A Typed Assembly Language for Real-Time Programs[*]

Thomas A. Henzinger
EPFL and UC Berkeley
tah@epfl.ch

Christoph M. Kirsch
University of Salzburg and UC Berkeley
ck@cs.uni-salzburg.at

ABSTRACT

We present a type system for E code, which is an assembly language that manages the release, interaction, and termination of real-time tasks. E code specifies a deadline for each task, and the type system ensures that the deadlines are path-insensitive. We show that typed E programs allow, for given worst-case execution times of tasks, a simple schedulability analysis. Moreover, the real-time programming language Giotto can be compiled into typed E code. This shows that typed E code identifies an easily schedulable yet expressive class of real-time programs. We have extended the Giotto compiler to generate typed E code, and enabled the run-time system for E code to perform a type and schedulability check before executing the code.

Categories and Subject Descriptors: D.3.2 [Programming Languages]: Language Constructs and Features

General Terms: Languages

1. INTRODUCTION

In hard real-time programming one faces the challenge that the execution of a set of software tasks must satisfy given, application-specific timing constraints. Traditionally one uses a mix of worst-case execution time (WCET) analysis, scheduling theory, and testing to ensure that the timing requirements are met on the target platform. Our work has been aimed at reducing the role of testing —and the associated manual code tweaking— in this process. For this purpose, we have advocated the use of high-level programming languages that explicitly specify timing requirements, and the development of compilers that ensure that the generated code is "time-safe" (i.e., meets the specified timing requirements) on the target platform [2].

Such a time-assurance compiler can be partitioned into two phases: a platform-independent code generation phase, followed by a platform-dependent code analysis phase [3]. The code generation phase produces native task code, which is typically C code, as well as glue code that manages the release, interaction, and termination of software tasks. The glue code is the interface between code generation and code analysis, and as such must explicitly specify task release times and deadlines that guarantee the real-time semantics of the source code. For glue code, we have proposed a portable assembly language, called *E code* [3], which is interpreted by a virtual machine —the *E(mbedded) machine*— and therefore can be executed on any platform that offers an implementation of the E machine. However, before executing E code, the host must ensure that all deadlines that are specified in the code are met. This is done in the second compilation phase, which is platform-dependent, because the satisfaction of timing requirements depends on the WCETs of tasks on the target platform, and on the scheduler of the target platform. More ambitiously, the second compilation phase may bypass the system scheduler and generate explicit scheduling code [5].

The main difficulty with this scheme is that in theory as in practice, the schedulability analysis that must be performed by the second compilation phase is often hard [4]. However, we have succeeded in identifying a high-level language for control applications, called *Giotto* [2], which is based on periodic task invocations and time-triggered mode switches, for which the schedulability analysis is surprisingly simple [4]. This is because the E code generated from Giotto programs has a very special form. In this paper, we generalize this result by providing a *type system for E code* with the following three properties. First, for every E program, type derivation and type checking are simple (linear in the size of the code). Second, for every *typed* E program, schedulability analysis is simple. Third, from Giotto programs we can always generate typed E code, but there are also many typed E programs that do not correspond to any Giotto programs. In other words, typed E code extracts the features of Giotto that make scheduling easy, but removes other restrictions of Giotto, such as task periodicity.

The use of typed E code permits us to separate the two compilation phases in space as well as time (see Figure 1). The code-generating host produces native task code and typed E code. For practical purposes, it attaches to the E code not fully specified types, but much smaller "tips"[1], which contain a limited amount of information from which types are easily reconstructed. This code is provided to the code-executing host, which also has information about

[*]This research was supported in part by the AFOSR MURI grant F49620-00-1-0327 and by the NSF grants CCR-0208875 and CCR-0225610.

[1]The term "tip" is courtesy of Ranjit Jhala.

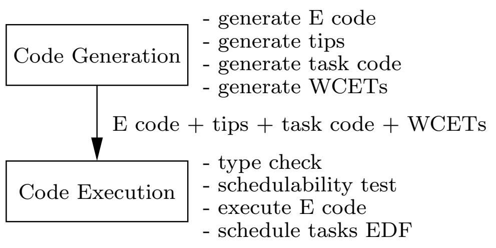

Figure 1: Code generation and execution

the WCETs of tasks. Note that different code-executing hosts may have different WCETs. If the code executor does not trust the code generator, it first type checks the E code. Then it performs the schedulability test, which amounts to computing several utilization equations. If both the type check (WCET independent) and the schedulability test (WCET dependent) pass, then the execution of the code with an EDF (earliest-deadline-first) scheduler [6] is guaranteed to be time-safe. Scheduling strategies other than EDF may also be used with E code but may require different schedulability tests not described here.

Typed E code

E code has, in addition to control-flow instructions (`if`, `jump`, and `return`), three instructions. The `schedule`(t) instruction releases a task t. The `call`(d_t) instruction reads the outputs of task t, where d_t is a so-called driver that accesses the outputs of t; it thus provides a deadline for t and we say that the `call` instruction "terminates" the task t. The `future`(n, a) instruction creates a new thread of E code; it specifies that the current thread continues at address a after waiting for n time units, and that a new thread is started immediately at the address that follows the `future` instruction. Thus, `future`(n, a) is a *real-time jump*: the time offset n is the *trigger* of the jump, and a is the destination. The left column of Figure 2 shows E code that releases the task t every 10 time units, and terminates t just before each release. This is single-threaded code, as the thread created by the `future` instruction immediately finishes.

At any program point, the *consumed time* $C(t)$ of a task t is the number of time units since the last `schedule`(t) instruction (since release), and the *remaining time* $R(t)$ is the number of time units till the next `call`(d_t) instruction (till termination). An E program can be typed if it satisfies two conditions. First, consumed and released times must always be independent of the program path. More specifically, if there are several possible paths to a program point, then for every task t, there is an integer $C(t)$ such that the consumed time of t along all paths is either $C(t)$ or $\perp$, where $\perp$ indicates that the task has not been released on that path. Similarly, if there are several possible paths from a program point, then for every task t which has been released, there is an integer $R(t)$ such that the remaining time of t along all paths is $R(t)$. Note that the E program in the left column of Figure 2 can be trivially typed, as there is only a single path. However, in the code of the right column, 5 time units after the release of task t, the `if` instruction is executed. Then either t is terminated immediately (true branch), or t is only terminated after another 5 time units (false branch). So at the time of the release, the remaining time of t is either 5 or 10, depending on which path is taken. This program cannot be typed.

```
a : call(d_t) : {t : 10}         a1 : call(d_t)
    schedule(t) : {t : 10}            schedule(t)
    future(10, a) : {}                future(5, a2)
    return()                          return()
                                 a2 : if(c, a1)
                                      future(5, a1)
                                      return()
```

Figure 2: E code and tips

The second condition for an E program to be typed is a non-interference condition for threads. It insists that no two concurrent threads can access the same task (by either `schedule` or `call` instructions). This ensures that consumed and remaining times of tasks are not only path-independent but also context-independent, i.e., independent of threads that are executed concurrently. It should therefore not be surprising that for typed E programs, schedulability analysis can be path-insensitive.

In summary, the *type* of a control location[2] is a triple (S, C, R), where S is the set of tasks that are available for release in the current thread, C is a function that maps every released task to its consumed time, and R is a function that maps every released task to its remaining time. In Section 2, we review the definition of E code, and in Section 3 we provide the type system, as well as a linear algorithm for type derivation. Type derivation proceeds in two phases. The first phase infers a tip for every E code instruction: the tip for a `schedule`(t) instruction is the remaining time for t, that is, the time to termination; the tip for a `call`(d_t) instruction is the consumed time for t, that is, the time since release; and the tip for a `future` instruction is the set of tasks that are assigned to the newly created thread. In the left column of Figure 2, each instruction is annotated (after the colon) by the corresponding tip. From these tips, in a second phase, it is easy to construct full types.

Schedulability test for typed E code

In Section 4, we define the timing-flow graph of an E program, which makes explicit the passage of time. If there are k threads, then the size of the timing-flow graph is $O(2^k)$. While this explosion is unavoidable due to concurrency, the number of E code threads is typically small; in particular, Giotto can be compiled into single-threaded E code [4]. Our main theorem shows that if an E program is typed and satisfies a utilization equation for each node of the timing-flow graph, then an EDF schedule will meet all deadlines specified by `call` instructions. (Note that EDF is only well-defined if the remaining times $R(t)$ of all tasks t are path-independent, as is the case for typed E programs.) The utilization tests are simple: they check that all released tasks t can be executed for time $w(t)/(C(t) + R(t))$ during the next time unit, where $w(t)$ is the WCET of t, and $C(t) + R(t)$ is the total available time for executing t. If all utilization tests succeed, then the program can be scheduled in a path-insensitive way, and by a standard argument, along each path the resulting schedule can be converted into EDF. By contrast, if the program is not typed, then a path-sensitive analysis must be performed for checking schedulability, and the best known algorithms are exponential in the number of tasks and program locations [4, 5].

In summary, schedulability analysis for E code can be sep-

[2]Note that we type control, not data.

arated into a platform (WCET) independent typing problem, followed by simple platform (WCET) dependent utilization tests. Note that we assume WCETs for tasks are given. We do not attempt to solve the problem of estimating WCETs but rely on methods developed elsewhere, e.g., in [1]. Also, our schedulability analysis abstracts the values of branch conditions and assumes that all syntactic paths of a program are feasible; it thus gives a sufficient condition for schedulability.

Implementation of typed E code

We have extended the binary format for E code to represent tips and we have extended the Giotto compiler with a module that derives tips for the E code generated by the compiler. We have also added modules for type checking and for EDF schedulability analysis to the E machine. When the E machine receives E code with tips, it first type checks the code, i.e., it checks if the tips are consistent with the program, and then it derives the full type of each control location if the program can be typed. If successful, the E machine runs the EDF schedulability test based on given WCETs for the tasks. If this also succeeds, the E machine starts executing the E code using an EDF scheduler for the tasks. The E machine also monitors all real-time requirements at run-time, for the case that erroneous WCET information was used in the analysis.

An interesting problem for future work is to combine typed E code with driver and task code represented by a typed assembly language (TAL) [8]. TAL originally inspired our work on typed E code. For example, a combination of typed E code with TAL (for the task code) may help to verify that a driver indeed reads the outputs of a particular task and of no other task. TAL has also been used in a programming language for embedded systems [7]. However, unlike typed E code, this language uses types in the standard way, e.g., for array-bounds checking or to ensure the correct initialization of variables.

2. E CODE

This section is a summary of the E Machine presented in [3]. The E machine has two input interfaces and one output interface: (1) physical processes communicate information to the E machine through *environment ports*, such as clocks and sensors; (2) application software processes, called *tasks*, communicate information to the E machine through *task ports*; and (3) the E machine communicates information to the physical processes and to the tasks by calling system processes, called *drivers*, which write to *driver ports*. The E machine also evaluates *conditions* that read driver ports in order to control the machine behavior. Thus, environment and task ports are input ports of the E machine, while driver ports are output ports. A change of value at an input port is called an *input event*. Every input event causes an interrupt that is observed by the E machine and may initiate the execution of E code. The E machine uses so-called *triggers* which read input ports in order to monitor event interrupts. Given a set Z of ports, a Z *state* is a function that maps each port in Z to a value.

E code supervises the execution of drivers and tasks, and evaluates conditions and triggers. Tasks, drivers, conditions, and triggers are functional code that is external to the E machine and must be implemented in some programming language like C. A *task* is a piece of preemptive, user-level code, which typically implements a computation activity. A task has no internal synchronization points. A task t reads from driver ports $I[t]$ and computes on task ports $O[t]$. In favor of a simpler presentation, we assume that the set $I[t] \cup O[t]$ of ports on which t operates is fixed, and that $O[t] \cap O[t'] = \emptyset$ for all tasks $t' \neq t$. Logically, t computes a function from $I[t] \cup O[t]$ states to $O[t]$ states. A *driver* is a piece of system-level code, which typically facilitates a communication activity. A driver may provide sensor readings as arguments to a task, or may load task results into actuators, or may provide task results as arguments to other tasks. A driver d reads from environment and task ports $I[d]$ and writes to driver ports $O[d]$. We write $\mathit{share}(d, t)$ if the driver d shares a port with the task t, that is, if $I[d] \cap O[t] \neq \emptyset$ or $O[d] \cap I[t] \neq \emptyset$. For simplicity, we assume that a driver shares ports with at most a single task, i.e., if $\mathit{share}(d, t)$, then $\neg\mathit{share}(d, t')$ for all tasks $t' \neq t$. Logically, d computes a function from $I[d] \cup O[d]$ states to $O[d]$ states. A driver executes in logical zero time, i.e., before the next input event can be observed. This is achieved by disabling event interrupts during the execution of a driver.

Conditions are used to query the state of driver ports. A *condition* c consists of a predicate that reads from driver ports $I[c]$ and determines the outcome of conditional branching instructions in E code. Triggers are used to monitor the occurrence of input events. Once a trigger is *activated*, it is logically evaluated with every input event, and an active trigger becomes *enabled* when it evaluates to true. In this paper, we consider only *time-triggered E code*, where all triggers are time triggers. A *time trigger* monitors the environment port p_c which represents the system clock. We denote a time trigger by its offset n, where n is the number of clock ticks that need to happen before the trigger goes off. Hence, once the trigger is activated, with every change of p_c the offset is decremented, and the trigger becomes enabled when $n = 0$. Similarly to drivers, conditions and triggers are system-level code that is evaluated in logical zero time with event interrupts disabled.

E code has three non-control-flow instructions. A $\texttt{call}(d)$ instruction initiates the execution of a driver d. As the implementation of d is system-level code, the E machine waits until d is finished before interpreting the next instruction of E code. A $\texttt{schedule}(t)$ instruction releases a task t to run concurrently with other released tasks by putting t into the ready queue of an external task scheduler. Then the E machine proceeds to the next instruction. The task t does not execute before the E machine relinquishes control of the processor to the scheduler. The `schedule` instruction itself does not order the execution of tasks. If the E machine runs on top of an operating system, the task scheduler may be implemented by the scheduler of the OS [3]. An alternative implementation of a task scheduler is the so-called S machine [5], which is a virtual machine that dispatches tasks to execute according to a control program called S code. A $\texttt{future}(n, a)$ instruction marks the E code at the address a for execution at some future time instant when the time trigger n becomes enabled. In order to handle multiple active time triggers, a `future` instruction puts the trigger-address pair into a queue of active triggers. Then, the E machine creates a new thread, which starts immediate execution at the instruction that follows the `future` instruction.

E code also has three control-flow instructions. The $\texttt{if}(c, a)$ instruction evaluates the condition c and, if c is true, jumps

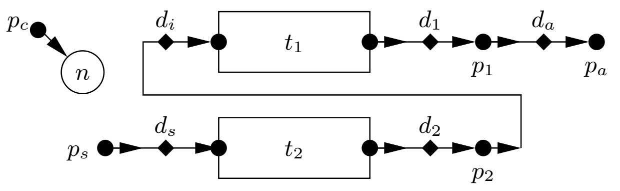

Figure 3: A simplified helicopter flight controller

to the E code at the address a; otherwise, the E machine proceeds to the next instruction. The `jump`(a) instruction is an absolute jump to the E code at the address a. The `return`() instruction finishes the execution of the current thread of E code.

An E code example

We use as example a simplified version of the flight-control program for an autonomous model helicopter built at ETH Zürich [2]. Figure 3 shows the topology of the program: we denote ports by bullets, tasks by rectangles, drivers by diamonds, and triggers by circles. There are two tasks, both implemented in native code: the control task t_1, and the navigation task t_2. The navigation task processes GPS input every 10 ms and provides the processed data to the control task. The control task reads additional sensor data (not modeled here), computes a control law, and writes the result to actuators (reduced here to a single port p_a). The control task is executed every 20 ms.

The data communication requires five drivers: an output driver d_1, which copies the output of the control task t_1 to a driver port p_1, and an output driver d_2, which copies the output of the navigation task t_2 to a driver port p_2; a sensor driver d_s, which provides the GPS data to the navigation task; an input driver d_i, which provides the result of the navigation task to the control task; and an actuator driver d_a, which loads the result of the control task into the actuator. The drivers may process the data in simple ways (such as type conversion), as long as their WCETs are negligible. There are two environment ports, namely, the system clock p_c and the GPS sensor p_s; two task ports, one for the result of each task; and five driver ports —the destinations of the five drivers, including the actuator p_a. Here is a high-level Giotto [2] description of the program timing in the hover mode of the helicopter, called m:

```
mode m() period 20 ms {
  actfreq 1 do p_a(d_a);
  taskfreq 1 do t_1(d_i);
  taskfreq 2 do t_2(d_s); }
```

The "`actfreq` 1" statement causes the actuator to be updated once every 20 ms; the "`taskfreq` 2" statement causes the navigation task to be invoked twice every 10 ms; etc. The E code generated by the Giotto compiler [4] is shown in the left column of Figure 4 starting at the address a_1. It consists of two blocks: a *block* of E code is a sequence of instructions that ends with a `return` instruction. The block at address a_1 is executed at the beginning of a period, say, at 0 ms: it calls the five drivers, which update the output ports of the tasks and provide new data for the actuator and the tasks, then releases the two tasks to the task scheduler, and finally activates a time trigger with offset 10 ms and address a_2. When the block finishes, the trigger queue of the E machine contains the trigger with offset 10 ms bound to address a_2, and the two tasks, t_1 and t_2, are ready to execute. Now, the E machine relinquishes control, only to wake up again 10 ms later. In the meantime, the task scheduler takes over and assigns CPU time to the released tasks according to some scheduling scheme.

At 10 ms the trigger is removed from the trigger queue, and the associated a_2 block is executed. It calls the output driver d_2, which reads a port written by task t_2, and the sensor driver d_s, which updates a port read by t_2. There are two possible scenarios: the earlier invocation of task t_2 may already have completed. In this case, the E code proceeds to release t_2 again and to trigger the a_1 block in another 10 ms, at 20 ms. In this way, the entire process repeats every 20 ms. The other scenario at 10 ms has the earlier invocation of task t_2 still incomplete. In this case, the attempt by the output driver to read a port written by t_2 causes a runtime exception, called *time-safety violation*. At 20 ms, when ports read by both tasks t_1 and t_2 are updated, and ports written by both t_1 and t_2 are read, a time-safety violation occurs unless both tasks have completed. In other words, an execution of the program is time-safe if the scheduler ensures the following: (1) each invocation of task t_1 at $20n$ ms, for $n \geq 0$, completes by $20n+20$ ms; (2) each invocation of t_2 at $20n$ ms completes by $20n + 10$ ms; and (3) each invocation of t_2 at $20n+10$ ms completes by $20n+20$ ms. Therefore, a necessary requirement for time safety is $\delta_1 + 2\delta_2 < 20$, where δ_1 is the WCET of task t_1, and δ_2 is the WCET of t_2. If this requirement is satisfied, then an EDF scheduler, which gives priority to t_2 over t_1, guarantees time safety.

Figure 5 shows an execution trace of the E code using such an EDF scheduler, assuming δ_1 is 12 ms and δ_2 is 4 ms. The second row from the bottom shows the addresses of the E code sequences that are executed at the various time instants. It also shows, for each task, the time since release and the time till termination. This information constitutes what we will call a type in Section 3.

The deadlines of the tasks are implicitly encoded in the E code but may also be given explicitly as tips. For example, the instruction `schedule`$(t_1) : \{t_1 : 20\}$ has the tip $\{t_1 : 20\}$, which assigns a deadline of 20 ms to the task t_1. After release of a task t, the first instruction `call`(d) with $share(d, t)$ is said to (logically) *terminate* the task t, because the task must be completed when the `call` instruction is executed. The tip of a `call` instruction that terminates a task specifies the time that has elapsed since the task was released. More precisely, the tip of the instruction `call`$(d_1) : \{t_1 : 20\}$ from the example states that d_1 shares ports with t_1, and that t_1 was released either 20 ms ago or not at all. The tip of `call`$(d_a) : \{\}$ asserts that the driver d_a does not share ports

```
a_1 : call(d_1) : {t_1 : 20}        a_3 : future(0, a_4) : {t_1}
      call(d_2) : {t_2 : 10}              future(0, a_5) : {}
      call(d_a) : {}                      return()
      call(d_s) : {t_2 : ⊥}         a_4 : call(d_2) : {t_2 : 10}
      call(d_i) : {t_1 : ⊥}               call(d_s) : {t_2 : ⊥}
      schedule(t_1) : {t_1 : 20}          schedule(t_2) : {t_2 : 10}
      schedule(t_2) : {t_2 : 10}          future(10, a_4) : {}
      future(10, a_2) : {}                return()
      return()                      a_5 : call(d_1) : {t_1 : 20}
a_2 : call(d_2) : {t_2 : 10}              call(d_a) : {}
      call(d_s) : {t_2 : ⊥}               call(d_i) : {t_1 : ⊥}
      schedule(t_2) : {t_2 : 10}          schedule(t_1) : {t_1 : 20}
      future(10, a_1) : {}                future(20, a_5) : {}
      return()                            return()
```

Figure 4: E code for the simplified flight controller

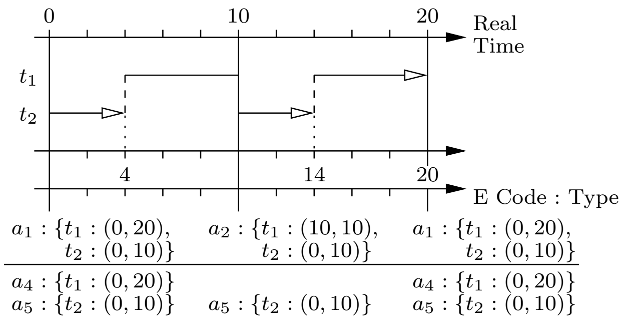

Figure 5: An execution trace of the E code from Figure 4 using an EDF scheduler

with any task. The tip of $\texttt{call}(d_i) : \{t_1 : \bot\}$ means that the driver d_i shares ports with t_1, but t_1 either has not been released or was terminated by a previous `call` instruction.

Finally, a `future` instruction may have a tip that constrains the set of tasks that can be released and terminated by the new thread of E code which follows the instruction. For an example, consider the E code in the right column of Figure 4 starting at the address a_3, which implements exactly the same behavior as the E code in the left column but uses two threads of E code, one for each task. The tip of the instruction $\texttt{future}(0, a_4) : \{t_1\}$ states that (1) the current thread, which continues at the address a_4, may release and terminate all tasks except t_1, and (2) the new thread, which follows the `future` instruction, may release and terminate only t_1. Then, the tip of $\texttt{future}(0, a_5) : \{\}$ asserts that (1) the thread that continues at a_5 may release and terminate t_1, and (2) the newly created thread, which finishes immediately, cannot release or terminate any tasks. Note that both threads (at a_4 and a_5) access different tasks but communicate through the output port p_2 of t_2. The multi-threaded E code produces the same execution trace as the single-threaded E code, as shown in Figure 5. The bottom row of the figure gives the addresses of the E code blocks which are executed in the multi-threaded case.

E code execution

A *configuration* (Q, S, M, I) of an E program consists of a trigger queue Q (containing the active triggers, a task set S (containing the released tasks), a port memory M (specifying the state of all ports), and a sequence I of E code instructions (the current block to be executed). The trigger queue Q is a sequence of pairs (n, I), where n is the offset of a time trigger and I is a sequence of E code instructions which will be executed when the trigger becomes enabled. The task set S is a function that maps a task t to either $\bot$, which indicates that t has completed but not been released again, or to a nonnegative integer, which indicates for how much time t has already executed since its last release.

The E machine consists of an E code interpreter and an event handler that maintains the trigger queue. Moreover, the E machine uses a task scheduler to determine which task among the released tasks executes next. The execution of an E program starts with the initial configuration (Q_0, S_0, M_0, I_0), where Q_0 is the empty queue, $S_0(t) = \bot$ for all tasks, M_0 is the initial state of all ports, and I_0 is the initial block of E code instructions. The E code interpreter implements the E code instructions as follows. A $\texttt{call}(d)$ instruction executes the code of the driver d. Here, the interpreter may check for a possible time-safety violation by verifying that all currently released tasks neither read from driver ports written by d nor write to task ports read by d. If there is a violating task t —i.e., $\mathit{share}(d, t)$— then the interpreter throws a run-time exception: it does not execute the `call` instruction but jumps to some other block of E code which handles the situation [3], e.g., it may terminate t and execute some other driver. A $\texttt{schedule}(t)$ instruction releases the task t by setting $S(t)$ to zero, which indicates that t has been released but not yet executed. Again, the interpreter may check for a possible time-safety violation by verifying that t was completed before releasing it, i.e., $S(t)$ was $\bot$. A $\texttt{future}(n, a)$ instruction appends the pair (n, I) at the end of the trigger queue, where I is the sequence of E code instructions starting at the address a. These E code instructions will be executed after n time units elapse.

The `return` instruction finishes a sequence of E code instructions. Then, if the trigger queue is empty, the E machine stops; otherwise, the first trigger binding (n, I) with the lowest offset n in the queue is selected. If $n = 0$, the trigger binding $(0, I)$ is removed from the queue and the sequence I of E code instructions is executed. If $n > 0$, then the event handler decreases all offsets in the trigger queue by n, and the dispatcher is called to execute tasks for n clock ticks in the order in which the task scheduler selects them from the task set. A task t completes at the latest when it reaches its WCET; at that time $S(t)$ is set to $\bot$. During task execution, the environment may change the state of environment ports.

$$(\mathit{refl})\ \frac{}{\vdash (S, \bot, \bot) \le (S, \bot, \bot)}\ S \subseteq T$$

$$(\mathit{trans})\ \frac{\vdash \Theta \le \Theta' \qquad \vdash \Theta' \le \Theta''}{\vdash \Theta \le \Theta''} \qquad (\mathit{relax})\ \frac{\vdash \Theta \le \Theta' \qquad \Theta \vdash I}{\Theta' \vdash I}$$

$$(\mathit{sub})\ \frac{}{\vdash \Theta\{t : (\delta_c, \delta_r)\} \le \Theta}\ t : (\bot, \bot) \in \Theta, \delta_c, \delta_r \ge 0, \delta_c + \delta_r \ge 1$$

Figure 6: Static semantics (subtyping)

3. THE TYPE SYSTEM

The type system for E code is shown in Figures 6 and 7. The purpose of the type system is to check if a given E program releases and terminates tasks (in finite time) in a path-independent way. Recall that a task is (logically) terminated when a driver is called that shares ports with the task. The type system also checks that tasks are always terminated before they are released again and that different E code threads release and terminate disjoint sets of tasks.

Suppose that, at some time instant during the program execution, the task set contains a task t that has previously been released but not yet terminated. There are two interesting scenarios. First, if the program branches with an `if` instruction, the type system checks that t will be terminated on both branches (in fact, on all future branches) at the same time instant in the future. More precisely, the type system checks that there is a nonnegative integer δ_r, the *remaining* time of task t, such that on all paths, t will be terminated δ_r clock ticks in the future. For schedulability analysis this means that t has the same deadline on all future branches of the program. Second, if the program is at a control location that can be reached from different branches, the type system checks that t has been released at the same

$$\frac{\Theta \vdash I}{\Theta \vdash \texttt{call}(d) : \{t:\bot\}; I}\; t:(\bot,\bot)\in\Theta \qquad \frac{\Theta\{t:(\bot,\bot)\} \vdash I}{\Theta \vdash \texttt{call}(d):\{t:\delta\}; I}\; t:(\delta,0)\in\Theta \qquad \frac{\Theta\{t:(0,\delta)\}\vdash I}{\Theta\vdash \texttt{schedule}(t):\{t:\delta\};I}\; t:(\bot,\bot)\in\Theta$$

$$\frac{(S',\bot,\bot)\vdash I \qquad (S\setminus S', C+n, R-n)\vdash I'}{(S,C,R)\vdash \texttt{future}(n,I'):S';I}\; S'\subset S, \forall_{t\in S'}(C(t)=R(t)=\bot), \forall_{t\in S\setminus S'}(C(t)=R(t)=\bot \vee (C(t)\ge 0 \wedge R(t)\ge n))$$

$$\frac{\Theta\vdash I \qquad \Theta\vdash I'}{\Theta\vdash \texttt{if}(c,I');I} \qquad \frac{\Theta\vdash I}{\Theta\vdash \texttt{jump}(I)} \qquad \frac{}{(S,\bot,\bot)\vdash \texttt{return}()}$$

Figure 7: Static semantics (instructions)

time instant in the past on all branches. More precisely, the type system checks that there is a nonnegative integer δ_c, the *consumed* time, such that on every path, either t was released δ_c clock ticks in the past, or it was not released at all. Note that even an E program without any `if` and `jump` instructions does not necessarily type check, because a `future` instruction may create a violating cycle, or a task may be released but never terminated. In addition to timing information, the type system also restricts the set of tasks that are *available* at each E code thread for being released and terminated. A type error occurs therefore also if two different threads access the same task. For example, the E code in both columns of Figure 4 type checks because both tasks, t_1 and t_2, are released with fixed deadlines of 20 ms and 10 ms, respectively, and in the right column of the figure, the two tasks are accessed by two different threads.

Type checking

Every E program operates on a finite set T of tasks. Let $\mathbb{N}_\bot = \mathbb{N}\cup\{\bot\}$. The type system computes for each control location of the program a *type* of the form (S,C,R) consisting of a set $S\subseteq T$ of available tasks, a function C: $S\to\mathbb{N}_\bot$ that maps every available task to a consumed time, and a function R: $S\to\mathbb{N}_\bot$ that maps every available task to a remaining time, such that for all available tasks $t\in S$, either $C(t)=R(t)=\bot$, or $C(t),R(t)\ge 0$ and $C(t)+R(t)\ge 1$. If $C(t)=R(t)=\bot$, this indicates that the task t has not yet been released (either since the beginning of program execution, or since the last time t was terminated). We overload $\bot$ to denote also the function that maps all available tasks to $\bot$. Types are abbreviated by Θ. The expression $t:(\delta_c,\delta_r)\in\Theta$ states that Θ is a type (S,C,R) with $t\in S$, $C(t)=\delta_c$, and $R(t)=\delta_r$. Then, the expression $\Theta\{t:(\delta_c',\delta_r')\}$ refers to the type (S,C',R') with $C'(t)=\delta_c'$, $R'(t)=\delta_r'$, and for all $t'\in S$, if $t'\ne t$, then $C'(t')=C(t')$ and $R'(t')=R(t')$. The function $C+n$ is defined by $(C{+}n)(t)=\bot$ if $C(t)=\bot$, and $(C{+}n)(t)=C(t){+}n$ otherwise. The function $R-n$ is defined similarly.

There are two kinds of judgments. The first kind of judgment $\vdash\Theta\le\Theta'$ asserts that the type Θ' is less specific than the type Θ. In particular, $\vdash\Theta\le\Theta'$ iff both types are the same except for some tasks t such that t has been released and not yet terminated in Θ, but t has not been released in Θ'. The subtyping relation $\le$ captures the fact that a scenario in which a task has been released but not yet terminated is worse in terms of CPU utilization than one in which the task has not been released. In other words, Θ' can be scheduled if Θ is schedulable. The subtyping relation is computed by the rules of Figure 6 in a standard way.

The second kind of judgment $\Theta\vdash I$ asserts that the E code block I has the type Θ. This relation is computed by the rules of Figure 7, which assume that the instructions in I are already annotated with tips. The rules check that the tips are correct and that types can be assigned to all control locations of the E code. The rules are organized according to E code instructions and propagate information provided by the tips. If the tip of a `call` instruction is $\{t:\bot\}$, then the type system checks that the task t has not been released without being terminated before the `call` instruction. If the tip of a `call` instruction is $\{t:\delta\}$, then the type system checks that t was released δ clock ticks ago and has not been terminated since, and it asserts that t is being terminated now. If the tip of a `schedule` instruction is $\{t:\delta\}$, then the type system checks that t has not been released without being terminated before the `schedule` instruction, and it asserts that t is being released now. If the tip of a $\texttt{future}(n,\cdot)$ instruction is S', then the type system propagates the tasks in S' to the new thread, which starts immediately (without delay) at the subsequent instruction, and it propagates the remaining tasks of the current tread to its continuation after n clock ticks, adjusting the consumed and remaining times of those tasks (in $S\setminus S'$) accordingly. Note that S' is required to be a proper subset of the currently available tasks S; that is, the future continuation of the current thread must retain at least one task. An interesting case occurs when two E code branches meet at some control location a, due to an `if` or `jump` instruction. Then, the type rules for these instructions together with the subtyping rules compute at a the least upper bound (join) of the types of both branches. Note that the type system computes types in time linear in the size of the tip-annotated E code.

As an example, consider the following Giotto program and the E code shown in Figure 8, which has been generated from the program by the Giotto compiler [4]:

```
start m {
  mode m() period 120 ms {          mode n() period 120 ms {
    exitfreq 3 do n(c);               exitfreq 2 do m(c);
    taskfreq 1 do t1();               taskfreq 1 do t1();
    taskfreq 2 do t2();               taskfreq 2 do t2();
    taskfreq 3 do t3(); }             taskfreq 4 do t4(); } }
```

The Giotto program consists of two modes, m and n, which represent different flight situations of the helicopter. For simplicity, we have omitted any sensor, actuator, and task input drivers. The program starts to execute in mode m, in which it checks the condition c every 40 ms in order to determine whether to switch to mode n or not. In mode n, the condition c is checked every 60 ms to determine whether to switch back to mode m. The control location in the E code that requires subtyping to type check is n_4, which can be reached from the E code blocks at n_3 and m_4. In terms of the Giotto program, n_4 is reached either 60 ms into mode n, or 60 ms into mode m when switching to n. In both cases, the task t_4 is terminated at 60 ms. However, only in the first case was t_4 actually released 30 ms earlier. Note that the task t_2 is also terminated at 60 ms, while t_2

```
m1 : call(d1) : {t1 : 120}          n1 : call(d1) : {t1 : 120}
     call(d2) : {t2 : 60}                call(d2) : {t2 : 60}
     call(d3) : {t3 : 40}                call(d4) : {t4 : 30}
     if(c, n2)                           if(c, m2)
m2 : schedule(t1) : {t1 : 120}      n2 : schedule(t1) : {t1 : 120}
     schedule(t2) : {t2 : 60}            schedule(t2) : {t2 : 60}
     schedule(t3) : {t3 : 40}            schedule(t4) : {t4 : 30}
     future(40, m3) : {}                 future(30, n3) : {}
     return()                            return()
m3 : call(d3) : {t3 : 40}           n3 : call(d4) : {t4 : 30}
     if(c, m4)                           schedule(t4) : {t4 : 30}
     schedule(t3) : {t3 : 40}            future(30, n4) : {}
     future(20, m5) : {}                 return()
     return()                       n4 : call(d2) : {t2 : 60}
m4 : future(20, n4) : {}                 call(d4) : {t4 : 30}
     return()                            if(c, m6)
m5 : call(d2) : {t2 : 60}                schedule(t2) : {t2 : 60}
m6 : schedule(t2) : {t2 : 60}            schedule(t4) : {t4 : 30}
     future(20, m7) : {}                 future(30, n5) : {}
     return()                            return()
m7 : call(d3) : {t3 : 40}           n5 : call(d4) : {t4 : 30}
     if(c, m8)                           schedule(t4) : {t4 : 30}
     schedule(t3) : {t3 : 40}            future(30, n1) : {}
     future(40, m1) : {}                 return()
     return()
m8 : future(10, n5) : {}
     return()
```

Figure 8: E code for the 2-mode Giotto program

$$\frac{\Delta : I}{\Delta\{(t,\bot)\} : \texttt{schedule}(t) : \{t : \Delta(t)\}; I} \qquad \frac{\Delta : I'}{\Delta + n : \texttt{future}(n, I'); I}$$

$$\frac{\Delta : I}{\Delta\{(t,0)\} : \texttt{call}(d); I}\ share(d,t) \qquad \frac{\Delta : I}{\Delta : \texttt{if}(c, I'); I} \qquad \frac{\Delta : I}{\Delta : \texttt{jump}(I)}$$

Figure 9: Tip derivation for schedule instructions

may have been released 60 ms earlier by the E code block at m_2 or n_2. In other words, no matter in which mode t_2 was released, t_2 is terminated exactly 60 ms later independently of any mode switching. The same is also true for the task t_1.

Tip derivation

Tips are a space-efficient representation of information that is relevant to compute types. Given E code without tips as well as the driver-task access relation $share(\cdot,\cdot)$, tips can be derived using the rules from Figures 9, 10, and 11.[3] The tips for `schedule` and `future` instructions are computed backward. For `schedule` instructions, we compute a function Δ: $T \rightarrow \mathbb{N}_\bot$ that maps each task to $\bot$ if it is terminated, or to the time till termination. This is achieved by summing up all trigger offsets from the termination of a task back to its release. Note that the `if` rule considers only the false branch. Hence the derived tips may not be consistent with all program paths, which means that the program may not type check. For `future` instructions, we compute a set S that contains the tasks that are available for release in each thread. This is done by collecting all tasks from their termination back to their release. Again, the derived tips may not be consistent with all program paths.

For `call` instructions, the computed function Δ: $T \rightarrow \mathbb{N}_\bot$ maps each task to $\bot$ if it has not been released, or to the time since release. This computation proceeds forward, and is therefore slightly more involved. In particular, the rules of Figure 11 need to be augmented by additional rules (not shown here) which handle joins: at a control location with two incoming branches Δ' and Δ'', we compute the new set Δ by $\Delta(t) = \Delta'(t)$ if $\Delta''(t) = \bot$, and $\Delta(t) = \Delta''(t)$ if $\Delta'(t) = \bot$, and otherwise $\Delta(t)$ is arbitrarily chosen to be $\Delta'(t)$; this can be achieved by subtyping rules similar to those in Figure 6. Finally, if Δ : `call`(d) and $share(d,t)$, then the tip for the `call`(d) instruction is $\{t : \Delta(t)\}$.

$$\frac{S : I \quad S' : I'}{S \cup S' : \texttt{future}(n, I') : S'; I} \qquad \frac{S : I}{S \cup \{t\} : \texttt{call}(d); I}\ share(d,t)$$

$$\frac{S : I}{S \cup \{t\} : \texttt{schedule}(t); I} \qquad \frac{S : I \quad S' : I'}{S \cup S' : \texttt{if}(c, I'); I} \qquad \frac{S : I}{S : \texttt{jump}(I)}$$

Figure 10: Tip derivation for future instructions

Tips can be derived in time linear in the size of the E code. We say that an E program is *typed* if after tip derivation, the typing rules succeed in deriving a judgment of the form $\Theta \vdash I_0$, where I_0 is the initial block of E code. The check if an E program is typed (tip derivation followed by type checking) is linear in the size of the program.

4. SCHEDULABILITY CHECKING

Given a typed E program P over a finite set T of tasks, and WCETs for the tasks in T, we present a sufficient condition for the schedulability of P. If P is generated from a Giotto program without redundant modes, then the condition is also necessary. The schedulability condition can be checked in time $O(m \cdot (c+1)^{k-1})$, where m is the number of instructions and k is the number of threads of P, and c is the largest trigger offset that occurs in P. We also show that if the condition is satisfied, then all deadlines are met by an EDF schedule. This is of practical importance, because for typed programs, it is simple to implement an EDF schedule. The schedulability analysis of a typed E program P is based on the notion of a timing-flow graph for P, which is induced by the control-flow graph for P. The analysis is abstract in that all syntactic program paths are considered feasible.

Control-flow graphs

Let T be a set of tasks and let P be an E program over T. The *control-flow graph* $F_P = (L, a_0, E, s)$ for P consists of the following components:

- A set L of control locations, an initial location $a_0 \in L$, and a set $E \subseteq L \times L$ of directed edges.
- A function s that labels each edge with a (non-control-flow) instruction: for all edges $(a, a') \in E$, we have either $s(a,a') =$ `call`(d) for a driver d, Or $s(a,a') =$ `schedule`(t) for a task $t \in T$, or $s(a,a') =$ `future`(n, a'') for a trigger $n \in \mathbb{N}$ and a location $a'' \in L$.

The control-flow graph F_P is abstract: it contains no information about port values. If the program P contains an `if` instruction at address a, then the location a of F_P has two nondeterministic successors; if P contains a `return` instruction at address a, then the location a of F_P has no successor;

$$\frac{\Delta : \texttt{call}(d); I}{\Delta\{(t,\bot)\} : I}\ share(d,t) \qquad \frac{\Delta : \texttt{schedule}(t); I}{\Delta\{(t,0)\} : I} \qquad \frac{\Delta : \texttt{future}(n, I'); I}{\bot : I \qquad \Delta + n : I'}$$

$$\frac{\Delta : \texttt{if}(c, I'); I}{\Delta : I \qquad \Delta : I'} \qquad \frac{\Delta : \texttt{jump}(I)}{\Delta : I} \qquad \frac{\Delta : \texttt{call}(d); I}{\Delta : I}\ \forall_{t \in T}\, \neg share(d,t)$$

Figure 11: Tip derivation for call instructions

[3]Rules like the last rule of Fig. 11, saying that tip derivation is not influenced by `call`(d) instructions if $\neg share(d,t)$ for all tasks $t \in T$, must be added also to Figures 9 and 10.

all other locations of F_P have exactly one successor. If the program P is multi-threaded, then the control-flow graph F_P may consist of several subgraphs —one for each thread— which are not connected with each other.

If the program P is typed, then we can associate a type with every location of F_P. Recall that a *type* (S, C, R) is a triple consisting of a set $S \subseteq T$ of available tasks, a function C: $S \to \mathbb{N}_\perp$ that maps every available task to a consumed time, and a function R: $S \to \mathbb{N}_\perp$ that maps every available task to a remaining time, such that for all available tasks $t \in S$, either $C(t) = R(t) = \perp$, or $C(t), R(t) \geq 0$ and $C(t) + R(t) \geq 1$. A *typed control-flow graph* $\langle F_P, \theta\rangle$ consists of the control-flow graph F_P for a typed program P, together with a function θ that maps every location of F_P to a type. We write $(S(a), C(a), R(a))$ for the type $\theta(a)$ of a location $a \in L$. The type system enforces that once $S(a) = \emptyset$, only `call` instructions can be performed.

Timing-flow graphs

The edges of a control-flow graph represent E code instructions, which are executed in zero time. The real-time behavior of an E program is made explicit in the so-called timing-flow graph, which in addition to instruction edges also contains edges that represent the passage of time. Informally, the vertices L^τ of the timing-flow graph F_P^τ for the program P are trigger configurations, each consisting of a control location and a trigger queue, and the edges are either (1) instruction edges E^τ labeled by the E code instructions of P, or (2) zero-time edges τ_0 indicating the removal of triggers from the trigger queue, or (3) unit-time edges τ_1 indicating the advance of time. Formally, the control-flow graph $F_P = (L, a_0, E, s)$ induces a *timing-flow graph* $F_P^\tau = (L^\tau, a_0^\tau, E^\tau, s^\tau, \tau_0, \tau_1)$ with the following components:

- A set $L^\tau = L \times \mathit{Queues}(\mathbb{N} \times L)$ of trigger configurations $\langle a, q\rangle$, each consisting of a control location a and a trigger queue q. The trigger queue q is a queue of trigger bindings $(n, a') \in \mathbb{N} \times L$. The trigger binding (n, a') indicates that after n time units, control jumps to the location a'.

- An initial configuration $a_0^\tau = \langle a_0, \emptyset\rangle$, where $\emptyset$ is the empty queue.

- A set $E^\tau \subseteq L^\tau \times L^\tau$ of instruction edges, labeled by the function s^τ, such that $(\langle a, q\rangle, \langle a', q'\rangle) \in E^\tau$ if one of the following: (1) $(a, a') \in E$ and $s(a, a') = \texttt{call}(d)$ and $q' = q$. In this case, $s^\tau(\langle a, q\rangle, \langle a', q'\rangle) = \texttt{call}(d)$. (2) $(a, a') \in E$ and $s(a, a') = \texttt{schedule}(t)$ and $q' = q$. In this case, $s^\tau(\langle a, q\rangle, \langle a', q'\rangle) = \texttt{schedule}(t)$. (3) $(a, a') \in E$ and $s(a, a') = \texttt{future}(n, a'')$ and $q' = q \cdot (n, a'')$; that is, the trigger binding (n, a'') is entered at the end of the queue q. In this case, $s^\tau(\langle a, q\rangle, \langle a', q'\rangle) = \texttt{future}$.

- A set $\tau_0 \subseteq L^\tau \times L^\tau$ of zero-time edges such that $(\langle a, q\rangle, \langle a', q'\rangle) \in \tau_0$ if (1) there is no E^τ edge outgoing from $\langle a, q\rangle$, (2) the queue q contains a trigger binding of the form $(0, \cdot)$, (3) the queue q' results from q by removing the first trigger binding of the form $(0, \cdot)$, and (4) the trigger binding that is removed from q has the second component a'; that is, $q' = q \backslash \{(0, a')\}$.

- A set $\tau_1 \subseteq L^\tau \times L^\tau$ of unit-time edges such that $(\langle a, q\rangle, \langle a', q'\rangle) \in \tau_1$ if (1) there is no E^τ or τ_0 edge outgoing from $\langle a, q\rangle$, (2) $a' = a$, and (3) the queue q' results from q by replacing every trigger binding $(n, \cdot)$ by $(n-1, \cdot)$. The decrements are all nonnegative, because by condition (1), the queue q contains no trigger binding of the form $(0, \cdot)$.

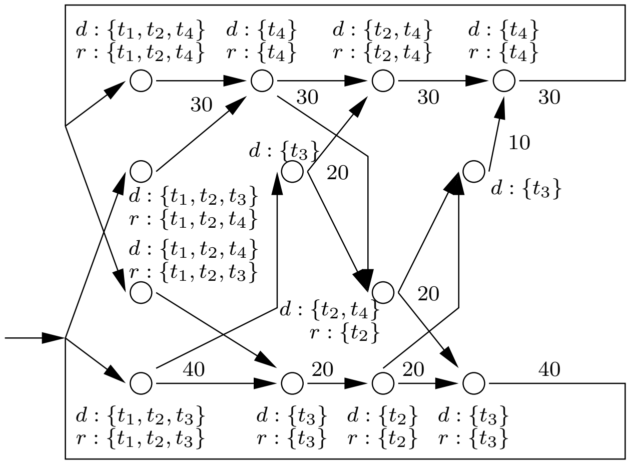

Figure 12: Condensed timing-flow graph for the E code in Figure 8

Figure 12 shows a condensed version of the timing-flow graph for the E code in Figure 8. In the figure, paths of consecutive unit-time edges are condensed into single (weighted) edges, and paths of consecutive instruction and zero-time edges are condensed into vertices. Each vertex is labeled with the set d of tasks that are terminated and the set r of tasks that are released at the vertex.

The execution of the program P corresponds to a path through the timing-flow graph F_P^τ, where time advances by one unit whenever a τ_1 edge is traversed. A configuration $\langle a, q\rangle \in L^\tau$ has two successors in F_P^τ iff the location $a \in L$ has two successors in the control-flow graph F_P. These configurations correspond to `if` instructions in the program P. All other configurations in L^τ have zero or one successors; that is, all nondeterminism in the timing-flow graph F_P^τ is caused by abstracting the port values. The program execution may either finish, by reaching a configuration without successors, or run forever. Thus, the relevant part of the timing-flow graph F_P^τ is its reachable subgraph, defined as follows. A *path* of the program P is a finite sequence $r_0 r_1 \ldots r_n$ of configurations $r_i \in L^\tau$ such that $r_0 = a_0^\tau$ and for all $0 \leq i < n$, we have $(r_i, r_{i+1}) \in E^\tau \cup \tau_0 \cup \tau_1$. A configuration $\hat{r} \in L^\tau$ is *reachable* if there is a path $r_0 r_1 \ldots r_n$ of P such that $r_n = \hat{r}$. The restriction of the timing-flow graph F_P^τ to the reachable configurations is denoted $\hat{F}_P^\tau$.

If the program P is typed, then the reachable configurations $\langle a, q\rangle$ of the timing-flow graph F_P^τ have a special form: each location can occur at most once in $\langle a, q\rangle$; i.e., the locations that occur in the trigger queue q are all pairwise distinct and different from a. This is implied by the following lemma and by the property of typed control-flow graphs that every `future` instruction provides at least one available task to the future continuation of the current thread.

Typing lemma. *If the program P is typed, then every reachable configuration $\langle a, q\rangle$ of the timing-flow graph F_P^τ satisfies the following three conditions. First, if $(\cdot, a')$ is a trigger binding in the queue q, then $S(a) \cap S(a') = \emptyset$.*

Second, if $(\cdot, a')$ and $(\cdot, a'')$ are two distinct trigger bindings in q, then $S(a') \cap S(a'') = \emptyset$. Third, if (n, a') is a trigger binding in q and $C(a')(t) \neq \bot$, then $C(a')(t) \geq n$. $\Box$

The typing lemma follows from the way the type system propagates the set of available tasks: with each `future` instruction the available tasks are partitioned into two disjoint sets —one for the future continuation of the current thread, and the other for the newly created thread.

The typing lemma implies that for typed programs P, the reachable subgraph $\hat{F}_P^\tau$ of the timing-flow graph is finite. However, $\hat{F}_P^\tau$ may be exponentially larger than the control-flow graph F_P, whose size is linear in the size of the program P. The exponential is caused by concurrency. To see this, recall that threads are created by `future` instructions: a typed E program P has k *threads* if P contains $k-1$ `future` instructions whose tips are nonempty. In particular, P is single-threaded iff all `future` instructions of P make no tasks available to the newly created thread, which means that the newly created threads can only perform `call` instructions (typically for data "clean-up"). The typing lemma implies that if P is typed and has k threads, then for every reachable configuration $\langle a, q\rangle$ of the timing-flow graph F_P^τ, the queue q contains at most $k-1$ trigger bindings. Moreover, for each trigger binding $(n, \cdot)$ in q, if c is the largest offset that occurs in a `future` instruction of P, then $0 \leq n \leq c$. Consequently, if a typed program P has m instructions and k threads, and all time triggers are bounded by c, then the reachable subgraph $\hat{F}_P^\tau$ of the timing-flow graph has size $O(m \cdot (c+1)^{k-1})$.

The typing lemma ensures that for typed programs, we can associate with every configuration $\langle a, q\rangle$ of $\hat{F}_P^\tau$ a global type, which is obtained by glueing together the type of a and the types of the locations that occur in the trigger queue q. A *global type* (C, R) is a pair consisting of a function C: $T \to \mathbb{N}_\bot$ that maps every task to a consumed time, and a function R: $T \to \mathbb{N}_\bot$ that maps every task to a remaining time, subject to the usual constraint that for all tasks $t \in T$, either $C(t) = R(t) = \bot$, or $C(t), R(t) \geq 0$ and $C(t) + R(t) \geq 1$. A *typed timing-flow graph* $\langle \hat{F}_P^\tau, \theta^\tau\rangle$ consists of the reachable subgraph $\hat{F}_P^t$ of the timing-flow graph for a typed program P, together with a function θ^τ that maps every configuration $r = \langle a, q\rangle$ of $\hat{F}_P^\tau$ to the following global type (C, R): (1) if $t \in S(a)$, then $C(r)(t) = C(a)(t)$ and $R(r)(t) = R(a)(t)$; (2) if $t \in S(a')$ and $(n, a') \in q$, then $C(r)(t) = C(a')(t) - n$ and $R(r)(t) = R(a')(t) + n$ (the difference $C(a')(t) - n$ is nonnegative because of the third condition of the typing lemma; we assume $\bot \pm n = \bot$); and (3) in all other cases, $C(r)(t) = \bot$ and $R(r)(t) = \bot$ (this applies to tasks that are available neither in a nor in any of the locations that occur in the trigger queue q). We write $(C(r), R(r))$ for the global type $\theta^\tau(r)$ of a reachable configuration $r \in L^\tau$. A task $t \in T$ is *active* at a reachable configuration r if $C(r)(t) \neq \bot$; that is, the task has been released but not yet terminated. We write $A(r) \subseteq T$ for the set of tasks that are active at r.

Note that the reachable timing-flow graph $\hat{F}_P^\tau$ may contain zero-time cycles, i.e., cycles that contain no τ_1 edges. Each zero-time cycle either is inherited from the control-flow graph F_P, in which case it consists of E^τ edges only, or it is caused by `future` instructions, in which case it contains some τ_0 edges. As all port values are abstracted, zero-time cycles may be present even if P is *time-live*, i.e., if every program execution traverses infinitely many τ_1 edges. We are not concerned here with time-liveness, as the issue is orthogonal to scheduling (i.e., time-safety) and not present in E code generated from Giotto programs [3].

Schedulability test

A *scheduling point* of a program P is a reachable configuration $r \in L^\tau$ of the timing-flow graph such that r has an outgoing τ_1 edge in F_P^τ, which advances time (if this is the case, then r cannot have any other outgoing edges). At each scheduling point, a scheduler can decide which tasks to execute during the time unit that corresponds to the outgoing edge. We write $X \subseteq L^\tau$ for the set of scheduling points. A *time-based schedule* for the program P is a function σ: $X \to (T \to [0,1])$ that maps every scheduling point r and every task t to a real number $\sigma(r,t) \in [0,1]$ such that $\sum_{t\in T} \sigma(r,t) \leq 1$. The real $\sigma(r,t)$ indicates how much CPU time is assigned by the schedule to the task t during the time unit that follows the scheduling point r. A *priority-based schedule* for P is a function π: $X \to \mathit{Perms}(T)$ that maps every scheduling point r to a permutation $\pi(r)$ of the tasks in T. If $\pi(r) = \langle t_1, \ldots, t_m\rangle$, then during the time unit that follows the scheduling point r, the first task t_1 is executed until either the task completes or the time unit expires, whichever happens first. If t_1 completes before the time unit expires, then the second task t_2 is executed next, etc. For two tasks $t, t' \in T$ and a permutation $\pi \in \mathit{Perms}(T)$, we write $t \succ_\pi t'$ if t precedes t' in π; that is, t has a higher priority than t'. Both time-based and priority-based schedules, as defined here, are memoryless, in that the scheduling decision at a scheduling point r does not depend on the path leading up to r, and execution-time independent, in that the scheduling decision at r does not depend on which tasks have already completed their execution.

Consider a finite path $\rho = r_0 r_1 \ldots r_n$ of the program P. A task $t \in T$ is *released* (resp. *terminated*) at the position $0 \leq i < n$ of ρ if $(r_i, r_{i+1}) \in E^\tau$ and $s(r_i, r_{i+1}) = \texttt{schedule}(t)$ (resp. $s(r_i, r_{i+1}) = \texttt{call}(d)$ such that $\mathit{share}(d, t)$). An *execution* $\langle \rho, \lambda\rangle$ of P consists of a finite path ρ together with a function λ: $\{0, \ldots, n-1\} \to (T \to [0,1])$ that maps every position $0 \leq i < n$ of ρ and every task $t \in T$ to a real $\lambda(i,t) \in [0,1]$ such that the following three conditions hold: (1) $\sum_{t\in T} \lambda(i,t) \leq 1$; (2) if for all $0 \leq j \leq i$, the task t is not released at j, then $\lambda(i,t) = 0$; and (3) if $(r_i, r_{i+1}) \in E^\tau \cup \tau_0$, then $\lambda(i,t) = 0$. If $(r_i, r_{i+1}) \in \tau_1$, then $\lambda(i,t)$ indicates how much CPU time is spent on the task t during the corresponding time unit. The execution $\langle \rho, \lambda\rangle$ is *time-safe* if for all tasks $t \in T$ and all positions $0 \leq i < n$, if t is terminated at i, then either $\lambda(j,t) = 0$ for all $i \leq j < n$, or there exists a position $i < k < n$ such that t is released at k and $\lambda(j,t) = 0$ for all $i \leq j < k$. In other words, a time-safety violation occurs if a task t is executed after a $\texttt{call}(d)$ instruction, where d is a driver that shares ports with t, but before the next $\texttt{schedule}(t)$ instruction; in this case the task is not completed at the time of the `call` instruction.

According to a time-based schedule σ, a task $t \in T$ is *completed* at the position $0 \leq i < n$ of $\langle \rho, \lambda\rangle$ if r_i is a scheduling point and $\lambda(i,t) < \sigma(r_i, t)$; that is, t does not use the full amount of time assigned by the scheduler, and therefore must be completed. According to a priority-based schedule π, a task $t \in T$ is *completed* at the position $0 \leq i < n$ of $\langle \rho, \lambda\rangle$ if r_i is a scheduling point and there exists a task $t' \in T$ such that $t \succ_{\pi(r_i)} t'$ and $\lambda(i, t') > 0$; that is,

t would have priority over a task that uses the CPU, and therefore must be completed. The execution $\langle \rho, \lambda \rangle$ *complies* with a (time-based or priority-based) schedule ψ if for all tasks $t \in T$ and all positions $0 \leq i < n$, if t is completed at i according to ψ, then either $\lambda(j,t) = 0$ for all $i < j < n$, or there exists a position $i < k < n$ such that t is released at k and $\lambda(j,t) = 0$ for all $i < j < k$. In other words, once a task t is completed, it cannot be executed until the next `schedule`(t) instruction.

A *WCET map* w for a set T of tasks is a function w: $T \to \mathbb{R}_{\geq 0}$ that maps every task t to a nonnegative real $w(t)$, which represents the worst-case execution time of t on a specific CPU. The *interim execution time* $\Lambda(i,t) \in \mathbb{R}_{\geq 0}$ of a task $t \in T$ at a position $0 \leq i < n$ of an execution $\langle \rho, \lambda \rangle$ is defined inductively as follows: (1) $\Lambda(0,t) = 0$; and (2) if t is released at the position $i+1$ of ρ, then $\Lambda(i+1,t) = 0$; otherwise, $\Lambda(i+1,t) = \Lambda(i,t) + \lambda(i,t)$. Thus the interim execution time $\Lambda(i,t)$ accumulates all execution times $\lambda(i,t)$ from the last release of t to position i. The execution $\langle \rho, \lambda \rangle$ *complies* with the WCET map w if $\Lambda(i,t) \leq w(t)$ for all tasks $t \in T$ and all positions $0 \leq i < n$. A (time-based or priority-based) schedule ψ for P is *correct* with respect to the WCET map w if every execution of P which complies with both ψ and w is time-safe. The program P is *time-based schedulable* for the WCET map w if there exists a time-based schedule for P which is correct with respect to w. The following lemma gives a sufficient condition for the time-based schedulability of typed programs.

Time-based schedulability lemma. *Let P be a typed E program over a set T of tasks, and let w be a WCET map for T. The program P satisfies the* utilization criterion *for w if all scheduling points r of P satisfy*

$$\sum_{t \in A(r)} \frac{w(t)}{C(r)(t) + R(r)(t)} \leq 1.$$

If P satisfies the utilization criterion for w, then P is time-based schedulable for w. $\square$

Recall that $A(r)$ is the set of active tasks at r, and $C(r)(t)$ and $R(r)(t)$ are the consumed and remaining times of task t at r. The proof of the time-based schedulability lemma relies on the typing lemma. There, it is also shown that the number of scheduling points, and thus the number of utilization tests, is linear in the number of instructions and exponential in the number of threads (which is typically small). If P is not typed, then the time-based schedulability lemma does not hold. If the utilization criterion is satisfied, then a correct time-based schedule is $\sigma(r,t) = w(t)/(C(r)(t) + R(r)(t))$; that is, in each time unit, the schedule assigns a time slice to all tasks that have been released but not yet terminated. The utilization criterion is not a necessary condition for schedulability. However, if P is generated from a Giotto program without redundant (i.e., unreachable) modes, then the utilization criterion is both sufficient and necessary for time-based schedulability [4].

EDF scheduling

Let P be a typed E program. A priority-based schedule π for P is an *EDF schedule* if for every scheduling point r of P and all tasks $t, t' \in T$, if $R(r)(t) < R(r)(t')$, then $t \succ_{\pi(r)} t'$; that is, if t has less remaining time than t', then t has a higher priority than t'. The typed program P is *EDF schedulable* for the WCET map w if all EDF schedules for P are correct with respect to w. The following theorem shows that the utilization criterion implies EDF schedulability.

EDF schedulability theorem. *Let P be a typed E program over a set T of tasks, and let w be a WCET map for T. If P satisfies the utilization criterion for w, then P is EDF schedulable for w.* $\square$

The proof relies on the time-based schedulability lemma and the fact that for each execution, a correct time-based schedule can be converted, step by step, into an EDF schedule [6].

5. CONCLUSION

We have presented a type system for E code that checks, given an E program, whether the task deadlines specified in the program are path-insensitive. In other words, the type system checks for each task invocation in the E program whether the task has the same deadline on all future program paths. We have shown that typed E programs allow, for given WCETs of tasks, a simple schedulability analysis when using an EDF scheduler. E code may also be executed in conjunction with scheduling strategies other than EDF. However, for this purpose, appropriate schedulability tests have yet to be identified.

The real-time programming language Giotto can be compiled into typed E code, which identifies an easily schedulable yet expressive class of real-time programs. We have extended the Giotto compiler to generate typed E code, and enabled the run-time system for E code to perform a type and schedulability check before executing the code. Thus, the run-time system need not trust the compiler in order to gain assurance, provided the WCET information is accurate, that all deadlines specified by the E code are met.

6. REFERENCES

[1] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *Proc. EMSOFT*, LNCS 2211, pp. 469–485. Springer, 2001.

[2] T.A. Henzinger, B. Horowitz, C.M. Kirsch. Giotto: A time-triggered language for embedded programming. In *Proc. IEEE* 91(1):84–99, 2003.

[3] T.A. Henzinger and C.M. Kirsch. The embedded machine: Predictable, portable real-time code. In *Proc. PLDI*, pp. 315–326. ACM, 2002.

[4] T.A. Henzinger, C.M. Kirsch, R. Majumdar, S. Matic. Time-safety checking for embedded programs. In *Proc. EMSOFT*, LNCS 2491, pp. 76–92. Springer, 2002.

[5] T.A. Henzinger, C.M. Kirsch, S. Matic. Schedule-carrying code. In *Proc. EMSOFT*, LNCS 2855, pp. 241–256. Springer, 2003.

[6] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM* 20(1):46–61, 1973.

[7] S. Mork, K. Larsen, H.R. Andersen, P. Sestoft. PMC: A programming language for embedded systems. In *Proc. Int. Workshop Formal Methods for Industrial Critical Systems*, July 1999.

[8] G. Morrisett, D. Walker, K. Crary, N. Glew. From System F to typed assembly language. In *ACM TOPLAS*, 21(3):528–569, 1999.