From Control Loops to Real-Time Programs

Paul Caspi and Oded Maler

Verimag-CNRS
2, av. de Vignate
38610 Gires
France
www-verimag.imag.fr
Paul.Caspi@imag.fr
Oded.Maler@imag.fr

1 Introduction

This article discusses what we consider as one of the central aspects of embedded systems: the realization of control systems by software. Although computers are today the most popular medium for implementing controllers, we feel that the state of understanding of this topic is not satisfactory, mostly due to the fact that it is situated in the frontier between two different cultures and world views (*control* and *informatics*) which are not easy to reconcile. The purpose of this article is to clarify these issues and present them in a uniform and, hopefully, coherent manner.

The article is organized as follows. We start with a short high-level discussion of the two phenomena involved, *control* and *computation*. In Section 2 we explain the basic issues related to the realization of controllers by software using a simple proportional-integral-derivative (PID) controller as an example. In Section 3 we move to more complex multi-periodic control loops and describe various approaches for scheduling them on a sequential computer. Section 4 is devoted to discrete event (and hybrid) systems and their software implementation. Finally, in Section 5 we briefly discuss distributed control and fault tolerance.

1.1 Control

A controller is a mechanism that interacts with part of the world (the "plant") by measuring certain variables and exerting some influence in order to steer it toward desirable states. The rule that determines what the controller does as a function of what it observes (and of its own state) is called the feedback function. In early days of control, the feedback function was "computed" mechanically: for example, in the famous Watt governor, analyzed mathemati-

cally by Maxwell, the angle of the governor was determined by the angular velocity by purely mechanical means.

With the advent of electronics, the process of computing that function was decoupled from measurement and actuation. Physical magnitudes of different natures were transformed into low-power electric signals. These signals were fed into an analog computer whose output signals were converted into physical quantities and fed back to the plant. From a mathematical standpoint, this architecture posed no conceptual problems. The underlying model of the plant and of the analog computer were of the same nature. The former was a continuous dynamical system with evolution defined by the differential equations of the corresponding physical theory (mechanics, thermodynamics, etc.), and the latter consisted of an electrical circuit with dynamics governed by similar types of laws. Schematically, we can define the evolution of the plant by the equation $\dot{x} = f(x, d, u)$ with x being the state of the plant, d some external disturbance and u the control signal. The dynamics of the controller implemented by an analog circuit can be likewise written as $\dot{u} = g(u, x, x_0)$, with x_0 being a reference signal, and the evolution of the controlled plant is obtained by the composition of these two equations. This is the conceptual framework underlying classical control theory, where the feedback function is "computed" continuously at each and every time instant.

The introduction of digital computers changed this picture completely. To start with, the computation of a function by digital means is an inherently discrete process. Numbers are represented by binary encoding rather than by physical magnitudes. Consequently, sensor readings should be transformed from analog to digital representation before the computation; conversely, the results of the computation should be transformed back from digital to analog form. The computation is done by a sequence of discrete steps that take time, and the electrical values on different wires are meaningless until the computation terminates. Thus it makes no sense to connect the computer to the plant in a continuous manner. The transition from physical to digital control is illustrated in Fig. 1.

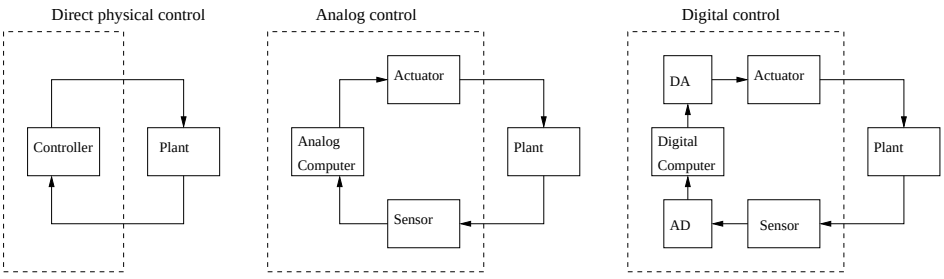

Fig. 1. From physical to analog to digital control

To cope with these changes a comprehensive theory of digital "sampled" control has been developed [2]. Within this theory, the interaction of the controller and the plant is restricted to sampling points, a (typically periodic) discrete subset of the real-time axis. At these points sensors are read, and the values are digitized and handed over to the computer, which computes the value of the feedback function, which is converted to analog and fed back to the plant via the actuators. From the control point of view, the sampling rate is determined by the dynamics of the plant, with the obvious intuition that a faster and more complex dynamics requires more frequent sampling. The sampling period is determined by the desired level of approximation and by the properties of the signal.

The role of the computer in this collaboration is to be able to compute the value of the function, (including the analog-to-digital (A/D) and digital-to-analog (D/A) conversions) fast enough, that is, between two sampling points. Once this is guaranteed, the control engineer can regard the computer as yet another (discrete-time) block in the system and ignore its "computerhood". This is certainly true for simple single-input-single-output (SISO) systems, but, becomes less and less so when the structure of the control loops becomes more complex. Before discussing these issues, let us take a look at computation.

1.2 Computation

In the early days of digital computers, their interaction with the outside world was rather limited. A typical batch program for producing a payroll or for performing an intensive numerical computation did not interact with the external world during execution. Such systems, termed "transformational" systems by Harel and Pnueli [12], read their input at the beginning, embark on the computation process and output the result upon termination. The fundamental theories of computability and complexity are tailored to this type of "autistic" computation. They can say which types of function from input to output can be computed at all, and for those that can, how the number of computation steps grows asymptotically with the size of the problem.

If we insist on philosophical rigor, we must admit that even computations of this type are "embedded" in some sort of a larger process. The batch numerical computation could have been, for example, a finite element algorithm to determine the stability of a building. Such a computation is part of the construction process and should be invoked each time a new building is designed or when a change is initiated by the architect. The computation time of such a program, even in the early days when it was measured by hours and days, was still reasonable with respect to the time scale of a typical construction project. Likewise, a payroll program is part of the "control loop" of an organization which reads the time sheet of the employees and prints checks at the end of the month. If the execution time of such a program were on the order of magnitude of weeks, it could not fulfill its role in that control loop.

So the difference with respect to the progressively more interactive computations, that will be described in the sequel, is also a quantitative matter of time scales.

With the development of time-sharing operating systems, the nature of computation became more interactive. A typical example would be a text editor, a command shell or any other program interacting with one or more users via keyboards and screens.[1] What is the function that such an interactive program "computes"? People familiar with automata theory can see that it is a *sequential function*, something that transforms sequences of input symbols (commands) to sequences of output symbols (responses). The important point in such functions is that the process of computation is no longer isolated from the input/output process but is rather interleaved with it: the user types a command, the computer computes a response (and possibly changes its internal state) and so on. These are called "reactive" systems in [12].

While such interactive systems differ considerably from batch programs that operate within a static environment which does not change during computation, they still operate under certain restricting assumptions concerning their environment, which is typically a human user or a computer program that follows some protocol. The implicit assumption is that the environment behaves in a manner similar to a player in a turn-based game like chess; that is, the user waits for the response of the computer before entering the next input. As in the case of batch systems, this metaphor is valid as long as the the computer is not slower than the external environment against which it works. When a person's typing speed exceeds the reaction speed of the text editor, or when a transmitter transmits faster than a receiver receives, everything breaks down.

Digital implementations of continuous control systems, the subject of this chapter, interact with the physical world, a player which is assumed to be governed by differential equations, and which evolves independently of whether the computer is ready to interact with it. Of course, in the same way as a text editor may ignore characters that are typed too fast, a slow computer may ignore sensor readings or not update actuator values fast enough. However, in many "time-critical" systems, the ability of the computer to meet the rhythm of the environment is *the key to the usefulness of the system*. Failing to do so may lead in some cases to catastrophic results, and in others, to severe degradation in performance. Such systems are often called real-time systems to distinguish them from the types of programs previously described and to indicate the tight coupling between the internal time inside the computer and the time of the external world.[2] In the next section we discuss various differences between such programs and the control loops that they realize.

[1] Today it is hard to imagine how computing could be otherwise, but the passage from batch to terminal-based computation was revolutionary at the time, and the authors are old enough to remember that.

[2] Sometimes the terms online versus offline are used for similar purposes.

2 From Mathematical Descriptions to Programs

Programs implementing control systems differ from their corresponding discrete-time recurrence equations in several aspects, the first of which is not particular to control systems but is concerned with different levels of abstraction in which algorithms can be described. For instance, algorithms for searching directed graphs may be defined in terms of the abstract structure of the graph, the mathematical $G = (V, E)$, without paying attention to the way the graph is stored in memory. An abstract algorithm may contain statements such as "for each successor of a vertex v do" without being explicit about the way the successors of a node are retrieved from the data structure representing the graph. More concrete programs, written in languages such as C, need to specify these details. Between these levels and the actual physical realization there are many intermediate levels (assembly and machine code, micro code, architecture, etc.) and one of the great achievements of computer science and engineering is that most of the transformations between these levels are done automatically using computer programs.

As an illustrative example we consider one of the most popular forms of control, the PID controller, and see how it is transformed into a program. An important feature of feedback functions is that they are typically dynamical systems by themselves, admitting a *state* which influences their output and future behavior. Fig. 2 shows the Simulink diagram of a typical sampled-data PID controller. The annotation of the Simulink blocks is written in the z-transform formalism, which is a discrete version of a frequency-domain representation of systems, where delay and memory are expressed using the $1/z$ operator. An explanation of this formalism can be found elsewhere in the handbook, and we focus here on a more "mechanical" state-space description of the controller. What a PID controller essentially does is to take the input signal I, compute its derivative D and integral S and then compute the output O as some linear combination of I, S and D. The state variables of the system include the integral S and the previous value of the input J, which is needed for computing the derivative. The following system of recurrence equations defines the semantics of the controller as a set O_n of output sequences whose relation with the input sequence I_n is defined by

$$\begin{aligned} S_{-1} &= I_{-1} = 0.0 \\ S_n &= S_{n-1} + 0.1 \cdot I_n \\ O_n &= 5.8 \cdot I_n + 4 \cdot S_n + 3.8 \cdot 10.0 \cdot (I_n - I_{n-1}) \end{aligned} \tag{1}$$

The first line defines the initial values of state variable S and the second line defines its subsequent value for every $n \geq 0$. The last line determines the output, using $I_n - I_{n-1}$ as the derivative. Since old values of the input are not typically kept in memory, we will need to store this information in an auxiliary state variable J satisfying $J_n = I_n$, and replacing I_{n-1} in the definition of O_n by J_{n-1}.

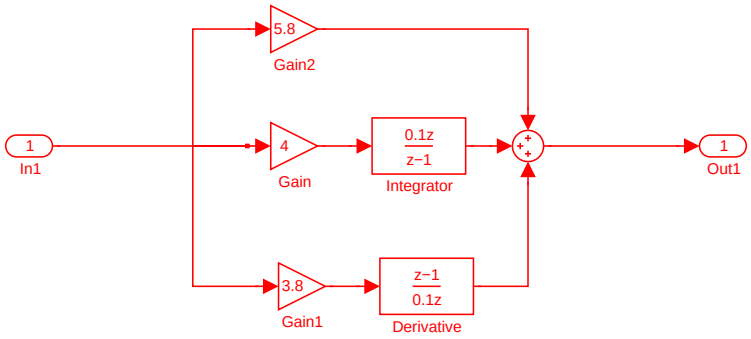

Fig. 2. A PID controller represented by a Simulink block diagram

Before showing the corresponding program, let us note that since (1) involves memory that has to be maintained and propagated between *successive invocations* of the program, the corresponding programming construct is better viewed as a *class* in an object-oriented language such as C^{++} or Java. However, since this point of view is probably not so familiar to most readers, we will realize it as a C program with *global variables*. These variables continue to exist between successive invocations of the program (like latches in sequential digital circuits when the clock signal is low). The program shown in Table 1 is a result of a rather straightforward transformation of (1).

```
/* memories */
float S = 0.0, J = 0.0;

void dispid_cycle (){
  float I,O;
  float J_1,S_1;

  I = Input();

  J_1 = I;
  S_1 = S + 0.1 * I * 4.0;
  O = I * 5.8 + S_1 + 10.0 * 3.8 * (I-J);
  J = J_1;
  S = S_1;

  Output(O);
}
```

Table 1. A program realizing a PID controller

The first part of the program is the declaration and initialization of the global variables `J` and `S`. The second part, the `dispid_cycle` procedure, describes the computation to be performed at each invocation of the program. It uses auxiliary variables `J_1` and `S_1` into which the new state is computed. The procedure presupposes two auxiliary functions `Input` and `Output` provided by the execution platform, which take care of bringing (digitized) sensor inputs into `I` and writing `O` onto the actuators. The implementation details of these functions are outside the scope of this article. The computational part of the procedure consists of taking the input and propagating it through a network of computations to produce the output. We first compute the next values of the state variables, then compute the output, write the new state values into the global variables and finally write the output and exit.

Upon closer inspection one can see that we do not really need the auxiliary variable `S_1` because only the *new* value of `S` is used while computing `O`. Consequently, we can replace the computation of `S_1` by direct computation of `S`, use `S` in the computation of `O` and discard the assignment statement `S = S_1`. In fact, we can do similar things with `J`, by putting the statement `J=I` after the computation of the output, to obtain the optimized program in Table 2.

```
/* memories */
float S = 0.0, J = 0.0;

void dispid_cycle (){
  float I,O;

  I = Input();

  S = S +0.1 * I * 4.0;
  O = I * 5.8 + S + 10.0 * 3.8 * (I-J);
  J = I;

  Output(O);
}
```

Table 2. An optimized program for the PID controller

Saving two variables and two assignment statements is not much, but for complex control systems that should run on cheap micro controllers, the accumulated effect of such savings can be significant.

The reader can easily appreciate that the process of writing, modifying and optimizing such programs manually is error prone and that it would be much safer to derive it automatically from the high-level Simulink model. We have derived a program similar to the program in Table 2 from the Simulink model of Figure 2 in two steps. First, the Simulink-to-Lustre translator [6] was used to transform the model into a program in Lustre, a language [11] which provides

rigorous syntax and semantics for expressing data-flow equations such as (1). Then the Reluc Lustre-to-C code generator [9] produced the program after automatic analysis of state variables, dependencies and other optimizations.

The story does not end with the generation of machine code by the C compiler, as there are some additional conditions associated with the execution platform that need to be met. To begin with, the platform should support the I/O functions and be properly connected to all the machinery for conversion between digital and analog data. Second, the proper functioning of the program depends crucially on its being invoked every T time units, where T is the sampling period of the discrete-time system according to which the parameters of the PID controller were derived. Not adhering to this sampling period may result in a strong deviation of the program behavior from the intended one. This is a very particular (and rather unexpected) class of software errors inherent in control applications.

To ensure the correct periodic activation of the program we need access to a *real-time clock* that will trigger the execution every T time units. But this is not enough due to yet another important difference between an abstract mathematical function and a program that computes it: the former is *timeless* while the latter takes some time to compute. For a program such as `dispid_cycle` to function, the condition $C < T$ should hold, where C is its worst-case execution time (WCET). If this requirement is not met, the program will not terminate before its next invocation (see the timing diagram in Fig. 3). Measuring and estimating the WCET of a program on a given architecture is not an easy task, especially for modern processors, and it is subject to extensive ongoing research [25].

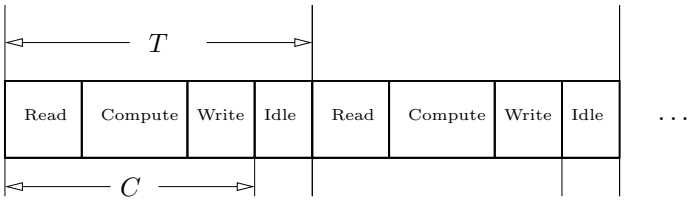

Fig. 3. The execution of a control program with a period T

Once these conditions are fulfilled, several implementation techniques can be used. Historically, such controllers were first implemented on a bare machine, without using any operating system (OS). The real-time clock acts as an interrupt that transfers control to the program. If the scheduling condition $C < T$ is satisfied, this interrupt occurs after the program has terminated and the computer is idle. Hence, unlike preemptive scheduling, there is no need for context switching and complex OS services. This implementation technique is thus both simple and safe and does not need to rely on a complex piece

of software like an OS, which is difficult to validate. Much progress has been made in real-time OS (RTOS) technology, and today commercial systems are available that have been exercised and debugged by a large number of users and can be considered quite safe. Hence the role of monitoring the real-time clock and dispatching the program for execution can be delegated to an OS.

This concludes the discussion on the implementation of simple control programs where we have tried to touch upon the key relevant computational aspects. In the next section we focus on the timing-related aspects of implementing more complex control loops.

3 Complex Periodic Controllers

In many control applications, systems have several degrees of freedom that must be controlled *simultaneously*. Mathematically each controller c_i is just another recurrence equation that coexists with the other equations. Computationally, these loops should be realized on a *sequential* computer that can do one thing at a time. The problem of how to "sequentialize" and schedule these parallel processes is one of the major topics in real-time systems. It is important that each invocation of a controller will have its relevant inputs ready before it starts executing and that the computation of all its outputs and their transmission to the outside world terminate in due time. This is the basic functional requirement from real-time control software, a fact sometimes obscured by details of operating systems and scheduling policies.

3.1 Single period

We start with the simple case where all controllers share the same sampling period T. This means that all of them should be invoked at every cycle of the system. A necessary condition for realizing these controllers sequentially on a given architecture is that all the computations (including input and output) should fit inside the cycle or, in other words, the condition $\sum C_i < T$ is satisfied where each C_i is the WCET of controller c_i on that architecture.

In this setting, the code of each controller can be generated separately as described in Section 2. The sequential implementation of the whole control program can be achieved by a simple scheduler that invokes the controllers one after the other. However, a somewhat less modular but more efficient method consists of gathering all the controllers into a single program and using an optimizing compiler to generate the code of the global controller. By analyzing the structure of the controllers and their data dependencies, a smart code generator can find out that some parts of the computation are shared by several controllers and need not be computed more than once. Such optimizations may reduce the number of operations and a slower computer can be used to achieve the required sampling rate. With the progress of these code generators, this technique is becoming more popular. Verifying the correctness of such optimizing compilers is, by itself, an active research topic.

3.2 Multiple periods

When a system has to control several variables, it is often the case that the variables follow dynamics of different speeds and need to be controlled at different sampling rates. The specification of such a multi-rate system can be given by a collection of pairs $\{(c_i, T_i)\}_{i=1..n}$ where T_i is the period of controller c_i, which can be considered as an integer. With such a task specification we associate two numbers, the basic period $T = gcd(T_1, \ldots, T_n)$, and the super-period $P = lcm(T_1, \ldots, T_n)$, where gcd and lcm are, respectively, the greatest common divisor and the least common multiple of the task periods. As a running example we consider the 3 task system $S_{123} = \{(c_1, 1), (c_2, 2), (c_3, 3)\}$ with $T = 1$ and $P = 6$, depicted graphically in Fig. 4. The implementation of this abstract specification consists of allocating portions of the timeline of the processor to instances of the controllers (tasks) so that their execution times satisfy the implied constraints. Due to periodicity, if a schedule is found for the first P cycles, it can be repeated indefinitely for the rest of the timeline.

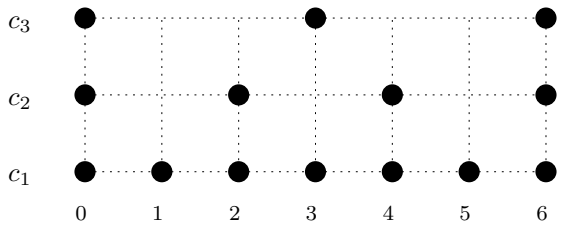

Fig. 4. A multi-rate specification $S_{123} = \{(c_1, 1), (c_2, 2), (c_3, 3)\}$

Cyclic Executive

The most straightforward solution is to execute c_1 every cycle, c_2 every second cycle and c_3 every third cycle (see Fig. 5). While this solution is simple and natural, it is not very efficient in utilizing the computer time. As we can see, there are very "busy" cycles where all three controllers need to be executed, while in others the computer is mostly idle. Using this approach, it is the most busy cycle which determines the relation between platform speed and feasibility of the schedule. In this example the schedule is feasible only on platforms satisfying $C_1 + C_2 + C_3 < T$.[3]

More efficient solution schemes are based on the assumption that the nth instance of task c_i can be executed anywhere in the interval $[(n-1) \cdot T_i, n \cdot T_i]$.

[3] Improvements can sometimes be achieved by using different phases (offsets) for the periodic computations.

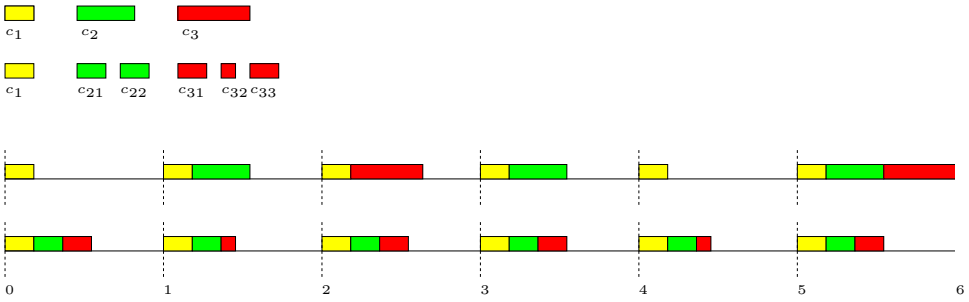

Fig. 5. Schedules for example S_{123}: a simplistic imbalanced schedule versus static partitioning

The lower and upper bounds of the interval are often called, respectively, the *release time* and *deadline* of the task. The set of all such intervals for our example is depicted below:

$$\begin{array}{ll} c_1\text{:} & [0,1],[1,2],[2,3],[3,4],[4,5],[5,6] \\ c_2\text{:} & [0,2],[2,4],[4,6] \\ c_3\text{:} & [0,3],[3,6] \end{array}$$

Instead of restricting the execution of the slow controllers c_2 and c_3 to the cycle where they need to produce their outputs, we can execute parts of them in earlier cycles when the processor is available. Technically there are different methods for splitting the execution of the slow tasks to obtain a more balanced distribution of the computational effort.

Offline splitting

One approach consists in partitioning the code of every slow controller *offline* into pieces of approximately equal execution times and distributing their execution over all cycles inside its period. In our example this means splitting c_2 into c_{21} and c_{22}, splitting c_3 into c_{31}, c_{32} and c_{33} and using a cyclic executive to schedule the modified tasks, leading to a schedule like the one illustrated in Fig. 5. The corresponding schedulability condition becomes:

$$\max_j \sum_i C_{ij} < T.$$

This solution, which has many advantages, is quite popular in practice. For instance, it is the one adopted in the time-triggered architecture (TTA) framework [15], where it is handled by several commercial tools. One disadvantage of this approach is that the splitting of a control loop into subparts of similar execution time is not easy to accomplish at the application level (Simulink

model) and possibly requires several iterations until a feasible schedule is found. Doing it directly on the code of the control program one loses some of the methodological advantages of automatic code generation. The variability in the execution times of the same program on modern processors does not make this job easier.

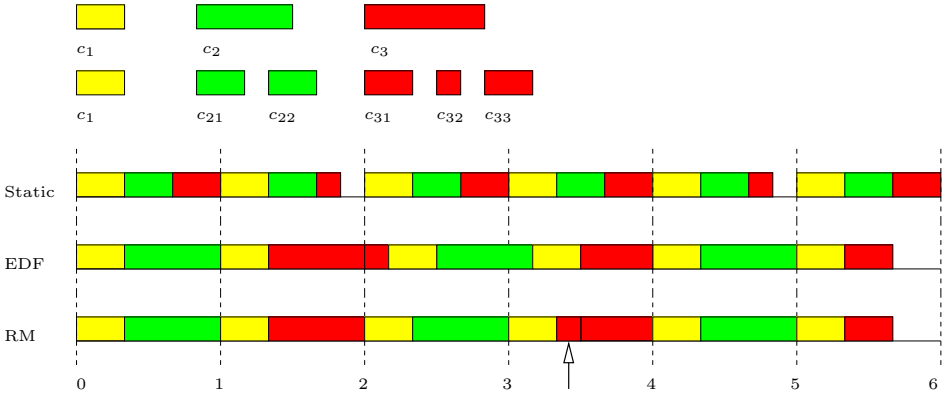

Fig. 6. Schedules for the S_{123} example: static splitting, EDF and RM

Preemptive solutions

The other class of solutions is more dynamic and is based on the preemption services of an RTOS. Every controller is compiled into a simple program, each instance of which is viewed as an independent task dispatched for execution by a scheduler according to some policy. The basic principle is that a slow process may execute when the computer is available, but when a more urgent task is released, the active computation is stopped and resumes when the urgent task terminates. This "context switching" (saving the contents of registers) takes some time, which we ignore in this discussion. The classical result of Liu and Layland [18] shows that, for preemptive scheduling, a set of tasks is schedulable if the amount of computation time to be consumed in P cycles is smaller than $P \cdot T$, that is,

$$\sum_i C_i/T_i < 1.$$

The two most popular scheduling policies are *earliest deadline first* (EDF) and *rate-monotonic* (RM).

Earliest deadline first: The simplest and most natural way to allocate the time budget of the processor is to prefer most urgent tasks: at any moment, choose among the enabled tasks the one with the nearest deadline. If two or more tasks have the same deadline, an arbitrary choice can be made, with preference to tasks that are already executing (to minimize context switching). An example of an EDF schedule obtained for S_{123} appears in Fig. 6. Note that when the third instance of c_1 arrives, it does not preempt the first instance of c_3, because they have the same deadline. EDF was introduced in [18] and has been proven to be optimal.

Rate-monotonic: The alternative and rather popular approach is to use a *static* priority relation among tasks based on their frequency ($c_1 \prec c_2 \prec c_3$ in our case). Then at every time instant the task with the highest priority among the enabled ones is selected for execution. RM schedules tend to make many more preemptions than EDF and, even if we ignore context switching, they are provably less efficient than EDF schedules. As one can see in Fig. 6, S_{123} is not schedulable by RM on the same platform for which it is schedulable by EDF as the computation of the first instance of c_3 misses its deadline. The popularity of RM can be partly explained by the fact that fixed priority policies are easier to implement in existing operating systems, and that the degradation in performance with respect to EDF is only 1/3 in the worst case.

3.3 Semantic issues

The discussion in the previous section was based on a simplified abstract view of the controllers, assuming their I/O to be atomic operations that take place within zero time at the endpoints of each of their respective periods. We also implicitly assumed that the controllers are independent and do not communicate. In reality, the I/O operations are often part of the code of each task, and the timing of their execution may depend on the scheduling policy used. We mention two issues related to this fact: data consistency and determinism.

Data consistency

The first low-level problem to be resolved is due to the possibility that preemption occurs in the middle of an I/O operation, leading to corrupted data. For example, a task may be interrupted after having read some part of a long piece of data and resume operation only after some other task has modified it. Several solutions exist for this problem:

1. Protection by semaphores: This technique, used extensively in operating systems when resources are shared by several tasks, consists of preventing the interruption from occurring during I/O operations. From the point of view of priority-based scheduling this means that the task increases its priority when it enters its "critical section". This feature makes the scheduling problem more complex because the blocking time has to be evaluated

and added to the WCET of the corresponding tasks. This can raise the well-known *priority inversion* problem for which solutions such as the *priority inheritance* or *priority ceiling* protocols have been invented [24].

2. *Lock-free* methods: Here the reading task may detect the fact that the data it has been reading has changed and it may restart reading, attempting to get uncorrupted data [17, 16, 1]. Although the number of times this may happen is finite, the time that can be spent on retrying should be accounted for in the schedulability analysis.
3. *Wait-free* methods: Here data that are shared by several tasks are duplicated (double- or triple-buffers) so that the reader and the writer use different "lock-free" copies and than toggle between them. Consequently, the schedulability analysis need not be modified, but more space is needed to store the shared data [8, 14].

Determinism

Under this title we group all phenomena related to the deviation of the implementation from the "nominal" control loop that may result from the potential variability in execution times of different instances of the same task. We illustrate this class of problems and compare the influence of such variability on the three types of scheduling policies previously mentioned (simple, static splitting and preemptive). No attempt is made to cover the whole panorama of considerations and practical solutions.

Consider example S_{123} where controller c_1 has a state variable y_1 which is computed every iteration as $y_1' = f(y_1, y_2, y_3)$ where y_2 and y_3 are computed by c_2 and c_3, respectively (note that this also covers the special case where y_2 and y_3 are just inputs sampled at a lower frequency). Before continuing, it is worth contemplating the definition of the computed controller in terms of the *external time* of the controlled environment. If we were dealing with continuous time or with uniform sampling, the values of y_1, y_2 and y_3 used in every invocation of c_1 would be of the same real-time "age", that is, something of the form

$$y_1(t') = f(y_1(t), y_2(t), y_3(t)). \tag{2}$$

Since the y's are computed/sampled at different rates, each invocation of c_1 inside the super-period can use only the most recent values of y_2 and y_3 that are available, which leads to six different variations on (2), one for each cycle (see Fig. 7). For example, in the last cycle we compute $y_1(t) = f(y_1(t-1), y_2(t-2), y_3(t-3))$.

This "non-uniform" relation, expressed naturally using the under-sampling features of Simulink, is the starting point of multi-periodic control loops.[4] Under the simple scheduling policy, this relation is robust under variations

[4] In fact, the exact definition of this relation may vary according to the details of the I/O mechanism, but the important point is that the same pattern repeats every P cycles.

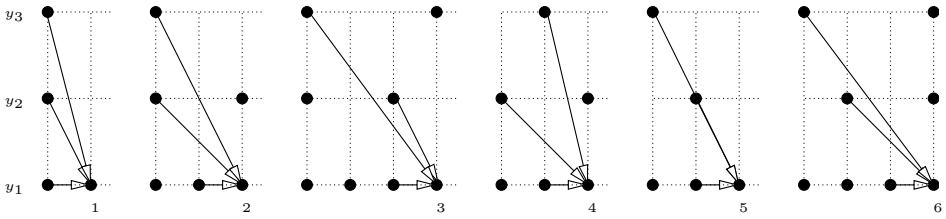

Fig. 7. Six different computations of $y_1' = f(y_1, y_2, y_3)$, each with a different external time characterization of the relation between the variables

in execution time because each task is executed in a predefined cycle. The situation is not much different if we use the static splitting approach, because the I/O operations appear in fixed portions of the code of each task, which are executed at predefined cycles.

On the other hand, preemptive methods are less robust in this sense as the I/O operations of a given instance of a task may occur at different cycles in different instances of the super-period depending on the point in the program where preemption takes place. For example, in the EDF schedule of Fig. 6, if c_3 takes less time and terminates within the second cycle, then the third invocation of c_1 may use this value, i.e., $y_3(t-1)$, instead of $y_3(t-3)$. A similar type of non-determinism, also known as *jitter*, is associated with the variation in the timing of the output operations. These types of non-determinism constitute one of the main criticisms of preemptive solutions for control applications. To alleviate this problem, various "time-triggered" solutions for the communication between different parts of the controller and for I/O in general have been proposed. Among them are the time-triggered architecture [15], to be discussed in Section 5, and the Giotto language [13] which allows preemption but isolates the execution of I/O operations from the rest of the code and forces them to take place in predefined time slots.

Let us remark that the attempts to maintain this determinism seem somewhat questionable, at least for periodic implementation of *continuous* control. The fact that the age of the value used by a controller deviates by a cycle or two between invocations need not have a significant effect on the performance of the control loop, given that such age variability already exists between consecutive cycles. Moreover, due to the measurements process and the variability of the external environment, there is not much sense in speaking of determinism in the actual execution of the control loop, although determinism is a convenient feature for debugging and simulation purposes. The situation may be different for a hybrid system where continuous and discrete event control are combined (see Section 4.3).

4 Discrete Events and Hybrid Systems

So far we have focused on classical continuous control, whose implementation by computers is supported by the mature theories of sampled-data control and periodic scheduling. In this section we address the implementation of *discrete event control* systems, which constitute an important ingredient of any modern control system and whose interaction with continuous control led to the emergence of a new field of research known as *hybrid systems*. Although such systems have been intensively studied in recent years, there is no comprehensive theory concerning their implementation, despite some recent efforts [10, 5].

4.1 Comparison with continuous control

The specification of a discrete event controller is given in terms of a *transition system*, a generic term which covers automata, Petri nets or variants of Statecharts (state machines augmented with features such as parallelism and hierarchy). A transition system is defined over a discrete set of states and discrete sets of input and output events (alphabets). The dynamics is given in terms of a transition function consisting of tuples of the form (q, a, b, q') with the following intended meaning: when an input event a occurs while in state q, an output event b is generated and the controller moves to state q' (see Fig. 8). Note that the execution of the transition is not merely a table lookup operation as in textbook finite-state automata, but may involve manipulation of complex data structures which are part of the state of the system. The software implementation of a transition system is a program that decides according to the current state and the input event which reaction to compute.

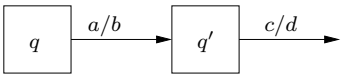

Fig. 8. A transition system

Although discrete event systems are defined using the same abstract scheme of dynamic systems, that is, read input, update state and write output, their nature is quite different from that of continuous systems (see a more detailed discussion in [19]). In the latter, the dependence of the dynamics on the values of the state and the input is more or less continuous as these are variables appearing in the numerical recurrence equation. In discrete systems, the dynamics is defined by if-then-else statements where the values of state and input variables serve to choose among the branches of the program.

This leads to a much larger variability in the execution time for subsequent invocations of the controller.

The second major difference is associated with the time axis with respect to which the system is defined. The specification of continuous control systems is tightly and explicitly embedded in the real-time axis through the sampling rates which determine when inputs are read and what the deadline is for each invocation of a controller. Discrete transition systems are typically defined with no reference to real time and operate on a *logical time scale*, defined by the events. In other words, the model says that after input a there will be an output b, but any amount of time may separate the two events. The only implicit constraint is that the transition should be completed before the arrival of the next input event.

Without constraints on the environment, only an infinitely fast controller that reacts in zero time can respond to any event at any time. Assuming the existence of such a fast machine is often called the *synchrony hypothesis*, and it is advocated, among others, by the proponents of the Esterel language [4]. Although such machines do not exist, it is claimed that this *zero time* approximation provides reactive programming languages with a much cleaner and simpler semantics. As benefits, programs are easier to understand, debug and validate. Let us also note that this assumption is implicitly accepted during simulation, for example, with tools such as Simulink/Stateflow: each time the controller has an action to perform, the simulation time is frozen, and resumes only after the action is completed. Of course, stopping "real" time is much more difficult. We mention a recent variation on the synchrony hypothesis proposed in [21] where zero is replaced by a fixed and uniform delay (the *logical execution time*) in the semantics of the specification. The choice of this number, however, requires looking into the properties of the execution platform, except, perhaps, for systems where the reactions are very simple.

When moving to software implementations of such systems, we must bring real metric time into the picture, both at the specification level (refining the response time requirements, adding assumptions concerning the speed of the environment) and at the implementation level (execution times of the reactions on a given platform, event detection mechanisms). As no system can detect and handle events that arrive with an unbounded frequency, we need to convert the ideal "untimed" specification into a realistic one by adding constraints to the model of the environment so that such "Zeno behaviors" are prevented.

A simple and natural way to restrict the environment is to assume a positive lower bound on the inter-arrival time of events (events that violate this constraint are ignored by the controller). This is a very sensible requirement when the events are determined by changes in the values of discrete signals. An implementation of a system admitting such a lower bound d is guaranteed to meet the specifications if the WCET of each transition is smaller than d. Sometimes it is reasonable to assume such a lower bound for each type of

input event separately. This does not prevent an event of one type from arriving while the (sequential) implementation is busy reacting to another event. However, if the respective WCETs are small enough, the system can cope with these events using a *bounded buffer* that stores pending events (this is similar to the schedulability of multi-period systems discussed in Section 3).

Alternatively, one can explicitly set deadlines for each reaction or simply assign priorities so that the system will respond to the more important events and postpone the treatment of others while it is overloaded (this approach is common in "soft" real-time systems). As we have already noted, the determination of the real-time requirements is less systematic than in the case of continuous systems, and in many cases this part of the specification will be derived a posteriori from the constraints of the execution platform rather than in a top-down fashion.[5]

4.2 Implementation strategies

Let us illustrate two popular implementation styles without attempting to be exhaustive in the coverage of all existing approaches.

Single program time-triggered implementation

This is probably the most popular implementation strategy. It attempts to treat discrete event systems using the same principles used for continuous ones. It is similar to the cyclic executive for multi-rate systems with which it can be easily combined, although no deep theory has been developed for it. We assume without loss of generality that events correspond to changes in values of Boolean signals. The set of controllers that specify the system is compiled into a single program, a sampling rate is chosen and it determines the deadline for the reactions to events. The input signals are sampled at a fixed rate and if a signal value is found to be different than in the previous sampling point, an event is declared. The reactions to all detected events are then executed sequentially and should terminate within the sampling period.

To see how this approach integrates easily with continuous control, consider, e.g., a train controller which must maintain a reference velocity using standard continuous control but which should react as well to events such as requests for emergency stops or other user commands. At every sampling point such a controller will read the continuous variables as well as the events. Then, it will execute the reaction for the events (some of which may cause mode switching in the continuous dynamics) followed by the computation of the continuous feedback function. Typically, no preemptive scheduling is used in this implementation style and no attempt is made to make efficient use

[5] In fact, this is also sometimes the case in continuous control where sampling rates are determined based on known limitations of the intended implementation platform.

of the computer. To be schedulable, the sum of WCETs of all the possible reactions (computed over the set of all input events that may occur within one sampling period) plus the WCET of the continuous control loop should be smaller than the sampling period.

Tasks and event-triggered implementation

Another popular implementation strategy starts with a collection of discrete controllers, each handling one class of events. Each controller is compiled into a *separate task* which is invoked when the event occurs. This approach requires using an RTOS and some scheduling policy: each event generates an interrupt and the scheduler decides whether to execute the corresponding task or wait for the termination of a task already being executed.

Fixed priority scheduling seems to be the most popular policy for this implementation style where, naturally, higher priority is assigned to tasks with closer deadlines (*deadline monotonic* policy). A nice schedulability analysis has been proposed in [3] for this policy under a minimum inter-arrival time condition. When such a condition holds, the approach does not suffer from the "unpredictability" charges that proponents of the time-triggered solutions tend to put on event-triggered systems [15].

The approach combines nicely with periodic and multi-periodic activations, for instance, by using a fixed priority preemptive scheduling policy for the periodic tasks. Actually, real-time clock activations can be seen as events among others, which are naturally endowed with a minimum inter-arrival time, the period itself. In this sense, this approach generalizes the multi-periodic one and is well adapted to hybrid systems.

Note that the two aspects mentioned, a single program versus separate tasks and periodic versus event-triggered sampling, are somewhat orthogonal. For example the implementation of a program written in the Esterel language is compiled into a single application task as in the time-triggered implementation. Then, this application task runs concurrently with another task, the *event handler*, which detects events and dispatches them for execution when the application task is idle.

4.3 Semantic issues

As we have noted in Section 3.3, variations in execution or communication time may cause changes in the external I/O behavior of controllers. In continuous systems this is restricted to the age of data used by the controller, but in discrete interacting systems the effect of such variations on the behavior of the controller can be more dramatic.

To illustrate this important phenomenon, consider the two automata appearing in Fig. 9 together with their composition. The first automaton reacts to a while the second reacts to b but its reaction depends on the state of the first. As one can see, the state of the system depends on the order in which

the two events arrive. In particular, according to the standard synchronous composition of automata, if a and b occur simultaneously, the outcome (for this example) is different than in the case where a occurs before b. Hence, in order to be faithful to the semantics of the model, the implementation should be able to make unrealistic distinctions. Only "confluent" automata admitting a diamond-like shape (as the one depicted on the right of Fig. 9) have their semantics robust under such variations, but such global automata are obtained only when the individual controllers are practically independent.

As an illustration, consider a periodic sampling implementation and the two signals of Fig. 10 whose respective risings generate the events a and b. A slight shift in the sample times may lead to two different interpretations: in the first a and b are perceived as occurring at the same time while in the second a occurs before b. How do designers take this phenomenon into account? It seems that they apply (consciously or not) tricks borrowed from the asynchronous hardware domain where such phenomena are called hazards or critical races.[6] For instance, they try to ensure that any possible race acts on independent state variables, and if this is not possible, they try to avoid the critical race by forbidding the inputs from changing at almost the same time. This, in turn, is obtained by imposing delays or causality relations between any two inputs that could possibly be involved in a critical race.

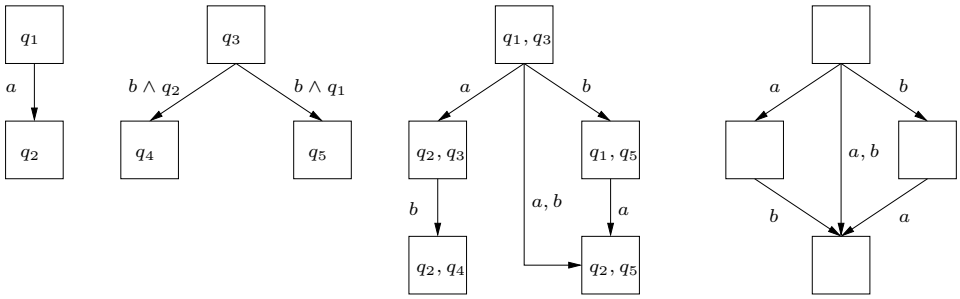

Fig. 9. Two interacting systems and their composition (the transition label a, b indicates that a and b occur simultaneously); a confluent automaton

For event-triggered preemptive implementations this problem is, of course, more severe, and several solutions for it have been proposed. As mentioned previously, the Giotto approach and its extension to discrete events [21] guarantee semantic consistency by deterministic timing of the I/O operations. On

[6] In many applications, software-based control has evolved from previous hardware implementations and the hardware culture is still vivid.

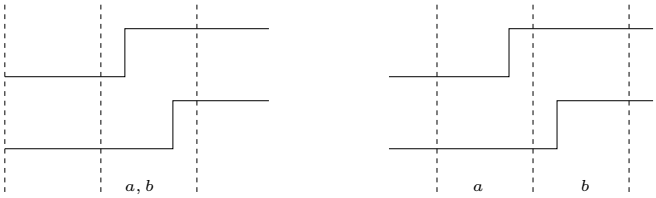

Fig. 10. A pair of signals interpreted differently depending on the sampling

the other hand, the solution proposed in [22] using a multi-buffer protocol achieves the same goal without insisting on timing determinism.

5 Distribution and Fault Tolerance

The preceding sections dealt with control systems implemented on a single computer. However, many large control applications are distributed for various reasons such as location of sensors and actuators, performance or fault tolerance. As a matter of fact, distribution and fault tolerance are strongly related issues: on one hand, fault tolerance usually requires some redundancy which can be implemented as distribution and, on the other hand, distribution raises consistency problems [20] that make fault tolerance more difficult to implement. For this reason we treat them in the same section, which is somewhat superficial, given the huge amount of work dedicated to distributed computing during the past thirty years. We simply mention the major problems and discuss briefly two classes of solutions used in control applications.

A distributed platform consists of several computers, sensors and actuators (nodes) linked together via some communication network through which data can be transmitted. An implementation of a control system on such an architecture consists of assigning controllers to processors, scheduling them and specifying the communication protocol according to which different nodes in the network interact. This architecture aggravates the semantic problems associated with a single computer implementation, namely, variability in execution times and ordering of events, due to communication delays, clock drifts between different processors, etc.

5.1 Local clocks solutions

This is the most widely adopted solution in distributed control systems up to now. The idea is quite simple:

- Each computer has a local real-time clock and runs a periodic (or multi-periodic) application as described in Section 3.

- Each computer samples its external world periodically based on its local clock. This world is made of its physical environment and variables produced by other computers. This amounts to a shared buffer (shared memory) inter-computer communication mechanism.

This solution has many advantages as each computer is complete and acts autonomously. This feature matches pretty well modern aspects of computation and control, as manifested in sensor networks and Internet-based control. The implementation does not require specialized hardware and can thus take advantage of the fast performance improvements and world-wide debugging of mass market products.

Yet, this approach has several drawbacks. Due to the lack of clock synchronization, it yields large jitters that may become larger than the periods. For a purely continuous system this problem is not so severe, because the deviation in the real-time age of data items is always bounded. However, it can become more serious when discrete events are involved. Another drawback is that when two systems are not synchronized, they should observe each other more frequently in order not to miss events.

As shown in [7], redundancies can be implemented on top of such systems in order to achieve fault tolerance.

5.2 Global clock solutions

These are emerging solutions which have been subject to a large research effort in the past years. They are best known as *time-triggered* solutions [15, 23] and are based on the following principles:

- A redundant bus dispatches a common fault-tolerant real-time clock to each computer.
- Communication between computers takes place at fixed points in time determined by the global clock.
- Each computer runs a periodic or non-preemptive multi-periodic (see Section 3.2) application driven by the global clock.

The major advantage of this solution is that it yields small jitters as the timing is very deterministic. It comes equipped with built-in fault-tolerance strategies and with toolboxes integrated with Simulink/Stateflow which alleviate the transition from models to implementation. The drawbacks are exactly the opposite of the advantages noted in Section 5.1: the approach is less flexible and may be more expensive and less efficient as it requires specialized hardware.

As a matter of fact, these two solutions can be seen more as complementary rather than competing. The local clock solution is well adapted to *loosely coupled* (autonomous, asynchronous) systems while the global clock solution matches *tightly coupled* ones. Moreover, in control systems distributed over

large distances, there will always be subsystems that are not synchronized and will need the local clock solution.

Another striking fact about this landscape is that both solutions are time triggered. It seems as if the event-triggered option has not been considered for control-dominated distributed systems, while it is the dominant approach for most distributed systems oriented toward communication and computing. This could be a topic for future research, especially as control and communication become more and more interdependent.

References

1. J. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free shared objects. In *Proceedings of the 16th Real-Time Systems Symposium*, pages 28–37. IEEE Computer Society, 1995. 14
2. K. J. Åström and B. Wittenmark. *Computer Controlled Systems — Theory and Design*. Prentice-Hall, Englewood Cliffs, NJ, 1996. 3
3. N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard Real-Time Scheduling: The Deadline Monotonic Approach. In *Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atlanta, GA, 1991. 19
4. G. Berry and G. Gonthier. The ESTEREL synchronous programming language, design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992. 17
5. P. Caspi and A. Benveniste. Toward an approximation theory for computerised control. In A. Sangiovanni-Vincentelli and J. Sifakis, editors, *2nd International Wokshop on Embedded Software, EMSOFT02*, volume 2491 of *Lecture Notes in Computer Science*, pages 294–304, 2002. 16
6. P. Caspi, A. Curic, A. Maignan, C. Sofronis, and S. Tripakis. Translating discrete-time Simulink to Lustre. In R. Alur and I. Lee, editors, *3rd International Conference on Embedded Software, EMSOFT03*, volume 2855 of *Lecture Notes in Computer Science*, pages 84–99, 2003. 7
7. P. Caspi and R. Salem. Threshold and bounded-delay voting in critical control systems. In Mathai Joseph, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1926 of *Lecture Notes in Computer Science*, pages 68–81, 2000. 22
8. J. Chen and A. Burns. Loop-free asynchronous data sharing in multiprocessor real-time systems based on timing properties. In *Proceedings of the Real-Time Computing Systems and Applications Conference*, pages 236–246, 1999. 14
9. J.-L. Colaço and M. Pouzet. Clocks as first class abstract types. In R. Alur and I. Lee, editors, *Third International Conference on Embedded Software (EMSOFT'03)*, volume 2855 of *Lecture Notes In Computer Science*, pages 134–155, 2003. 8
10. V. Gupta, T. A. Henzinger, and R. Jagadeesan. Robust timed automata. In O. Maler, editor, *Hybrid and Real-Time Systems, HART'97*, volume 1201 of *Lecture Notes in Computer Science*, pages 331–345, 1997. 16
11. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991. 7

12. D. Harel and A. Pnueli. On the development of reactive systems. In *Logic and Models of Concurrent Systems*, volume 13 of NATO *ASI Series*, pages 477–498. Springer–Verlag, Berlin, 1985. 3, 4
13. T. A. Henzinger, B. Horowitz, and Ch. M. Kirsch. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91:84–99, 2003. 15
14. H. Huang, P. Pillai, and K. G. Shin. Improving wait-free algorithms for interprocess communication in embedded real-time systems. In *USENIX 2002 Annual Technical Conference*, pages 303–316. http://www.usenix.org/publications/library/proceedings/, 2002. 14
15. H. Kopetz. *Real-Time Systems Design Principles for Distributed Embedded Applications*. Kluwer, Dordrecht, 1997. 11, 15, 19, 22
16. H. Kopetz and J. Reisinger. Nbw: A non-blocking write protocol for task communication in real-time systems. In *Proceedings of the 14th Real-Time System Symposium*, pages 131–137. IEEE Computer Society, 1993. 14
17. L. Lamport. Concurrent reading and writing. *Communications of ACM*, 20(11):806–811, 1977. 14
18. C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, 1973. 12, 13
19. O. Maler. Control from computer science. *Annual Reviews in Control*, 26:175–187, 2002. 16
20. M. Pease, R. E. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–237, 1980. 21
21. M. A. A. Sanvido, A. Ghosal, and T. A. Henzinger. xgiotto Language Report. Technical Report UCB/CSD-3-1261, Computer Science Division (EECS) University of California Berkeley, July 2003. 17, 20
22. N. Scaife and P. Caspi. Integrating model-based design and preemptive scheduling in mixed time- and event-triggered systems. In *Euromicro Conference on Real-Time Systems (ECRTS'04)*, Catania, June 2004. IEEE Computer Society. 21
23. C. Scheidler, G. Heiner, R. Sasse, E. Fuchs, H. Kopetz, and C. Temple. Time-Triggered Architecture (TTA). In *Proceedings EMMSEC'97*, Florence, Italy, November, 1997. 22
24. L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990. 14
25. R. Wilhelm. Determining bounds on execution times. Handbook on Embedded Systems, CRC Press, Boca Raton FL, 2005. 8