Principles of Real-Time Programming*

Christoph M. Kirsch

Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
cm@eecs.berkeley.edu

Abstract. Real-time programming is a software engineering discipline that has been around ever since the dawn of digital computing. The dream of real-time programmers is to unlock the virtually unlimited potential of software for embedded computer systems – digital computers that are supposed to behave like analog devices. The perfect embedded computer system is invisibly hybrid, it works according to the largely unidentified laws of embedded software but acts according to the laws of physics. The critical interface between embedded software and physics is real-time and yet, while physical processes evolve in real-time, software processes do not. Only the embedded computer system as a whole – embedded software and hardware – determines a complex notion of so-called *soft-time* to which the software processes adhere: mapping soft-time to real-time is the art of real-time programming. We discuss various real-time programming *models* that support the development of real-time *programs* based on different abstractions of soft-time. We informally introduce a real-time process model to study (1) the *compositionality* of the real-time programming models and (2) the *semantics* of real-time programs developed in these models.

1 Introduction

Figure 1 shows an example of a (distributed) software process that interacts with a physical process. At some real-time instant, the software process takes some input from the physical process. Some time later, when the software process is finished processing the input it will return the result of its efforts to the physical process. From the very beginning of taking the input until the very end of returning the output, the software process evolves in soft-time. From the perspective of the physical process, only *at* the real-time instants of input and output, soft-time becomes real-time. In *between* input and output, soft-time can be amazingly complex in relation to real-time when seen from the software perspective. In the example, right after taking the input the software process executes on some processor $\mathcal{A}$. After a while the process is granted a semaphore that controls the access to some shared resource. Handling the semaphore briefly delays the software process. Unavailable semaphores may result in longer and

* Supported in part by the DARPA SEC grant F33615-C-98-3614 and the MARCO GSRC grant 98-DT-660.

A. Sangiovanni-Vincentelli and J. Sifakis (Eds.): EMSOFT 2002, LNCS 2491, pp. 61–75, 2002.
© Springer-Verlag Berlin Heidelberg 2002

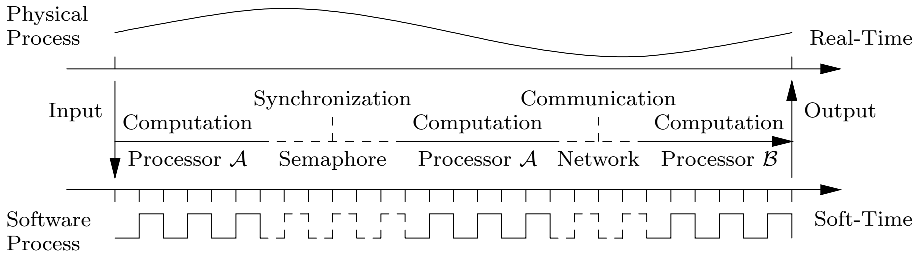

Fig. 1. Real-Time and Soft-Time

sometimes even unbounded delays: soft-time is *discontinuous*, unlike real-time. Then the software process continues to execute on processor $\mathcal{A}$ and computes an intermediate result that is transmitted across a network to some processor $\mathcal{B}$. Just like semaphores, network access also contributes to the discontinuity of soft-time. Before returning the output to the physical process, the software process now runs on processor $\mathcal{B}$. The real-time instant when it completes or, in other words, the relation between soft-time and real-time depends on a whole variety of factors: hardware performance and utilization, scheduling strategy and communication protocol, program and compiler optimizations. Improving any of these factors does not necessarily speed up soft-time. In fact, it can even slow soft-time down.

Software processes are not *composable* with respect to soft-time. For example, composing a software process $\mathcal{P}$ with a software process $\mathcal{Q}$ that shares some resources with $\mathcal{P}$ may result in a system with unbounded soft-time because of a *deadlock* although both processes run separately in bounded soft-time: the system behavior depends on their relative execution speed – a *race condition* on soft-time. An unrelated process $\mathcal{U}$, which may unintentionally yet non-deterministically delay $\mathcal{P}$ or $\mathcal{Q}$, can make the difference between good and bad. Programmers sometimes refer to $\mathcal{U}$ as "heavy load" under which the system exhibits unexpected behavior. In the presence of real-time constraints this scenario is known as the *priority inversion* phenomenon: a low-priority $\mathcal{Q}$ gains access to a shared resource before a high-priority $\mathcal{P}$ asks for it giving a medium priority $\mathcal{U}$ the chance to become the real winner by delaying $\mathcal{Q}$ and thus $\mathcal{P}$.

Yet real-time programming requires *compositional* models: the physical world is *concurrent*, so is embedded software; embedded hardware is *distributed*, so is embedded software. Section 2 discusses real-time programming models that address compositionality in different ways. Non-real-time, sequential programming models typically support a notion of procedural, functional, or logical composition of programs that are compatible in some sense, e.g., type-safe. Valid implementations are supposed to either compute reproducible results, or raise an exception – in case the compatibility check was wrong –, or may not even terminate. In Section 3, we follow this concept and introduce informally a real-time process model that lifts program composition to process composition: processes composed in that model compute, given a sequence of inputs, the same sequence of outputs (*value-determinism*) at the same time (*time-determinism*) provided

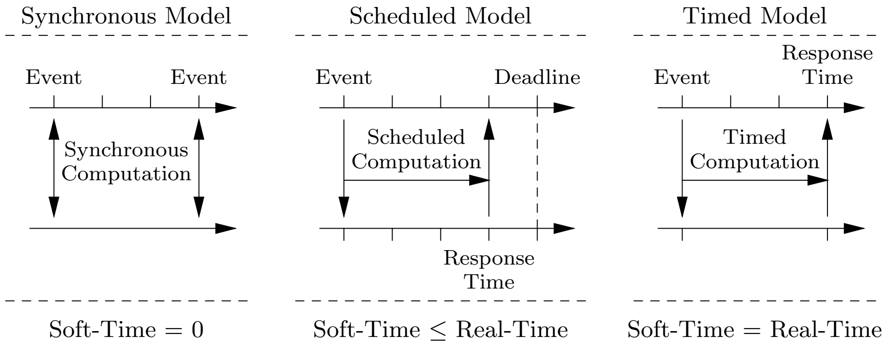

Fig. 2. Models for Real-Time Programming

the composition preserves the process timing (*time-invariant*) and is schedulable (*time-safe*), and the individual processes are value- and time-deterministic. In Section 4, we use this process model to study the semantics of real-time programs developed in the real-time programming models of Section 2.

2 Models for Real-Time Programming

This section presents three real-time programming models, compares their fundamental characteristics, and discusses typical application areas. We distinguish the *synchronous*, *scheduled*, and *timed* programming models shown in Figure 2. The selection is limited and by no means complete. Models for real-time programming have also been discussed elsewhere, e.g., in [10] or [14].

Synchronous Model. The principle of synchronous programming is based on the idea of zero time computation, see, e.g., [4]. The synchronous programmer assumes that any computational activity of a synchronous program including communication takes no time: soft-time in a synchronous program is always zero. A synchronous program is executed in the context of some physical or computational process that generates events as stimulus for the program. Figure 2 shows an example of an event timeline in the synchronous model. A synchronous program *reacts* to events in zero time by computing instantaneously some output (*reaction*) based on the input and control state of the program. A synchronous program is *deterministic* if it computes at most one reaction for any event and control state, and *reactive* if it computes at least one reaction for any event and control state, i.e., if it terminates. The *reactivity* of a synchronous program, i.e., the instantaneous and deterministic reactions to some stimulus, is what concerns the synchronous programmer. Synchronous programming is therefore often referred to as *synchronous reactive programming.* The problem of a compiler for synchronous programs is to implement reactivity. Since cyclic language constructs are present in many synchronous programming languages, the compiler typically needs to prove the existence of finite fixed-points or, in other words,

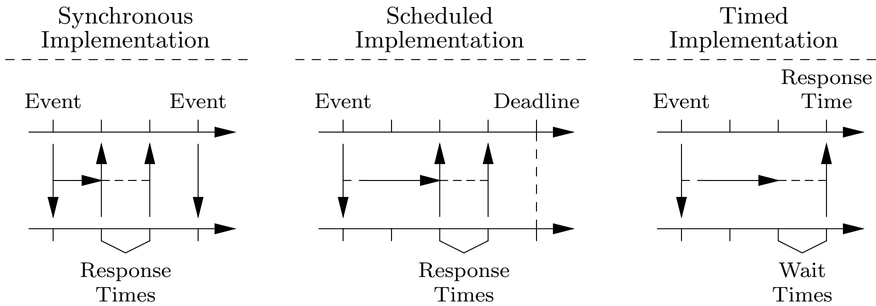

Fig. 3. Implementations of Synchronous, Scheduled, and Timed Programs

the absence of infinite cycles in a synchronous program. Depending on the representation of the control state of a synchronous program, proving reactivity is complex – in languages with explicit control flow such as Esterel [1] – or less difficult – in data-flow languages such as Lustre [5].

In the programmer's mind mapping soft-time in the synchronous model to real-time is simple. A synchronous and reactive program always runs at the speed of its context and, if the program is deterministic, even its output is determined by the behavior of its context. From the perspective of the (physical) context the behavior of a deterministic and reactive program in the synchronous model is perfect. Context and program are *synchronous*. An implementation of a synchronous program may approximate *synchrony* by computing any reaction to an event before the next event occurs while the exact time when a reaction is completed may vary as shown in Figure 3. Thus a compiler for synchronous programs ideally implements not only code generation but also a prover for reactivity and synchrony. A difficult part of showing synchrony is to estimate the (worst-case) execution times of program code [12].

Scheduled Model. The classical discipline of real-time programming [15] in languages such as Ada or, more recently, in Real-Time Java is based on the concept of scheduled computation. A scheduled program typically consists of processes, threads, or tasks. A scheduler determines which part of a scheduled program runs at what time. Soft-time in the scheduled model is the time it takes for some computational activity to complete. For example, the soft-time of a software task is the *response time* of the task. Thus soft-time is not abstract as in the synchronous model. Soft-time may vary depending on performance, utilization, and scheduling scheme and is not a priori determined in the scheduled model. Instead, soft-time in a scheduled program is constrained by real-time *deadlines*: soft-time must be less or equal than real-time. Figure 2 shows for the scheduled model an event that triggers some scheduled computation, which completes some time before its deadline. The exact time when it completes may vary in the implementation as shown in Figure 3. A compiler for a scheduled program typically implements only the functionality but not the scheduling, which is left to the runtime system, i.e., the scheduler of a real-time operating system. The

schedulability of a scheduled program, i.e., the fact that all deadlines are met, must be shown according to the scheduling scheme used in the real-time operating system. A schedulability analysis also requires the analysis of (worst-case) execution times.

The deadlines in a scheduled program make the scheduled model a real-time programming model. *Hard* real-time deadlines must be met under any circumstances whereas *soft* real-time deadlines leave scheduled computation some time to complete after a deadline, which may result in degraded but still acceptable performance. There are many soft real-time applications such as video and audio processing. Application areas such as process control or embedded control systems require hard deadlines and are often mission- or even safety-critical. Hard and soft deadlines do not require different compilers but different schedulers. Much work has been devoted to scheduling theory and practice as well as, more recently, to the problem of hybrid schedulers for scheduled programs with hard and soft deadlines [3]. In practice, however, designers of hard real-time systems have mostly resisted to adopt complex schedulers and still rely on well-understood scheduling schemes such as rate-monotonic scheduling [11]. Mechanisms like semaphores that can cause blocking – unbounded soft-time – are often banned from hard real-time designs. Section 4 gives for more details.

Timed Model. The principle of *timed programming* is based on the idea of computation and communication that logically takes a fixed, non-zero amount of time no matter how much time it actually takes: soft-time in a timed program is equal to real-time. Figure 2 shows for the timed model an event that triggers some timed computation, which completes exactly at the specified time. The timed programmer specifies the real-time it takes for a timed program to compute some output assuming that enough soft-time is available, i.e., the timed program is *time-safe* [7]. Time safety depends on performance, utilization, and scheduling scheme. A compiler for timed programs checks time safety, i.e., if enough soft-time is available and rejects programs if not. Checking time safety is difficult [8] – it requires schedulability tests and execution time analyses – and may not always be feasible at compile-time. If the timed program is indeed late, a runtime exception may be thrown [7]. If the timed program finishes early, it delays its output until the specified real-time has elapsed as shown in Figure 3. The closer the implementation of the timed program outputs at the specified time, the better the implementation approximates the timed model. A timed program may run in the context of some physical or some other computational process. Similar to the synchronous model, the behavior of a deterministic and time-safe program in the timed model is perfect from the perspective of its context.

The timed model is well-suited for embedded control systems, which require timing predictability, e.g., for precise, low jitter input and output. The timed programming language Giotto [6] supports the development of embedded control systems [9], in particular, on distributed hardware. The key element in Giotto is a *timed task* that is *periodic* and *deterministic* with respect to its input and state. The logical execution time of a Giotto task is the period of the task. For example, a 10Hz Giotto task runs logically for 100ms before its output becomes

available and its input is updated for the next invocation. As a consequence of the timed semantics, the behavior of a time-safe Giotto program is *determined* by its (physical) context, not by performance, utilization, or scheduling scheme. Since the parallel composition of timed programs does not change the timing behavior of the individual programs in the timed model, Giotto is *modular*: Giotto programs can be composed from smaller programs and Giotto programs can be compiled separately and even incrementally.

3 A Real-Time Process Model

We introduce a real-time process model that captures the semantics of the previously discussed real-time programming models. In Section 4 we use the process model to study the semantics of real-time programs developed in these models.

An *embedded process* $\mathcal{P}$ consists of program code and process variables X. The program code implements the *process actions* $A[\mathcal{P}]$ of the embedded process $\mathcal{P}$. Each process action $p \in A[\mathcal{P}]$ operates on a subset $X[p]$ of the process variables X. Process actions may share process variables but no control state: process actions compute or transport values of variables but do not invoke or control other process actions. Process actions are either *disabled*, or else *enabled* to be *scheduled* for execution. Process actions can be enabled at any time and then proceed according to any scheduling scheme. Enabling an already enabled process action leaves it enabled. Process actions complete voluntarily, they cannot be disabled. Thus the process activity of an embedded process evolves according to three orthogonal mechanisms: (1) a so-called *reactor* $\mathcal{R}$ that determines when a process action is enabled; (2) a *scheduler* $\mathcal{S}$ that determines which enabled process actions execute; and (3) the process actions themselves, which determine once enabled and scheduled when to complete and become disabled. Reactor and scheduler are part of an embedded process.

We define the semantics of embedded processes in terms of *process traces*: possibly infinite sequences $c_0, c_1, c_2, \ldots$ of process states c_i. The *process states* $C[\mathcal{P}]$ of an embedded process $\mathcal{P}$ consist of values for all process variables, an abstract *reactor* and *scheduler state*, and the *action states*: enabled or disabled. The sets of reactor and scheduler states are denoted by $R[\mathcal{P}]$ and $S[\mathcal{P}]$, respectively. A process trace starts with initial values for all process variables and initial action states in the process state c_0. The successor state c_{i+1} of a process state c_i is constructed in three steps: (1) the scheduler $\mathcal{S} \subseteq C[\mathcal{P}] \times A[\mathcal{P}] \times S[\mathcal{P}]$ chooses (non-deterministically) a process action p that is enabled in c_i to execute its next step and a new scheduler state s. We require that $\mathcal{S}$ always chooses well-defined actions and scheduler states. Let the intermediate process state c be equal to c_i except for the new scheduler state s; (2) the process action $p \subseteq X[p] \times X[p] \times \{enabled, disabled\}$ computes its next step, given the values of its action variables $X[p]$ in c_i, resulting in new values for $X[p]$ and its new action state, i.e., still enabled or now disabled. We require that p always computes well-defined values and action states. Let the intermediate process state c' be equal to c except for the new values of $X[p]$ and the new action state of p; and

(3) the reactor $\mathcal{R} : C[\mathcal{P}] \to 2^{A[\mathcal{P}]} \times R[\mathcal{P}]$ chooses, given the intermediate process state c', a subset E of all process actions to be enabled now and a new reactor state r. Let the successor state c_{i+1} be equal to c' except that the process actions in E are enabled in c_{i+1} in addition to the process actions that are already enabled in c and except for the new reactor state r. This completes the definition of the semantics of embedded processes. Note that the reactor is deterministic whereas the scheduler and the process actions, in particular, their duration can be non-deterministic. In order to model directly true concurrency in distributed systems, we could but, for simplicity, have not defined a scheduler $\mathcal{S}'$ to choose a subset of enabled process actions to execute simultaneously rather than a single enabled process action, cf., the definition of the reactor.

Process execution does not necessarily guarantee the *atomicity* of process actions: a process action p may execute in multiple steps interleaved with the execution of another process action that shares process variables with p. Guaranteeing atomicity is the problem of enabling process actions at the right time but not too rapidly and of scheduling process actions at the right time and not too late besides having process actions that complete fast enough. If we restrict the reactor to enable at most a single process action at the same time and not before any other process action completed, the scheduler disappears and we get atomicity and what valid implementations in the synchronous model do: a process trace is *event-safe* if in each state of the trace at most a single process action is enabled. An embedded process is *event-safe* if all its process traces are event-safe. Guaranteeing event safety requires the reactor not to enable process actions too rapidly while process actions must complete fast enough. This corresponds to proving reactivity and synchrony for implementations in the synchronous model. The following definition gives the reactor more freedom but requires a scheduler: a process trace is *time-safe* if a process action p never becomes enabled while another process action that shares process variables with p is enabled. An embedded process is *time-safe* if all its process traces are time-safe. Event safety implies time safety but not conversely. Implementations in the timed model require time safety. An equivalent definition of time safety, from the perspective of the scheduler and the process actions, is: a process trace is *time-safe* if each enabled process action p completes before another process action that shares process variables with p is enabled. In the following definition we put the burden onto the scheduler: a process trace is *space-safe* if in each state of the trace the scheduler never chooses an enabled process action to execute that shares process variables with another enabled process action that has previously been scheduled for execution but not yet completed. An embedded process is *space-safe* if all its process traces are space-safe. Time safety implies space safety but not conversely. Programmers in the scheduled model use synchronization mechanisms such as semaphores to ensure space safety.

Embedded processes can be composed to form an *embedded system* $\mathcal{E}$, which consists of the process actions of all processes, a reactor composed of the process reactors, and a scheduler composed of the process schedulers. The process actions of different processes may share process variables called *system variables*

of $\mathcal{E}$ but no control state. In an embedded system $\mathcal{E}$, we distinguish *reactions* and *coactions* of a process $\mathcal{P}$: a reaction is a process action that operates on process variables of $\mathcal{P}$, including system variables; a coaction is a process action that operates exclusively on process variables of $\mathcal{P}$ that are not system variables. Coactions are a restricted form of reactions that can be enabled and scheduled independently of the reactions of other processes without violating space-safety. An embedded system is semantically a single embedded process composed of multiple embedded processes but with a namespace for process actions and variables that is syntactically structured according to the original embedded processes. Suppose that an embedded system $\mathcal{E}$ is composed of two embedded processes $\mathcal{P}$ and $\mathcal{Q}$ where $\mathcal{P}$ models a physical process and $\mathcal{Q}$ models a software process. $\mathcal{P}$ consists of a coaction p that models the physical process as well as a reaction s for sensing p and a reaction a for actuating p. Thus s and a can be seen as device drivers that transport data between process variables of p (physical world) and system variables (system memory). Similarly, $\mathcal{Q}$ consists of a coaction t (software task) as well as a reaction i (task input) and a reaction o (task output). i and o transport data between process variables of t (task memory) and system variables. A typical execution order of the process actions is s, i, t, o, a, s, $\ldots$ where p logically may run concurrently with i, t, and o.

The *output behavior* of $\mathcal{Q}$ is *value-deterministic* if the system variables of $\mathcal{E}$ carry deterministic values at the instants when reactions of other processes than $\mathcal{Q}$, e.g., the reaction a, become enabled. $\mathcal{Q}$ must compute deterministic values based on the values of its process variables and the system variables at the instants when reactions of other processes than $\mathcal{Q}$, e.g., the reaction s, complete. Value-determinism means that $\mathcal{Q}$ computes, given the same sequence of inputs, the same sequence of outputs but not necessarily at the same time. The *timing behavior* of $\mathcal{Q}$ is *time-deterministic* if the reactions of $\mathcal{Q}$ are enabled and complete at deterministic instants independently of the instants when reactions of other processes than $\mathcal{Q}$, e.g., the reaction s or a, are enabled or complete. A weaker property than time-determinism is that the reactions of $\mathcal{Q}$ such as i and o always eventually complete and thus enable the interaction of $\mathcal{P}$ with $\mathcal{Q}$: from the perspective of $\mathcal{P}$, $\mathcal{Q}$ is *time-live* if the reactions of $\mathcal{Q}$ always eventually complete. We also say that $\mathcal{Q}$ is *bounded time-live* if there is an upper bound on the time any reaction of $\mathcal{Q}$ is enabled. The composition of embedded processes is *order-preserving* if the reactor of the composed system enables the process actions deterministically in the same order as the reactors of the individual processes. The composition is *time-invariant* if the reactor of the composed system also enables the process actions at the same instants as the reactors of the individual processes. An embedded system composed of embedded processes is value-deterministic (and time-deterministic) provided the composition is order-preserving (and time-invariant) and event- or time-safe, and the individual processes are value-deterministic (and time-deterministic). This is not true for space-safety since the composed scheduler may have to change the order in which process actions that share variables are scheduled in the composed system.

4 Implementations of Real-Time Programs

In this section, we show how real-time programs developed in the synchronous, scheduled, and timed model can be implemented as embedded processes. The following table gives an overview of the model and process properties:

	Synchronous	Scheduled	Timed
Model Composition	value-deterministic	–	value-, time-deterministic
Model to Process	event-safe	space-safe	time-safe
Process Composition	order-preserving	–	order-preserving, time-invariant

Synchronous Implementation. A *synchronous system* is the implementation of a synchronous program based on synchronous processes. A *synchronous process* reacts to the stimulus and input from other processes. For example, a synchronous process $\mathcal{Q}$ is typically triggered by another process, e.g., a physical process $\mathcal{P}$ that generates the input for $\mathcal{Q}$. As soon as the synchronous process is triggered, it takes the input and its internal state and computes some output and its next internal state. The output is returned to the physical process as soon as the synchronous computation is completed. Then the synchronous process waits for the next input. Synchronous processes can be modeled by *event-safe* embedded processes. Reactions model synchronous computation as well as input and output transfers from and to a synchronous process. Coactions are not required. Suppose that $\mathcal{Q}$ consists of a set R of reactions that implement $\mathcal{Q}$, and suppose that $\mathcal{P}$ consists of a reaction s to transfer input from $\mathcal{P}$ to $\mathcal{Q}$ and a reaction a to transfer output from $\mathcal{Q}$ to $\mathcal{P}$. The reactor enables s when $\mathcal{P}$ triggers $\mathcal{Q}$. As soon as s completes, the reactor enables, given the input and internal state of $\mathcal{Q}$, a reaction $r \in R$ to compute (part of) the output and next state of $\mathcal{Q}$. When r completes, the reactor either enables another reaction in R, or else enables a to return the output to $\mathcal{P}$. The reactor is typically implemented by some form of automaton. The job of the scheduler, on the other hand, is trivial since at any time in an event-safe execution of a synchronous process at most a single reaction is enabled. Non-trivial scheduling is only necessary if multiple synchronous processes are composed in parallel where some scheduling decisions are left open and not compiled into an automaton as, e.g., in *communicating reactive processes* with Esterel [2] or in the presence of *multiform time* with Lustre [13].

The design of a synchronous system requires a proof of event safety: before another input from $\mathcal{P}$ is available, the reactor must have stopped enabling reactions in R (reactivity) and all reactions must have completed (synchrony). Event safety implies time liveness if $\mathcal{P}$ always eventually generates new input for $\mathcal{Q}$. Thus event safety ensures that $\mathcal{P}$ can progress. $\mathcal{Q}$ is value-deterministic with respect to $\mathcal{P}$ in an event-safe embedded system $\mathcal{E}$ if the reactor and all reactions implement functions. The value-determinism of synchronous systems is important and supports the development of mission- and safety-critical software in the synchronous model. The synchronous model is compositional with respect to value-determinism because an event-safe system composed of value-deterministic synchronous processes is again value-deterministic (provided the composition of

the subsystem reactors is order-preserving). The composition of synchronous systems is an interesting research topic: if the subsystem reactors can be reused or even distributed onto multiple processors, modular and distributed synchronous programming as well as separate and incremental compilation may be possible.

Scheduled Implementation. A *scheduled system* is the implementation of a scheduled program based on scheduled processes. A *scheduled process* may run concurrently with other processes. Process communication is typically handled by semaphores that control the access to shared resources such as memory or I/O devices. A POSIX process is an example of a scheduled process. Program code that accesses a shared resource is called a *critical section* of a scheduled process. In order to preserve data consistency, a critical section must not be preempted by critical sections of other processes that access the same resource. Scheduled processes can be modeled by *space-safe* embedded processes. Coactions model non-critical program code whereas reactions correspond to critical sections. Thus taking or giving a semaphore requires two process actions and a transition from the process action that precedes the access to the semaphore to the process action that succeeds the access. The reactor handles the transitions by enabling succeeding process actions as soon as the preceding process actions complete. The scheduler can choose an enabled reaction r (critical section) for execution only if any other enabled reaction that shares system variables r has either completed or not yet started executing. The space-safe execution of the scheduled system is thus up to the scheduler, provided the original scheduled processes use the semaphores correctly.

The design of many scheduled systems is dominated by considerations on space safety through complex scheduling where neither the process actions nor the reactor have to worry about space safety. This is the strength and at the same time the weakness of the scheduled model. All the know-how and the tools for the development of complex but non-real-time systems is readily available in the scheduled model with deadlines because scheduled processes are not restricted in their control-flow and can be triggered to execute at any time. The scheduler, on the other hand, is responsible to guarantee space safety under as many circumstances as possible. This view has lead to an impressive amount of research in real-time scheduling, see, e.g., [3]. The downside of the scheduled model is that it is not compositional with respect to value- or time-determinism. In general, the composition of scheduled processes results in real-time behavior of the scheduled processes that is different from the real-time behavior of the processes when running individually. The problem is that the processes and the reactor can push the scheduler in situations such as *priority inversion* or *deadlock* that are hard or even impossible to handle. Traditionally, this problem has been addressed but not solved by making the scheduler smarter using so-called priority inheritance or priority ceiling protocols, see, e.g., [3].

An alternative approach is to shift the focus from space safety to time safety. Recall that the reactor in a time-safe embedded system never enables a process action p while another process action that shares variables with p is enabled. Thus the scheduler in a time-safe embedded system is only concerned with com-

pleting process actions before the reactor enables others but not with any particular ordering of the process actions: priority inversion and deadlock are not possible in a time-safe embedded system. Time safety requires the real-time programmer to keep in mind that the completion of a process action does not necessarily enable the next process action immediately. This means for a scheduled process that the process cannot simply *try to take* a semaphore but *has to accept* and *has to give up* a semaphore in a timely fashion. In particular, at the time when the process has to accept the semaphore the process must already be waiting for the semaphore: the process must be time-safe. We call a semaphore that must be accepted and released within some given time interval a *timed semaphore*. The semantics of a timed semaphore may be relaxed in applications where determinism is less important. For example, a scheduled process may be allowed to *anticipate* or even *reject* a timed semaphore. However, scheduled systems are not value- or time-deterministic even when using timed semaphores. Timed semaphores only prevent processes from delaying or blocking each other and thus avoid the problem of priority inversion and deadlock.

Timed Implementation. A *timed system* is the implementation of a timed program based on timed processes. A *timed process* consists of timed tasks that may run concurrently with other timed tasks or processes. A *timed task* is a sequential program with logically fixed execution time from invocation to completion. The invocation of timed tasks may be event- or time-triggered as well as task-triggered, i.e., triggered by the completion of a task. A Giotto task is an example of a timed process with a single timed task that is periodic and thus time-triggered. The logical execution time of the timed task is the period of the Giotto task. A timed task consists of a *task* for process computation and two *drivers* for process communication: an *input driver* that transports data from shared memory into task memory, the task that operates on task memory, and an *output driver* that transports data from task memory to shared memory. The execution of a timed task begins with the execution of the input driver followed by the execution of the task. The output driver executes after the task completes. The execution of a timed task ends with the completion of the output driver exactly at the time when the logical execution time of the timed task elapsed. Thus process communication with a timed task is only possible before and after but not during the execution of the timed task. Task memory may only be shared by multiple timed tasks of the same timed process. Since timed tasks have no means of synchronizing on task memory, timed tasks sharing task memory must be invoked and scheduled without preempting each other.

Timed processes can be modeled by *time-safe* embedded processes. Coactions model tasks whereas reactions correspond to drivers. The reaction that models the input driver of a timed task is enabled by the reactor as soon as the timed task is triggered. The coaction that models the task is enabled when the input driver completes. The reaction that models the output driver is enabled when the time is right to meet the specified completion time of the timed task. By then the task must have completed, i.e., the task must be time-safe. In a system with multiple timed processes, the reactor may enable multiple coactions (tasks)

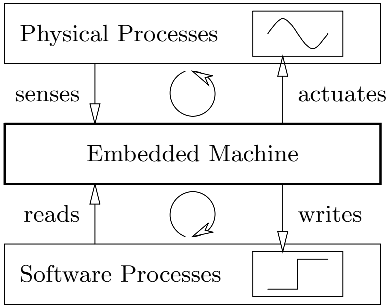

Fig. 4. The Embedded Machine

b: call(o)
call(a)
call(s)
call(i)
schedule(t)
future(g, b)

Fig. 5. An E Code Program

but only a single reaction (driver) at the same time. For example, if a timed task T_1 is invoked (completes) at the same time when a timed task T_2 of a different process is invoked (completes), the reactor enables the reactions that model the input (output) drivers of T_1 and T_2 sequentially in some order. The coactions that model the tasks of T_1 and T_2 may be enabled at the same time. For a time-safe execution the scheduler is assumed to schedule reactions with higher priority than coactions. Since at most a single reaction is enabled at the same time, scheduling reactions is trivial. A coaction, on the other hand, must be scheduled such that it completes before the reactor enables a reaction that shares memory with the coaction. Thus the reactor is responsible for the precise timing of inter-process communication while the scheduler takes care of process computation.

The reactor for timed processes can be implemented by a virtual machine called the *Embedded Machine* [7] (*E machine* for short). The E machine mediates in real-time the interaction of physical processes and software processes as shown in Figure 4. The E machine has already been used as a target machine for the compilation of Giotto programs [8] to *E code*, the code interpreted by the E machine. Besides some auxiliary instructions, e.g., for control-flow, the E machine has three key instructions: (1) the `call`(d) instruction enables a driver d and blocks the E machine until d completes; (2) the `schedule`(t) instruction enables a task t without blocking the E machine. t will not be scheduled for execution before the E machine is finished executing E code; and (3) the `future`(g, b) instruction makes the E machine execute the E code at address b when the *trigger* g becomes true. Triggers control the invocation of the E machine such as event-, time- or task-triggered invocations. For example, g may be a *time trigger* that becomes true when, say, 10ms elapsed after the execution of a `future` instruction with g. Then we say that the E code block is *enabled*. The E machine uses a *trigger queue* to maintain the list of triggers that currently *guard* an E code block. As soon as a trigger in the queue becomes true it is removed from the queue and the guarded E code block is executed. The E code blocks that are enabled at the same instant are executed in the order of the triggers in the queue. New triggers are appended to the queue. When all enabled E code blocks have

been executed, the E machine invokes a scheduler that takes care of scheduling the enabled tasks.

As an example of E code that requires only a singleton trigger queue, consider a timed process $\mathcal{Q}$ that invokes a timed task T periodically with a period of 10ms. The timed process $\mathcal{Q}$ is composed in parallel with a physical process $\mathcal{P}$. The E code program in Figure 5 implements the composition of $\mathcal{P}$ and $\mathcal{Q}$. The E machine begins executing the E code block at address b by calling the output driver o of the timed task T. Provided the task memory of T is initialized, o transports well-defined values from task memory to shared memory. The subsequent execution of the *actuator driver* a transports values from shared memory to the physical process $\mathcal{P}$. Thus o and a implement process communication from the timed to the physical process. The other direction is implemented by a *sensor driver* s of the physical process $\mathcal{P}$ and the input driver i of the timed task T. After calling the four drivers in the given order, the $\texttt{schedule}(t)$ instruction enables the task t of the timed task T for execution. The $\texttt{future}(g, b)$ instruction makes sure that the E machine wakes up after 10ms elapsed and executes the same E code block at address b again. In the programming model of the E machine, E code execution including driver execution is synchronous computation that takes no time. Since the actual execution of E code results in a delay of the driver execution, the above E code block implements the timed process $\mathcal{Q}$ correctly provided the E code execution is *time-safe*, i.e., task t always completes within 10ms minus the delay of the E code block. Then, from the perspective of the physical process $\mathcal{P}$, the behavior of the timed process $\mathcal{Q}$ is value-deterministic (provided the task and all drivers implement functions) and time-deterministic, up to the delay (jitter) caused by the output driver o but independent of the execution of task t. The E code can be further optimized such that the actuator and sensor drivers are executed independently of the E code for $\mathcal{Q}$. Optimizing E code for timed processes typically means more and smaller E code blocks with *more precise timing* rather than *faster execution*.

A sufficient criterion for value- and time-determinism with respect to the physical processes is to make sure that all drivers and tasks implement functions and all E code is event- or time-triggered. The composition of event- or time-triggered E code is time-invariant. Under these restrictions the timed model and thus Giotto is compositional with respect to value- and time-determinism. Notice that separate compilation of individual processes into E code is also possible. The following E code implements the same behavior as the E code of the previous example but can be generated separately for each process:

$b1$: $\texttt{call}(o)$	$b2$: $\texttt{call}(a)$	$b3$: $\texttt{call}(i)$	$b4$: $\texttt{future}(g, b2)$
$\texttt{future}(\mathsf{true}, b3)$	$\texttt{call}(s)$	$\texttt{schedule}(t)$	
	$\texttt{future}(\mathsf{true}, b4)$	$\texttt{future}(g, b1)$	

The E code blocks at $b1$ and $b3$ implement the timed process $\mathcal{Q}$. The E code blocks at $b2$ and $b4$ implement the interaction of the physical process $\mathcal{P}$ with $\mathcal{Q}$. Every 10ms the E code blocks will be executed in the order $b1$, $b2$, $b3$, and $b4$. The E machine starts executing with the initial trigger queue $\langle(0, b1), (0, b2)\rangle$ where,

e.g., $(0, b1)$ means that the E code block at $b1$ must be executed now. After the E machine is finished executing the E code block at $b1$, the output driver o has executed and the trigger queue is $\langle(0, b2),(0, b3)\rangle$. The $\texttt{future}(\texttt{true}, b3)$ instruction uses an already true trigger to make the E machine execute the E code block at $b3$ at the current instant but not before any other, already enabled E code blocks have been executed. The next step of the E machine is to execute the E code block at $b2$, which results in the trigger queue $\langle(0, b3),(0, b4)\rangle$. Then the E code block at $b3$ is executed, resulting in the trigger queue $\langle(0, b4),(10, b1)\rangle$ where $(10, b1)$ means that the E code block at $b1$ must be executed after 10ms from now have elapsed. The E machine finishes executing E code at the current instant with the trigger queue $\langle(10, b1),(10, b2)\rangle$ after executing the E code block at $b4$. Thus every 10ms the E machine will repeat the above behavior. Other timed and physical processes can now be added to the above system. For example, a second timed process could be compiled into E code, e.g., with a start address $b5$ and an entry $(0, b5)$ in the initial trigger queue. Similarly, partial Giotto programs can also be compiled separately into E code since Giotto programs are a special case of timed systems.

In general, the composition of timed and physical processes works in three steps: (1) *compilation* of each process into E code; (2) *linking* of E code; and (3) *validation* of E code with respect to time safety. In the previous example, E code is linked *online* through the trigger queue while executing the E code. Online linking of E code simplifies E code generation but suffers from runtime overhead for managing the trigger queue. On the other hand, E code could also be linked *offline*, which requires more complex E code generation with additional symbolic information but may result in linked E code that can be executed with less runtime overhead. Offline linking of E code does not necessarily mean that it has to be done at compile time. E code can be loaded and linked even dynamically while the E machine is running. Although E code is a static description of the timing behavior of real-time programs, linking E code has the potential to express dynamics that we know from traditional real-time operating systems while maintaining, e.g., time-determined system behavior. For example, changing the set of currently executing tasks in a real-time operating system corresponds to loading and linking appropriate E code in the E machine. Linking E code is therefore an interesting research topic. Validating E code with respect to time safety, in particular, in the context of dynamically changing E code is another interesting topic. The problem of checking time safety of embedded processes can be formulated as a three player game between the reactor, the scheduler, and the process actions. This approach has been demonstrated for the special case of the E machine as a two player game in which the E machine teams up with the *environment* (physical processes) against the scheduler [8].

5 Summary

The synchronous model supports the development of value-deterministic real-time programs for mission- and safety-critical applications. The scheduled model utilizes the experience and tools from the non-real-time world and is widely

used in practice. The timed model supports the development of value- and time-deterministic real-time programs but has only been applied in the context of embedded control systems. The synchronous and scheduled model are in some sense more general than the timed model. In both models value- and time-deterministic real-time programs can also be developed. The restrictions of the timed model, however, can be exploited, e.g., in optimized code generation of value- and time-deterministic real-time programs even for distributed hardware.

Acknowledgments. We thank Jörn Janneck and Marco Sanvido for many valuable comments and suggestions.

References

1. G. Berry. The foundations of Esterel. In C. Stirling G. Plotkin and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.
2. G. Berry, S. Ramesh, and R. K. Shyamasundar. Communicating reactive processes. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–98, 1993.
3. G. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer, 1997.
4. N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer, 1993.
5. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proc. of the IEEE*, 79(9), 1991.
6. T.A. Henzinger, B. Horowitz, and C.M. Kirsch. Giotto: A time-triggered language for embedded programming. In *Proc. First International Workshop on Embedded Software (EMSOFT)*, LNCS 2211, pages 166–184. Springer Verlag, 2001.
7. T.A. Henzinger and C.M. Kirsch. The Embedded Machine: predictable, portable real-time code. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 315–326, 2002.
8. T.A. Henzinger, C.M. Kirsch, R. Majumdar, and S. Matic. Time safety checking for embedded programs. In *Proc. Second International Workshop on Embedded Software (EMSOFT)*, LNCS. Springer Verlag, 2002.
9. C.M. Kirsch, M.A.A. Sanvido, T.A. Henzinger, and W. Pree. A Giotto-based helicopter control system. In *Proc. Second International Workshop on Embedded Software (EMSOFT)*, LNCS. Springer Verlag, 2002.
10. E.A. Lee. Embedded software. In M. Zelkowitz, editor, *Advances in Computers*, volume 56. Academic Press, 2002.
11. C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1), 1973.
12. S. Malik and Y.-T. Li. *Performance Analysis of Real-Time Embedded Software*. Kluwer, 1999.
13. D. Pilaud and N. Halbwachs. From a synchronous declarative language to a temporal logic dealing with multiform time. In *Proc. of the Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, LNCS 331, pages 99–110. Springer Verlag, 1988.
14. J. Sifakis. Modeling real-time systems – challenges and work directions. In *Proc. First International Workshop on Embedded Software (EMSOFT)*, LNCS 2211, pages 373–389. Springer Verlag, 2001.
15. N. Wirth. Toward a discipline of real-time programming. *Communications of the ACM*, 20(8):577–583, 1977.