *periodic E programs* $\mathcal{P}$ such that each E program $\Pi^e \in \mathcal{P}$ describes requests of an instance of a such periodic task set. For the purposes of this section we define the class with respect to the set of E program addresses, and *Instruction* and *Next* function part of an E program definition. Given a periodic E program $\Pi^e \in \mathcal{P}$ we next define a subclass $\mathcal{S}(\Pi^e)$ of time-triggered S programs, the class of S programs *SCC compliant with* Π^e. Such a class of programs describes all, for the purposes of this paper, interesting schedules of the *Tasks* set, while keeping the size of each program in it bounded with the size of Π^e.

Non-preemptive Scheduling for E Code. Let for a set of periodic tasks $Tasks = \{t_1, t_2, ..., t_n\}$ the arrival of each task t_i be zero, i.e. $t_i = (0, \pi_i, wcet_i)$. This assumption allows analysis to be performed up to $\pi = \texttt{lcm}(\pi_1, \pi_2, ..., \pi_n)$ time units, where `lcm` stands for the least common multiple function. If $\gamma = \texttt{lcd}(\pi_1, \pi_2, ..., \pi_n)$ is the least common divisor of the task periods and if $k = \pi/\gamma$, each request for a task execution in the interval $[0, \pi)$ is issued at a time instant $j\gamma$ for some integer j, $0 \le j \le k-1$. An E program $\Pi^e \in \mathcal{P}$ consists of k E code blocks Π^e_j for $0 \le j \le k-1$. Let, for $0 \le j \le k-1$, $Tasks_j = \{t_i \in Tasks \mid j\gamma \texttt{ mod } \pi_i = 0\}$ be the set of all tasks reqested at time $j\gamma$ and let $n^e_j = |Tasks_j|$ $(0 \le n^e_j < n)$. For $0 \le j \le k-1$ we define a set of addresses for E code block Π^e_j with $A^e_j = \{a^e_{j,i} \mid 0 \le i \le n^e_j + 1\}$. Each block Π^e_j is contained of a sequence of n^e_j `schedule`, one `future` and one `return` instruction in the specified order. The set of addresses of the E program Π^e is given with $A^e = \bigcup_{0 \le j \le k-1} A^e_j$. The initial address of Π^e is $a^e_{0,0}$. The successor address function satisfies $Next(a^e_{j,i}) = a^e_{j,i+1}$ if $0 \le i < n^e_j + 1$ and $Next(a^e_{j,n^e_j+1}) = \perp$. For $0 \le j \le k-1$ the instruction function *Instruction* satisfes the following three conditions:

1. for each $t \in Tasks_j$ there is an integer i such that $0 \le i < n^e_j$ and $Instruction(a^e_{j,i}) = \texttt{schedule}(t)$,

2. if $0 \le j < k-1$ then $Instruction(a^e_{j,n^e_j}) = \texttt{future}(p'_c = p_c + \gamma, a^e_{j+1,0})$;
in addition $Instruction(a^e_{k-1,n^e_{k-1}}) = \texttt{future}(p'_c = p_c + \gamma, a^e_{0,0})$,

3. $Instruction(a^e_{j,n^e_j+1}) = \texttt{return}$.

An S program Π^s from $\mathcal{S}(\Pi^e)$ consists of k S code blocks Π^s_j for $0 \le j \le k-1$. Each block Π^s_j is contained of a sequence of n^d_j `dispatch` instructions that are separated by at most one `idle` instruction, and ends with one `idle`, one `fork` and one `return` instruction in the specified order. A trigger g in a `schedule` or `idle` instruction is a time trigger specified with an integer Δ, i.e. g is enabled if $p'_c \ge p_c + \Delta$. If the size of the S code block Π^s_j is n^s_j we have that $n^s_j \le 2n^d_j + 3$. For $0 \le j \le k-1$ we define a set of addresses for S code block Π^s_j with $A^s_j = \{a^s_{j,i} \mid 0 \le i \le n^s_j - 1\}$. The set of addresses of the S program Π^s is given with $A^s = \bigcup_{0 \le j \le k-1} A^s_j$. The initial address of Π^s is $a^s_{0,0}$. The successor address function satisfies $Next(a^s_{j,i}) = a^s_{j,i+1}$ if $0 \le i < n^s_j - 1$ and $Next(a^s_{j,n^s_j-1}) = \perp$. The instruction function *Instruction* satisfes the following five conditions:

1. if for some $t \in Tasks$, some trigger g, $0 \le j < k-1$ and $0 \le i < n^s_j - 3$, $Instruction(a^s_{j,i}) = \texttt{dispatch}(t, g)$ then for some $t' \in Tasks$ and trigger g', $Instruction(a^s_{j,i+1}) = \texttt{dispatch}(t', g')$ or $Instruction(a^s_{j,i+1}) = \texttt{idle}(g')$,

2. if for some trigger g, $0 \le j < k-1$ and $0 \le i < n^s_j - 3$, $Instruction(a^s_{j,i}) = \texttt{idle}(g)$ then for some $t' \in Tasks$ and trigger g', $Instruction(a^s_{j,i+1}) = \texttt{dispatch}(t', g')$,

3. $Instruction(a^s_{j,n^s_j-3}) = \texttt{idle}(g)$ for some trigger g,

4. if $0 \le j < k-1$ then $Instruction(a^s_{j,n^s_j-2}) = \texttt{fork}(a^s_{j+1,0})$; in addition $Instruction(a^s_{k-1,n^s_{k-1}-2}) = \texttt{fork}(a^s_{0,0})$,

5. $Instruction(a^s_{j,n^s_j-1}) = \texttt{return}$.

Lastly, in order for the S program Π^s to be SCC compliant with the E program Π^e, we additionally require that the numbers of `dispatch` and `schedule` instructions, n^d_j and n^s_j, and the number of E (and S) code blocks k satisfy the condition

$$\sum_{j=0}^{k} n^d_j \le \sum_{j=0}^{k} n^e_j + k. \tag{1}$$

The size of an S program Π^s that satisfies the last condition is linear in the size of the E program Π^e. On the other hand, this condition allows execution of tasks that are preempted by some or even all k E code blocks Π^e_j. For the case of non-preemptive schedules, no task may be preempted by any other task, so no task may be dispatched twice in the same S code block. Therefore, a class of S programs that describes such schedules satisfies the condition 1. Checking if a given E program is periodic or if a given S program is SCC compliant with a given E program requires time linear in the size of the programs. Therefore, in the rest of this section we assume that only E and S programs that passed these tests may be input programs of the algorithm that we discuss next.

```
ETime := 0; ΔETime := 0; Period := 0
RunningTask := (⊥, ·); Preempted := false
a_e := a^e_{0,0}
TaskSet := {(⊥, (true, a^s_{0,0}, ETime))}
while Period < 2 do
  invoke task dispatch checker (Algorithm 9)
  invoke E code checker (Algorithm 10)
  invoke S code checker (Algorithm 11)
  if TaskSet ≠ ∅ then
    RunningTask := GetTask(TaskSet)
  else
    RunningTask := (⊥, ·)
  end if
end while
return ACCEPT
```

Algorithm 8: The Non-preemptive SCC Checker

The Algorithm 8 shows the main loop of the procedure for checking validity and non-preemption of the schedule for the schedule-carrying code Π consisting of a periodic E program $\Pi^e \in \mathcal{P}$ and an S program $\Pi^s \in \mathcal{S}(\Pi^e)$. We assume that $a^e_{0,0}$ and $a^s_{0,0}$ are initial addresses of A^e and A^s respectively, and that *wcet* is the worst case execution map. The structure of the Algorithm 8 is very similar to the main loop of the SCC Machine Algorithm 7. It follows the same steps by simulating task dispatching (Algorithm 9), E code interpretation (Algorithm 10), and S code interpretation (Algorithm 11). Since S programs from $\mathcal{S}(\Pi^e)$ describe only static schedules, i.e. at any time there is at most one enabled thread instance, invocation of the task scheduler from the SCC machine algorithm is replaced with the simple *GetTask* call. While thread instances in *TaskSet* are manipulated in the same manner as for the SCC machine, the trigger queue is not used, since at any time only one trigger is activated due to a single `future` instruction in each E code block. Instead, the algorithm keeps the time of the last E machine interpreter invocation in the $ETime$ variable, periodically updating it with $\Delta ETime$ time units remembered from the last `future` instruction. To verify the schedule it is enough to check its validity in the interval $[0, \pi]$, including the π instant since all of the periodic tasks have their deadlines at that time. At the same time E machine interpreter would again execute E code at the address $a^e_{0,0}$, so we use a variable $Period$, a counter variable for the executions of the instruction at $a^e_{0,0}$, as a test for the completion of the check in case of accepting the schedule as a valid one. The variable $STime$ keeps track of the times in which task dispatcher would have been invoked. To compute the next dispatching time $NextSTime$, the Algorithm 9 uses the current data of the running task t: the time δ for which t already executed from the task instance, and the trigger g and the time s of the activation of g from the trigger binding. $NextSTime$ is determined by the first event that would have come: g becomes enabled $(s + \Delta)$, the execution of t is completed $(STime + wcet(t) - \delta)$, or E machine is invoked $(ETime + \Delta ETime)$. In all cases the new time for which task t would have been executed up to that instant is updated in the task instance. Since no task actually executes the stored value for the program counter a_t is irrelevant. The Algorithm 10 is executed if E machine should have been invoked, i.e. when $STime = ETime + \Delta ETime$. When E code interpretation loop decodes `schedule`(t) instruction, it checks whether an instance of the same task t is already in the *TaskSet*. If it is, the execution of t for its previous request could not have been completed, time safety is

if $RunningTask = (\perp, B)$ **then**
  if $B = (p_c' \geq p_c + \Delta, a_s, s)$ **then**
    $NextSTime := \mathtt{min}(s + \Delta,\ ETime + \Delta ETime)$
    // Configuration: Idle Time Tick
  else
    $NextSTime := ETime + \Delta ETime$
    // Configuration: Environment Event
  end if
else
  $((t, a_t, \delta), (g, a_s, s)) := RunningTask$
  if $Preempted$ `and` $\delta = 0$ **then**
    // $\exists t'.\ (((t', \cdot, \delta'), \cdot) \in TaskSet\ \wedge\ t' \neq t\ \wedge\ \delta' > 0)$
    // Non-preemption Violation
    `return REJECT`
  end if
  if $g = (p_c' \geq p_c + \Delta)$ **then**
    $NextSTime := \mathtt{min}(s + \Delta,\ STime + wcet(t) - \delta,\ ETime + \Delta ETime)$
  else
    $NextSTime := \mathtt{min}(STime + wcet(t) - \delta,\ ETime + \Delta ETime)$
  end if
  if $NextSTime = STime + wcet(t) - \delta$ **then**
    $TaskSet := TaskSet \cup \{(\perp, (true, a_s, s))\}$
    $Preempted := false$
    // Configuration: Task Completion
  else
    $TaskSet := TaskSet \cup \{((t, a_t, \delta + NextSTime - STime), (g, a_s, s))\}$
    $Preempted := true$
    // Configuration: Task Preemption
  end if
end if
$STime := NextSTime$

Algorithm 9: The Task Dispatch Checker

```
if STime = ETime + ΔETime then
  ETime := ETime + ΔETime
  ProgramCounter := a_e
  if ProgramCounter = a^e_{0,0} then
    Period := Period + 1
  end if
  while ProgramCounter ≠ ⊥ do
    i := Instruction(ProgramCounter)
    if schedule(t) = i then
      if ((t, ·, ·), ·) ∈ TaskSet then
        return REJECT
        // Deadline Violation
      end if
      TaskSet := TaskSet ∪ {((t, a[t], 0), ⊥)}
    else if future(p'_c = p_c + Δ, a) = i then
      ΔETime := Δ; a_e := a
    end if
    ProgramCounter := Next(ProgramCounter)
  end while
end if
RunningTask := ⊥
```

Algorithm 10: The E Code Checker

violated and the algorithm terminates by rejecting Π. If it is not, there is a new request for t, so it is inserted in the *TaskSet* with zero time executed so far. Non-preemption violation could be similarly tested when dispatching a task t in the Algorithm 9, by checking if any other t' in *TaskSet* already started its execution ($\delta' > 0$). However, in order to make each execution of the Algorithm 9 independant of the number of tasks in *TaskSet*, even constant in time, we use a boolean variable *Preempted* to keep track if the task that was last dispatched completed its execution or was preempted. Non-preemption violation is detected when a first dispatch of a task invocation ($\delta = 0$) occurs while *Preempted* is *true*. The Algorithm 11 is the same as S code interpreter part of the SCC machine, except for the control-flow optimizations due to the fact that at any time there is at most one enabled thread instance.

Proposition 5.1
Let a schedule-carrying code Π be given with a periodic E program $\Pi^e \in \mathcal{P}$ and an S program SCC compliant with Π^e. Let the worst case execution map for tasks in Π be *wcet*. Checking if Π is time safe for *wcet* map and if no task preempts any other task can be done in time linear in the size of the E program Π^e, i.e. in $O(|\Pi^e|)$ time.

Proof Let k be the number of E (and S) code blocks. The worst case running time is achieved if the algorithm accepts the schedule after the variable *Period* becomes 2. Before that the Algorithm 10 is executed exactly $k+1$ times, each time starting from the address $a^e_{j \bmod k,0}$, $0 \le j < k+1$. Therefore, each E program instruction is decoded at most twice and since processing of each instruction takes constant time, the total time spent in the Algorithm 10 is $O(|\Pi^e|)$. The `while` loop in the Algorithm 11 runs until $dispatch(\cdot, g)$ or $idle(g)$ instruction is decoded with a trigger g not enabled. Since thread continuation is achieved through `fork` instruction, similarly as for an E program instruction,

if there is an enabled thread in $\mathit{TaskSet}$ **then**
  $(N, (g, a_s, s)) := \mathit{GetEnabledThreadInstance}(\mathit{TaskSet})$
  $\mathit{TaskSet} := \mathit{TaskSet} \setminus (N, (g, a_s, s))$
  if $N \neq \perp$ **then**
    $\mathit{TaskSet} := \mathit{TaskSet} \cup \{(N, \perp)\}$
  end if
  $\mathit{ProgramCounter} := a_s$; $\mathit{Yield} := \mathit{false}$
  while $\mathit{ProgramCounter} \neq \perp$ **do**
    $\mathit{Reset} := \mathit{false}$
    $i := \mathit{Instruction}(\mathit{ProgramCounter})$
    if $\texttt{dispatch}(t, g) = i$ **then**
      if there is a task invocation $((t, a_t, \delta), \perp)$ in $\mathit{TaskSet}$ `and` g is not enabled **then**
        $\mathit{TaskSet} := (\mathit{TaskSet} \setminus \{((t, a_t, \delta), \perp)\}) \cup \{((t, a_t, \delta), (g, \mathit{Next}(\mathit{ProgramCounter}), s))\}$
        $\mathit{Yield} := \mathit{true}$
      end if
    else if $\texttt{idle}(g) = i$ **then**
      if g is not enabled **then**
        $\mathit{TaskSet} := \mathit{TaskSet} \cup \{(\perp, (g, \mathit{Next}(\mathit{ProgramCounter}), s))\}$
        $\mathit{Yield} := \mathit{true}$
      end if
    else if $\texttt{fork}(a) = i$ **then**
      $s := ETime$
      $\mathit{Reset} := \mathit{true}$
      $\mathit{ProgramCounter} := a$
    end if
    if Yield **then**
      $\mathit{ProgramCounter} := \perp$
    else if `not` Reset **then**
      $\mathit{ProgramCounter} := \mathit{Next}(\mathit{ProgramCounter})$
    end if
  end while
end if
$\mathit{RunningTask} := \perp$

Algorithm 11: The S Code Checker

we have that each S program instruction is decoded at most twice and processed in constant time. Finally, the Algorithm 9 runs in constant time at most once per S program instruction. Since the condition 1 requires the size of S program to be bounded by the size of E program, the whole Algorithm 8 runs in time linear in the size of the E program. □

Non-preemptive Scheduling for Giotto. We conclude the section by showing that similar complexity gap between generating and checking a non-preemptive schedule exists if the task set description is given in the Giotto programming language [3].

NSGP Problem Non-preemptive scheduling of a single mode Giotto program.

Instance A Giotto program Π_G with a single mode m. A period of the mode m is $\pi[m] \in \mathbb{N}_{>0}$ and its set of task invocations is $Invokes[m]$. Each task t from $Invokes[m]$ is non-preemptible and given with a pair of positive integers $(\omega(t), wcet(t))$, where $\omega(t)$ is the task frequency relative to the mode period and $wcet(t)$ is the task worst case execution time.

Question Is the task set in the program Π_G feasible?

Proposition 5.2
The NSGP problem is NP-hard in the strong sense.

Proof We prove the proposition by a direct polynomial-time transformation from the NSPT problem. Given an instance of the NSPT problem, a simply periodic task set $Tasks = \{t_1, t_2, ..., t_n\}$ with $t_i = (0, \pi_i, wcet_i)$, we constuct a Giotto program with a single mode m, such that $\pi[m] = \mathtt{lcm}(\pi_1, \pi_2, ..., \pi_n)$ and $Invokes[m] = \{(\pi[m]/\pi_i, wcet_i) | t_i \in Tasks\}$. The equivalence of feasibility of the task sets in the two problems follows from the Giotto program semantics, which implies task arrival times at time zero, periodic task invocations and requires task completion before the next request occurs. □

Given an instance of the NSGP problem and a schedule, we next show how difficult is to check whether the schedule is a valid schedule. Let $|Invokes[m]| = n$ and for each $(\omega_i, wcet_i) \in Invokes[m]$ $(1 \leq i \leq n)$ let π_i be the period of the task t_i, $\pi_i = \pi[m]/\omega_i$. If $W = \mathtt{max}_{1 \leq i \leq n} \omega_i$ the total number of task requests in a single mode period $\pi[m]$, $\sum_{1 \leq i \leq n} \omega_i$, is bounded by nW. A schedule is given with a function $S(i,j)$ that maps a j-th request $(1 \leq j \leq \omega_i)$ of the task t_i $(1 \leq i \leq n)$ to the starting execution time of the request. From the three conditions of the definition of the valid non-preemptive schedule it follows that the function S must satisfy the corresponding three conditions:

1. $(j-1)\pi_i \leq S(i,j) < j\pi_i$, for each $1 \leq i \leq n$ and each $1 \leq j \leq \omega_i$,

2. $S(i_1, j_1) > S(i_2, j_2) \implies S(i_1, j_1) \geq S(i_2, j_2) + wcet_{i_2}$ for each $1 \leq i_1, i_2 \leq n$, each $1 \leq j_1 \leq \omega_{i_1}$, and each $1 \leq j_2 \leq \omega_{i_2}$, and

3. $(j-1)\pi_i < S(i,j) + wcet_i \leq j\pi_i$, for each $1 \leq i \leq n$ and each $1 \leq j \leq \omega_i$.

Proposition 5.3
Given a schedule function S for the NSGP problem, checking if the schedule is valid can be done in time no more than $O(nW\mathtt{log}(nW))$.

Proof Since the task execution times are positive numbers, for conditions 1 and 3 we need to check if $(j-1)\pi_i \leq S(i,j)$ and $S(i,j) + wcet_i \leq j\pi_i$ for $1 \leq i \leq n$ and $1 \leq j \leq \omega_i$. This can be done in $O(nW)$ time. To check the non-preemption condition 2 we first sort all $S(i,j)$ values in

$O(nW\mathtt{log}(nW))$ time and then for every two adjacent elements $S(i_1, j_1) > S(i_2, j_2)$ of the sorted list we check if $S(i_1, j_1) \geq S(i_2, j_2) + wcet_{i_2}$ ($O(nW)$ time). $\square$

References

[1] Y. Cai and M.C. Kong. Nonpreemptive scheduling of periodic tasks in uni- and multiprocessor systems. In *Algorithmica*, volume 15, pages 572–599. Springer-Verlag New York Inc., 1996.

[2] T.A. Henzinger and C.M. Kirsch. The embedded machine: predictable, portable real-time code. In *Proc. ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI)*, pages 315–326. ACM Press, 2002.

[3] T.A. Henzinger, C.M. Kirsch, Rupak Majumdar, and Slobodan Matic. Time-safety checking for embedded programs. In *Proc. International Workshop on Embedded Software (EMSOFT)*, volume 2491 of *LNCS*, pages 76–92. Springer, 2002.

[4] K. Jeffay, D.F. Stanat, and C.U. Martel. On non-preemptive scheduling of periodic and sporadic tasks. In *Proc. of the Twelfth IEEE Real-Time Systems Symposium*, pages 129–139. IEEE Computer Society Press, 1991.

[5] J.R. Nawrocki, A. Czajka, and W. Complak. Scheduling cyclic tasks with binary periods. In *Information Processing Letters*, volume 65, pages 173–178. Elsevier Science, 1998.