The Evolution of Real-Time Programming

Christoph M. Kirsch
Department of Computer Sciences
University of Salzburg
E-mail: ck@cs.uni-salzburg.at

Raja Sengupta
Department of Civil Engineering
University of California, Berkeley
E-mail: raja@ce.berkeley.edu

October 26, 2006

1 Introduction

Real-time programming has always been one of the most challenging programming disciplines. Real-time programming requires comprehensive command of sequential programming, concurrency, and, of course, time. Real-time programming has many application domains. In this chapter, however, we structure our discussion around digital control systems for the following two reasons. Firstly, digital control systems represent a large portion of real-time software in a diverse set of industries. Examples include automotive power train control systems, aircraft flight control systems, electrical drive systems in paper mills, or process control systems in power plants and refineries. Many of these systems distribute their components over networks with significant communication delays. Therefore, we also discuss networked real-time programming (Section 7). The second reason is pedagogical. Digital control is defined by a set of abstractions that are real-time programmable and mathematically tractable in the context of the dynamics of physio-chemical processes. The abstractions clearly define what application engineers (control engineers) expect of real-time programming. These abstractions therefore constitute a precise definition of the real-time programming problem (Section 2).

Control engineers design in difference equations and often use modeling tools such as Simulink[1] to simulate and validate their control models. The part of Simulink most commonly used to specify digital control, i.e., discrete-time with the discrete fixed-step solver and mode:auto, is well-explained by the so-called synchronous semantics [7, 38]. The key abstraction, we call the *synchronous* abstraction, is that any computation and component interaction happens instantaneously in zero time or with a delay that is exactly the same at each invocation. For example, if a controller computes every 20 msec, either the controller is assumed to read its inputs, compute its control, and write the corresponding outputs instantaneously at the beginning of each period in zero time, or to read its inputs instantaneously at the beginning of the period, compute during the period, and write the corresponding output exactly 20 msec (one period) later. In control parlance, these are the *causal* and *strictly causal* cases, respectively. Systems of difference equations can directly be expressed and structured in this model. The inclusion of strictly causal components (components with delays of a period or more) is typically required to break feedback loops and to account for the unavoidable computation or communication delays that will be present in the real control system.

Real-time programmers design in interrupt handlers, device drivers, and schedulers, which are concepts on levels of abstraction that are obviously unrelated to difference equations. The result is a conceptual and technical disconnect between control model and real-time code that bedevils many control building projects. The control engineer specifies a Simulink design the real-time programmer finds unimplementable. The real-time programmer then fills in gaps to produce an implementation the control engineer finds uncontrollable. The process of interaction as the two iterate to produce a correct design is prolonged by the different backgrounds of control engineers and real-time programmers. This gap explains many of today's problems in real-time software design for digital control such as high validation and maintainance overhead as well as limited potential for reusability and scalability.

[1]www.mathworks.com

However, during the last two decades real-time programming has come closer and closer to the computing abstractions of digital control. This convergence is visible in the form of an increasingly automated real-time programming tool chain able to generate real-time code from high-level specifications produced by control engineer for even large and complex control systems. We tell this story by tracing how real-time programming methodologies have evolved from early, so-called physical-execution-time (PET) and bounded-execution-time (BET) programming [5] to high-level programming models that incorporate abstract notions of time such as synchronous reactive or zero-execution-time (ZET) programming [13] and logical-execution-time (LET) programming [17].

PET programming had been developed to program control systems on to processor architectures with simple instructions that have constant execution times such as many microcontrollers and signal processors. The control engineer had to specify a sequence of instructions, sometimes in an assembly language. One could not declare concurrent, multi-component designs in the dataflow. While this forced the control engineer to work in a low-level language, it resulted in temporally accurate programs and a precise accounting of input-output delay.

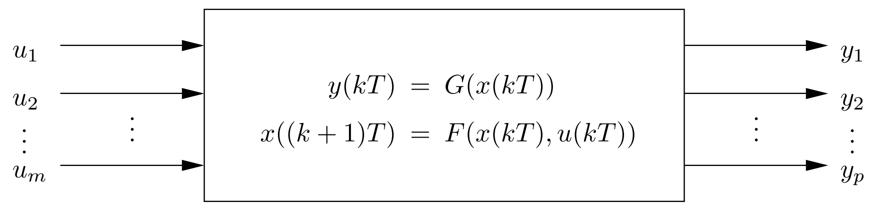
The emergence of operating systems and real-time scheduling led to PET programming being replaced by BET programming. BET programming aimed to handle concurrency and real time. A control engineer could express design in terms of multiple concurrently executing components, i.e, *tasks* in BET terminology. Each task could have a deadline, the period of the controller being the typical example of a deadline. This captured another essential aspect of the computing abstraction desired by the control engineer. Real-time programmers built real-time operating systems that then brought the advances of real-time scheduling theory to bear on the correct temporal execution of these concurrent programs. Each component was expected to execute in the worst case before its deadline or at the correct rate. Earliest-deadline-first (EDF) scheduling and rate-monotonic scheduling [27] emerged as the most successful scheduling disciplines in the context of the BET model. The BET model is still the most widely used real-time programming model. It is supported by a large variety of real-time operating systems and development tools.

The emergence of ZET programming has been motivated by a critical weakness of the BET model with regard to what we call *I/O compositionality*. While BET schedulability analysis techniques check that a collection of tasks with deadlines, when composed, will all execute before their respective deadlines, the same is not true of functional or temporal I/O behavior. The addition of new tasks in parallel to a set of old tasks can change the behavior of the old tasks even if the new tasks have no input-output interactions with the old tasks and the entire task collection continues to meet all deadlines. This is because the order and times when the old tasks interact are not necessarily maintained when a system is augmented. Thus the BET model lacks I/O compositionality.

ZET programming is based on the so-called synchronous semantics in the sense that the semantics of ZET programs are in terms of a zero computing time abstraction. Well-known examples of ZET programming languages are Lustre [14] and Esterel [3]. ZET programs are I/O-compositional in the sense that if new tasks are added to old tasks as previously discussed, the I/O behavior of the old tasks will remain unchanged. This is guaranteed by ZET compilers, which accept concurrent, multi-component control designs but produce a sequential program to be executed by the operating system. Thus ZET compilers take no advantage of the scheduling facilities of real-time operating systems. Thus, while ZET programming brought I/O compositionality to programming real-time control, it lost the BET connection with real-time scheduling.

The most recent of the real-time computing abstractions is the LET model. The notion of LET programming was introduced with Giotto [17]. A LET program is assumed to take a given, strictly positive amount of logical time called the LET of the program, which is measured from the instant when input is read to the instant when output is written. The time the program actually computes between reading input and writing output may be less than the LET. The LET compilation process reproduces the I/O compositionality of the ZET model in its first-stage of compilation. This produces code composed of multiple tasks with their I/O times specified to ensure a concurrent execution that is restricted only to the extent necessary to be consistent with the I/O dependencies between the tasks [16]. This multi-task code with its I/O specifications can be handled by a real-time operating system thereby leveraging the strengths of real-time scheduling during execution.

Both the ZET and LET models close the semantical gap between digital control design and implementation, and have been shown to integrate well with simulation environments such as Simulink [7, 21]. The LET model is closer in the sense that it tries to handle input and output exactly at times modeled in the



$$\begin{aligned} u &= (u_1, u_2, \ldots, u_m) \\ y &= (y_1, y_2, \ldots, y_p) \\ x &= (x_1, x_2, \ldots, x_n) \\ k &= 0, 1, 2, \ldots \end{aligned}$$

Figure 1.1: Basic type of component in a control system

control system, e.g., at the beginning and the end of the controller period. This corresponds exactly to the assumptions made by digital control as discussed in Section 2. ZET programs may produce output before the end of the period, but in the case of strictly causal designs, can conform to the digital control abstraction by buffering the output until the end of the period. The other difference between the two is the use of real-time scheduling and operating systems [21]. Only the buffered, i.e., LET portions of ZET programs have recently been shown to utilize real-time scheduling services such as EDF [37]. Recent research discussed in Section 7 shows the ZET and LET models also extend to some control systems distributed over networks.

In our discussion, we show how the evolution of real-time programming enables modern control software engineering. In the following section, we begin by describing the computing abstractions of control engineering. Then, we continue by covering the previously mentioned four real-time programming models in detail. The final section provides an overview of the most recent work on control software engineering for distributed architectures using deterministic and non-deterministic communication channels.

2 The Computing Abstractions of Control Engineering

In this section, we explain the dominant abstraction in control engineering for the specification of digital control. Figure 1.1 shows the basic kind of component in a control system. Control engineers design digital control in difference equations. The component in the figure has two equations, the first being called the *output* equation and the second the *state* equation. The input signals u are typically measurements from sensors, commands from operators, or commands from other controllers. The output signals y are typically commands to actuators, commands to other controllers, or information displayed to operators. x, called the state in control parlance, is the memory of the component. T is the period of the component. At each expiration of the period T, the component

1. reads its inputs u,
2. computes its outputs y, and
3. updates its state x.

Thus the control engineers assumption about real-time behavior is as illustrated by Figure 1.2. On expiration of each period T, a read and write operation with the environment is expected to occur. It is not expected to occur earlier or later. Thus it is expected that the embedded computing environment will produce an execution that conforms to the idealization as closely as possible. The red arrows indicate that the output values expected at the end of the period depend on the input values read at its beginning. The component has a so-called one-step delay. The intervening time T makes it possible to implement embedded computing processes that conform to the assumptions of the control engineer. Often, the different input and output signals constituting u and y will have different periods or frequencies. If so, the period or frequency of the component will be the highest common factor or the lowest common multiple of the period or frequency

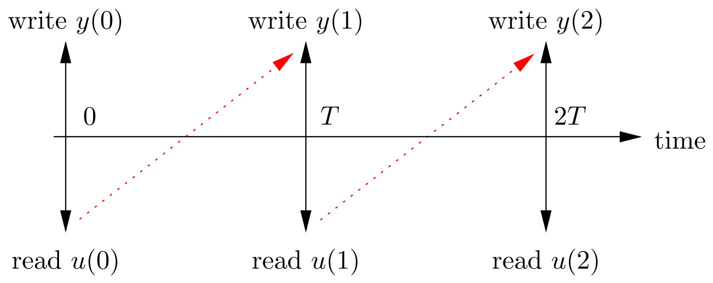


Figure 1.2: Basic timing behavior of a controller

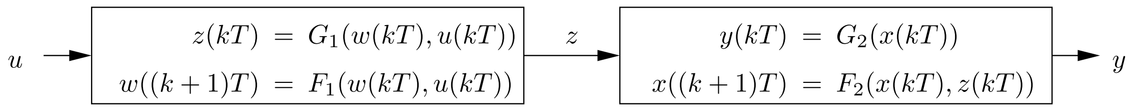

u
$z(kT) = G_1(w(kT), u(kT))$
$w((k+1)T) = F_1(w(kT), u(kT))$
z
$y(kT) = G_2(x(kT))$
$x((k+1)T) = F_2(x(kT), z(kT))$
y

Figure 1.3: A two-component control system

respectively. For example, if the component in Figure 1.1 were to have 2 Hz and 3 Hz input and output signals, the component would compute at 6 Hz, i.e., T would be 1/6 seconds.

Control engineers design large control systems compositionally. The rules of composition are also reasonably well established. Figure 1.3 illustrates a two-component control system. The real-time behavior of each component will be assumed to be similar to Figure 1.2. Note that the first component has no one-step delay, i.e., the output at time T depends on the input at time T. However, this is not a problem if the signal z is internal to the program and only signals u and y interact with the environment. One can see by working through the equations that the dependence between u and y is still delayed by one period. Thus if the embedded computing environment is able to compute the equations in both components within period T it will be able to conform to the control engineers timing assumptions at the interface with the environment, which is all that matters for the correct control of the environment.

Figure 1.4 illustrates a three-component control system to clarify another aspect of the rules of composition. To see this in a simple fashion, we have dropped the state equations and assumed the periods of the three components in Figure 1.4 are the same.

Figure 1.5 illustrates a behavior of the three-component control system. At each expiration of the period, the system would be expected to execute the following actions in the following order:

1. read u,
2. execute block A to output y and z using the new value of u,
3. execute block B to output w using the new value of z, and
4. execute block C to output v based on the new values of w and y.

Thus the data dependencies between components as indicated by the arrows between components constrain the order in which the components are to be executed. In general, the components in the system have to be

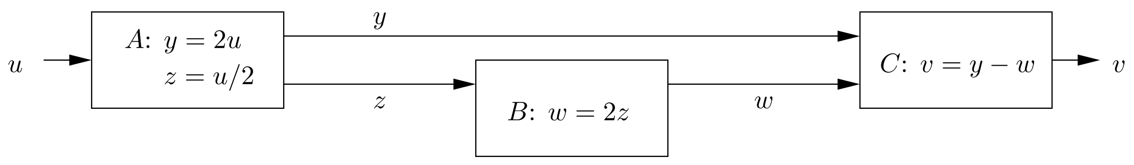
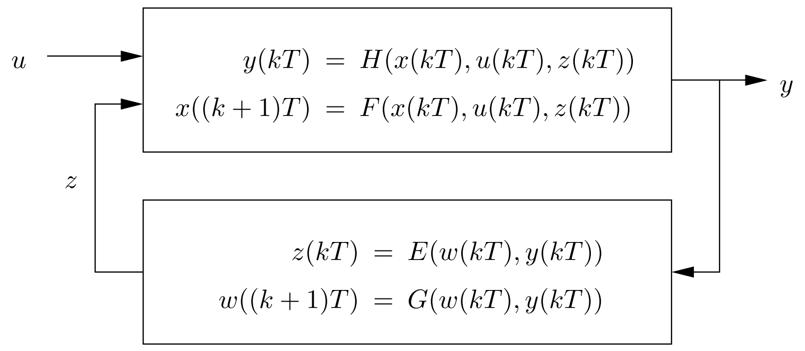


Figure 1.4: A three-component control system

clock	0	1	2	3	4
u	2	4	2	4	2
y $(= 2u)$	4	8	4	8	4
z $(= u/2)$	1	2	1	2	1
w $(= 2z)$	2	4	2	4	2
v $(= y - w)$	2	4	2	4	2

Figure 1.5: Behavior of the three-component control system



u
y(kT) = H(x(kT), u(kT), z(kT))
x((k + 1)T) = F(x(kT), u(kT), z(kT))
y
z
z(kT) = E(w(kT), y(kT))
w((k + 1)T) = G(w(kT), y(kT))

Figure 1.6: A feedback control system

linearized in an order consistent with the arrows inter-connecting the components [38]. Figure 1.5 shows the values of the variables that should be obtained at four consecutive sample times if u were to be as shown in the figure and execution were to conform to the above semantics.

Finally, control engineers connect components in feedback as illustrated by Figure 1.6. In such a case, the signal y is expected to solve the equation

$$y(kT) = H(x(kT), u(kT), E(w(kT), y(kT)))$$

Without restrictions on H or E, it is difficult to provide real-time guarantees on the time required to find a solution. The restriction most commonly assumed is to require at least one of the components in the feedback loop to have a one-step delay. For example, if we assume $z(kT) = E(w(kT))$, the equation above becomes

$$y(kT) = H(x(kT), u(kT), E(w(kT)))$$

The problem now becomes similar to that of computing any other block. In general, a multi-component system with feedback loops having a one-step delay can be handled as before by linearizing the components in any order consistent with the input-output data dependencies.

3 Physical-Execution-Time Programming

```
read u;
compute x := F(x, u);
compute y := G(x);
write y;
```

Program 1.1: A task t

Real-time systems interact with the physical world. Despite functional correctness, temporal accuracy and precision of I/O are therefore particularly relevant properties. Early real-time programmers have addressed

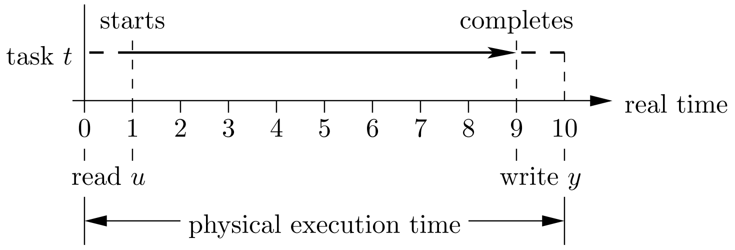


Figure 1.7: The execution of Program 1.1

the challenge of programming such systems by taking the execution times of machine code into account. As an example, consider Program 1.1, which shows an implementation of a task t, and Figure 1.7, which depicts the execution of t. Throughout this chapter, we assume that a *task* is code that reads some input u, then computes from the input (and possibly some private state x) some output y (and new state), and finally writes the output. A task does not communicate during computation except through its input and output. In Figure 1.7, task t is reading input for one time unit from time instant 0 to 1, and then starts computing for eight time units. At time instant 9, t completes computing and is writing its output for one time unit from time instant 9 to 10. Thus the *physical execution time* of task t, i.e., the amount of real time from input to output, which is the only physically relevant time, is ten time units. Here, the intention of the programmer is to have t write its output exactly after eight time units have elapsed since t read its input in order to achieve a physical execution time of ten time units. In other words, the implementation of t is correct only if t computes its output in exactly eight time units. We call this method *physical-execution-time (PET) programming*.

PET programming only works on processor architectures where execution times of instructions are constant and programs can have exclusive access to the processor. Widely used architectures enjoy these properties, e.g., many microcontrollers and signal processors. PET programming results in cycle-accurate real-time behavior and enables high I/O throughput. However, a critical drawback of the PET model is the lack of compositionality. Integrating multiple PET programs on the same processor or distributing PET programs on multiple processors with non-trivial communication latencies is difficult. In the evolution of real-time programming, the following bounded-execution-time model (or scheduled model [24]) can be seen as an attempt to address the lack of compositionality and support for other, less predictable processor and system architectures.

4 Bounded-Execution-Time Programming

```
initialize x;
int n := 0;
while (true) {
  wait for n-th clock tick;
  read u;
  compute x := F(x,u);
  compute y := G(x);
  write y;
  n := n + period;
}
```

Program 1.2: A periodic task t

Real-time systems interact with the physical world. Handling multiple concurrent tasks in real time is therefore an important requirement on such systems. This challenge has traditionally been addressed by

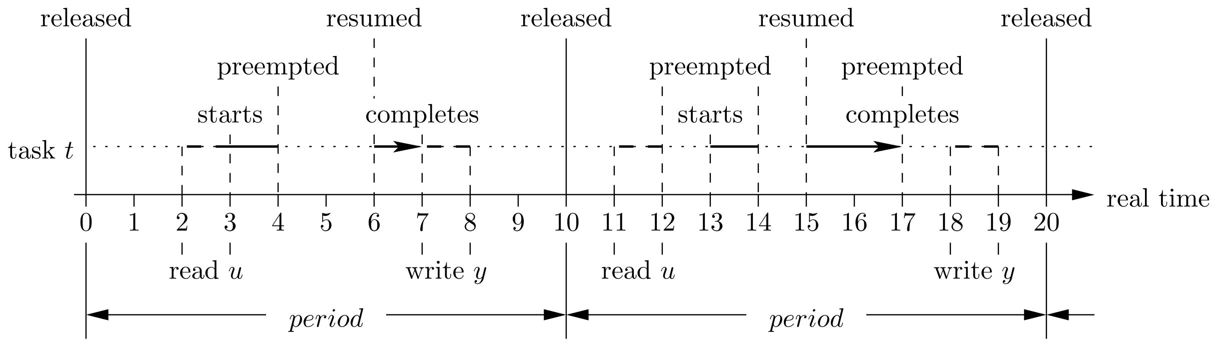


Figure 1.8: The execution of Program 1.2 in the presence of other tasks

imposing real-time bounds on the execution times of tasks. As an example, consider Program 1.2, which shows an implementation of a periodic task t, and Figure 1.8, which depicts the execution of t in the presence of other tasks. At time instant 0, t is released but does not read its input before time instant 2 because other tasks are executing. Task t is reading input for one time unit and then starts computing until time instant 4 at which t is preempted by some other task. Only two time units later, the execution of t is resumed. At time instant 7, t completes computing and immediately starts writing its output for one time unit until time instant 8. At this point, t loops around in its while loop and waits for time instant 10. In the example, the period of t is ten time units. At time instant 10, t is released again but executes differently now because of different interference from other tasks. This time t starts reading input already after one time unit has elapsed since t was released. Moreover, t gets preempted three times instead of just one time, and t needs one time unit more to compute. As a result, compared to t's previous invocation, t starts writing output one time unit later with respect to the time t was released. Nevertheless, as long as t completes reading, computing, and writing before it is released again, i.e., within its *bounded execution time*, the while loop of t will execute correctly. In the example, the bounded execution time of t is equal to t's period, i.e., ten time units. In other words, the implementation of t is correct only if t reads input, computes state and output, and writes output in less than its bounded execution time. We call this method *bounded-execution-time (BET) programming*.

BET programming works on processor architectures for which upper bounds on execution times of individual tasks can be determined, and runtime support is available that handles concurrent tasks according to scheduling schemes for which upper bounds on execution times of concurrent tasks can be guaranteed. An upper bound on the execution time of an individual task is commonly referred to as the *worst-case execution time (WCET)* of the task. WCET analysis is a difficult problem, depending on the processor architecture and the task implementation [15]. For example, memory caches and processor pipelines often improve average performance significantly but are inherently context-sensitive concepts and therefore complicate WCET analyses, in particular, if tight WCET bounds are needed. If loops are present in the task implementation, WCET analysis usually requires programmers to provide upper bounds on the number of loop iterations because of the undecidability of the halting problem.

Many real-time operating systems today provide the necessary runtime support for BET programming. The problem of scheduling concurrent real-time tasks has been studied extensively [6]. There is a large variety of real-time scheduling algorithms and schedulability tests, many of which have been implemented in state-of-the-art real-time operating systems and tools. The most widely used scheduling algorithms in BET systems are probably rate-monotonic (RM) and earliest-deadline-first (EDF) scheduling [27]. The main advantage of BET programming is compositionality with respect to bounded execution times, a property that effectively rests on real-time scheduling theory. A system of BET tasks can be extended by a new task provided the extended system is still schedulable. In this case, all tasks in the extended system will still execute within their bounded execution times. While BET programming is compositional in the sense that the bounds on execution times of individual tasks do not change, testing schedulability usually is not, i.e., testing the extended system may involve reconsidering the entire original system. So-called *compositional scheduling* aims at reducing the need to reconsider already scheduled tasks. Compositional scheduling techniques have only recently received more attention [32][20][11]. The ZET model also benefits from the results in scheduling hybrid sets of real-time and non-real-time tasks [6], which have also made it into real-time operating systems.

```
initialize $x$;
int $n$ := 0;
while (true) {
  wait for $n$–th clock tick;
  read $u$;
  compute $x$ := $F(x, u)$;
  lock $s$;
  share $x$;
  unlock $s$;
  compute $y$ := $G(x)$;
  write $y$;
  $n$ := $n$ + $period$;
}
```

Program 1.3: A periodic task h sharing its state using a mutex s

The BET model is probably the most widely supported real-time programming model. Many real-time programmers are familiar with this model. Since enabling BET programming is essentially a matter of adequate operating system support, existing programming languages can easily be utilized in developing BET programs. Moreover, there are many real-time programming languages [5] that are based on the BET model such as Ada and PEARL. Recently, a real-time Java specification (RTSJ), which is also based on the BET model, has been proposed and implemented. Real-time scheduling theory even provides results on how to schedule processes and threads in real time that share data through potentially blocking mechanisms such as semaphores and mutexes. In other words, the BET task model may even be and has been extended to, e.g., standard process and thread models. As an example, consider Program 1.3, which shows a periodic task h that shares its state with other, possibly periodic tasks using a mutex s, and Figure 1.9, which depicts an execution of h sharing its state with a task l. At time instant 0, tasks h and l are released where h has a period of ten time units and l has a period of, say, thirty time units. The example shows an EDF schedule of both tasks, i.e., h is scheduled to execute first. We assume in this example that deadlines are equal to periods. The first attempt of h to lock s at time instant 2 succeeds immediately because no other task holds the lock on s. At time instant 6, h has completed writing its output and l is scheduled to execute next. At time instant 9, l's attempt to lock s succeeds, again immediately because no other task holds the lock on s. At time instant 10, however, h is released again and scheduled to execute next because h has an earlier deadline than l. As a consequence, l is preempted while holding the lock on s. At time instant 12, h attempts to lock s but is blocked because l still holds the lock on s. Since l is the only unblocked task, l is scheduled to execute next. One time unit later, l unlocks s. As a result, h is unblocked and immediately receives the lock on s. Then, h is scheduled to execute next because h has an earlier deadline than l. Finally, h unlocks s at time instant 15 and completes writing its output at time instant 18. Here, the interesting system anomaly is from time instant 12 to 13, which is traditionally called *priority inversion* because a task with lower priority (task l, which has a later deadline) is scheduled to execute in the presence of a task with higher priority (task h, which has an earlier deadline). Priority inversion is problematic because it gives tasks, which may even never attempt to lock s, the chance to prevent both task h and l from executing. For example, if a task m is released during priority inversion and m has a priority that is less than h's priority but greater than l's priority, then m is scheduled to execute instead of l. Therefore, m may not only prevent l from ever unlocking s again but also, as a result, prevent h from ever executing again although m has a lower priority than h. Priority inversion is often explained using all three tasks l, m, and h at once. However, priority inversion itself merely requires h and l whereas m is only necessary to explain the danger of priority inversion. In other words, only in the presence of tasks such as m, priority inversion may be harmful and needs to be avoided using techniques such as *priority inheritance* or *priority ceiling* [31]. For example, with priority inheritance, l would inherit the priority of h from time instant 12 to 13 and thus prevent m from being scheduled to execute during that time.

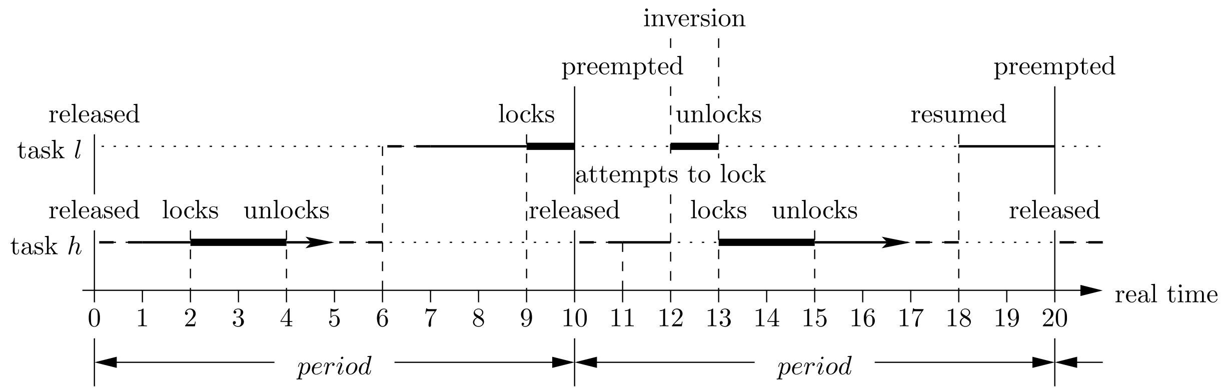


Figure 1.9: The execution of Program 1.3 sharing its state with a task l

Instead of blocking mechanisms such as semaphores and mutexes, BET tasks may also utilize non-blocking communication techniques such as the *non-blocking write protocol (NBW)* [26] or extensions thereof [23] in order to avoid priority inversion. For example, with NBW techniques, write attempts always succeed while read attempts may have to be repeated because write attempts were in progress. In any case, there remains the burden of showing that all task executions stay within their bound execution times. Blocking techniques require that the time to communicate is bounded using scheduler extensions such as, e.g., priority inheritance. Non-blocking techniques require that the number of retries to communicate is bounded, which is possible if the maximum amount of time needed to communicate is considerably shorter than the minimum amount of time between any two communication attempts.

The main drawback of the BET model is the lack of compositionality with respect to I/O behavior, which is a semantically stronger concept than compositionality with respect to bounded execution times. A system of BET tasks may change its I/O behavior, i.e., output values and times, as soon as tasks are added or removed, the scheduling scheme changes, or the processor speed or utilization are modified, even though the modified system might still be schedulable. Real-time programmers therefore frequently use the infamous expression "priority-tweaking" to refer to the BET programming style because it is often necessary to modify scheduling decisions manually to obtain the required I/O behavior. Once a BET system has been released, it is hard to change it again because of the system-wide scheduling effects. As a consequence, BET systems are often expensive to build, maintain, and reuse, in particular, on a large scale. The following zero-execution-time model (or synchronous model [24]) can be seen as an evolutionary attempt to address the lack of semantically stronger notions of compositionality in the BET model.

5 Zero-Execution-Time Programming

```
initialize x;
while (true) {
  read u at next occurrence of event;
  compute x := F(x,u);
  compute y := G(x);
  write y;
}
```

Program 1.4: A synchronous reactive task t

Real-time systems interact with the physical world. Designing and implementing such systems in a way that is compositional with respect to I/O behavior is therefore an important challenge. Compositionality in this sense requires that a real-time program always produces the same sequence of output values and times

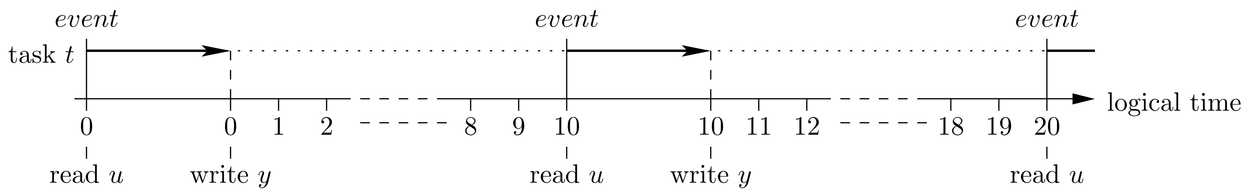


Figure 1.10: The logical semantics of Program 1.4

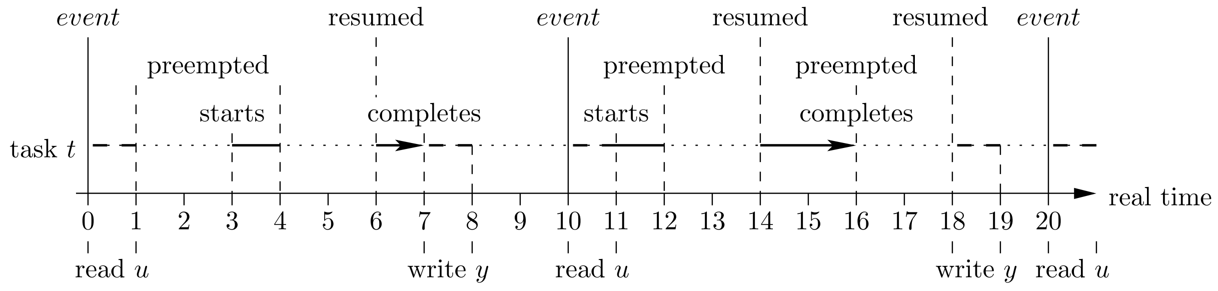


Figure 1.11: An execution of Program 1.4 in real time

for the same sequence of input values and times, even when running in the possibly changing presence of other programs, or on different processors or operating systems. We say that a program, i.e., its I/O behavior, is *input-determined* if, for all sequences I of input values and times, the program produces, in all runs, unique sequences $f(I)$ of output values and times. So-called *synchronous reactive programming* [13] aims at designing and implementing input-determined real-time programs. Note that we deliberately avoid the term "deterministic" here because it has been used with very different meanings in different communities. BET communities such as the real-time scheduling and real-time operating systems communities call a real-time program deterministic if the program always runs with bounded execution times. Language-oriented communities such as the synchronous reactive programming community call a real-time program deterministic if the I/O behavior of the program is fully specified by the program. In other words, the term "input-determined" is related to the term "deterministic" in the sense of the language-oriented communities.

The key abstraction of the synchronous reactive programming model is to assume that tasks have logically *zero execution time*. As an example, consider Program 1.4, which shows an implementation of a synchronous reactive task t, and Figure 1.10, which depicts the semantics of t based on some logical clock. At the occurrence of the first event at time instant 0, t reads its input, computes state and output, and writes its output logically in zero time. The same happens at the occurrence of the next event that triggers t at time instant 10, and so on. In other words, each iteration of t's while loop is completed instantaneously before any other event can occur. In terms of the logical clock, task t is therefore an input-determined program. In terms of real time, Figure 1.11 depicts an execution of t that approximates the logical semantics in the sense that t's output values are still input-determined while the time instants at which t writes its output are only bounded by the duration between the current and next event, i.e., by ten time units in the example. Therefore, we also say that the synchronous reactive programming model is compositional with respect to bounded I/O behavior. Note that for t's output values to be input-determined, it is necessary that t reads its input as soon as the event occurs that triggered t while the actual computation of t may be preempted at any time. For example, t is preempted at time instant 1 right after t read its input, or at time instant 16 right after t completed computing before writing its output. As a consequence, output may be written with latency and jitter. In the example, the output jitter is one time unit. In other words, the implementation of t is correct if t reads input at the occurrences of the triggering events but then computes state and output, and writes output only some time before the next event occurs. Despite synchronous reactive programming, we also call this method figuratively *zero-execution-time (ZET) programming*.

Similar to BET programming, ZET programming works on processor architectures for which WCETs of individual tasks can be determined. The runtime support required for ZET programming is often simpler than for BET programming because ZET tasks are usually compiled into a single sequential program that implements some form of finite state machine. As a consequence, a ZET runtime system may only provide mechanisms for event handling and buffering of input values. Shifting complexity from ZET compilers to runtime systems by utilizing dynamic scheduling schemes such as EDF at runtime has only recently received more attention. The challenge is to minimize the amount of memory (buffers) required to preserve the logical semantics of ZET tasks when using dynamic schedulers. Similar to BET programming, the advantage is potentially more effective processor utilization but possibly at the expense of temporal accuracy and precision of I/O. There is a large variety of ZET programming languages. *Esterel* [4] and *Lustre* [14] are probably the most widely used languages. Programming in Esterel is similar to imperative programming while programming in Lustre is more declarative and oriented towards data rather than control flow.

ZET programming has several advantages. The ZET semantics is close to the semantics of modeling tools such as *Simulink*, which are used to develop and simulate controller designs. ZET programming therefore enables model-based design of embedded control systems in the sense that control models may directly be translated into executable ZET programs [29]. Moreover, ZET programs may be verified for functional correctness because most ZET programming languages have formal semantics. ZET-based implementations of embedded control systems are thus suitable for mission- and safety-critical applications and have already been deployed successfully by railway and avionics companies.

The ZET model also has several disadvantages. Firstly, ZET programming distributed systems with non-negligible communication latencies is complex but has been shown to be possible on platforms with [7] and without built-in clock synchronization [2]. In the presence of unreliable communication channels, the problem is even more difficult, see Section 7 for more details. Secondly, in contrast to the BET model, most ZET programming languages are self-contained, i.e., offer combined timing and functional primitives, and thus have often not been integrated with standard programming languages. This may not be an inherent problem of the ZET model but still an economical obstacle. Thirdly, ZET programs are typically compiled into static executables with no support for dynamic changes at runtime. In other words, while the ZET model is compositional with respect to bounded I/O behavior, the compositionality may only be utilized at design time. The following logical-execution-time model (or timed model [24]) is a recent attempt to address these issues.

6 Logical-Execution-Time Programming

```
initialize $x$;
int $n$ := 0;
while (true) {
  int $k$ := $n$ + $offset$;
  read $u$ at $k$–th clock tick;
  compute $x$ := $F(x,u)$;
  compute $y$ := $G(x)$;
  $k$ := $k$ + $let$;
  write $y$ at $k$–th clock tick;
  $n$ := $n$ + $period$;
}
```

Program 1.5: An LET task t

Real-time systems interact with the physical world. The I/O behavior of such systems and, in particular, the timing of I/O is therefore an important aspect, which has recently received more attention in the latest evolutionary attempt at defining real-time programming models. We call this method *logical-execution-time (LET) programming*. The key abstraction of the LET model can be phrased in a computation-oriented and an I/O-oriented way. In the LET model, a task computes logically, i.e., from reading input to writing output, for some given, positive amount of time called its *logical execution time*. Equivalently, the input and output

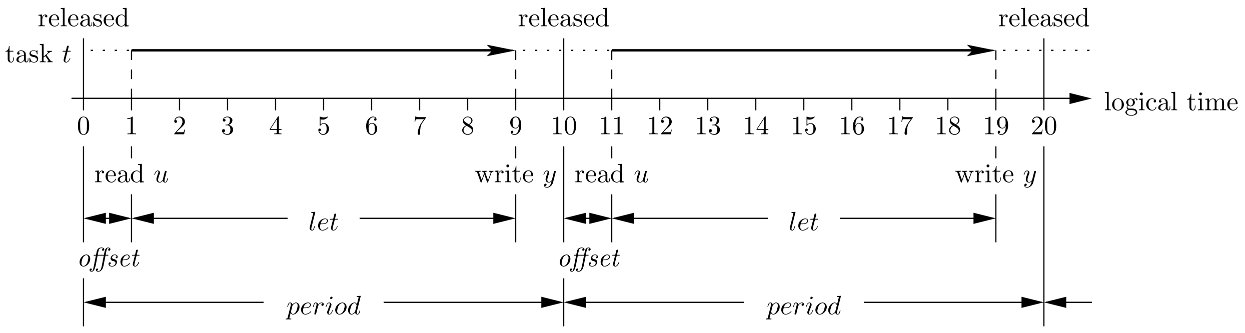


Figure 1.12: The logical semantics of Program 1.5

of a task in the LET model is read and written, respectively, at some given instants of time, independently of when and for how long the task actually computes. As an example, consider Program 1.5, which shows an implementation of an LET task t, and Figure 1.12, which depicts the semantics of t based on some logical clock. At time instant 0, task t is released but only reads its input with an offset of one time unit at time instant 1. Then, t is computing for eight time units. At time instant 9, t writes its output. Thus the logical execution time of t is eight time units. At time instant 10, t is released again and repeats the same behavior. Similar to the ZET model, an LET task such as t is therefore an input-determined program in terms of the logical clock. In terms of real time, Figure 1.13 depicts an execution of t that approximates the logical semantics in the sense that t's output values are still input-determined while the time instants at which t writes its output are only bounded by the amount of time needed to perform I/O of other tasks at the same time instants. For example, if there is another task u that needs to write its output at time instant 9, then either u's or t's output gets delayed by the amount of time it takes to write the output of the other task. Therefore, similar to the ZET model, we say that the LET model is compositional with respect to bounded I/O behavior but, unlike the ZET model, the output times are bounded by the overall I/O load rather than the duration between events. In the LET model, if I/O load is low, which is true for sensor and actuator I/O in many control applications, then temporal accuracy and precision of I/O may be close to hardware performance. Note that the input times can in fact not be arbitrarily accurate and precise as well but are also just bounded by the overall I/O load. The ZET model is affected by the same phenomenon. We omitted this observation in the section on the ZET model to simplify the discussion. Figure 1.13 shows that the actual times when task t computes do not affect the times when t reads input and writes output. During t's first invocation, t starts computing with a delay of one time unit at time instant 3, is preempted once at time instant 4, and requires two time units to compute. During the second invocation, t starts computing right after reading its input, is preempted once at time instant 13, and requires three time units to compute. Note that even though t completes computing early, t waits until time instants 8 and 18 to start writing output in order to approximate its logical execution time as close as possible. In other words, the implementation of t is correct if t starts computing some time after t completes reading its input, and t completes computing some time before t starts writing its output. In this case, we say that t's implementation is *time-safe* [18].

LET programming works on processor architectures for which WCETs of individual tasks can be determined. The runtime support required for LET programming is a combination of the runtime support necessary for BET and ZET programming. LET tasks need to be scheduled to execute, like BET tasks, according to some real-time scheduling strategy such as, e.g., RM or EDF scheduling. Accurate and precise timing of task I/O requires mechanisms for event handling similar to the mechanisms needed for ZET programming. There are four LET programming languages, called *Giotto* [17], *xGiotto* [12], *HTL* [11], and *TDL* [36]. The notion of LET programming was introduced with Giotto, which supports the possibly distributed implementation of periodic tasks whose logical execution times must be equal to their periods. Giotto also supports switching modes, i.e., switching from one set of tasks to another. xGiotto extends Giotto with support for event-triggered tasks. TDL extends a subclass of Giotto with a module concept. HTL is the most recent language, which supports LET programming of tasks with logical execution times less than their periods. All four languages are timing languages, which still require another programming

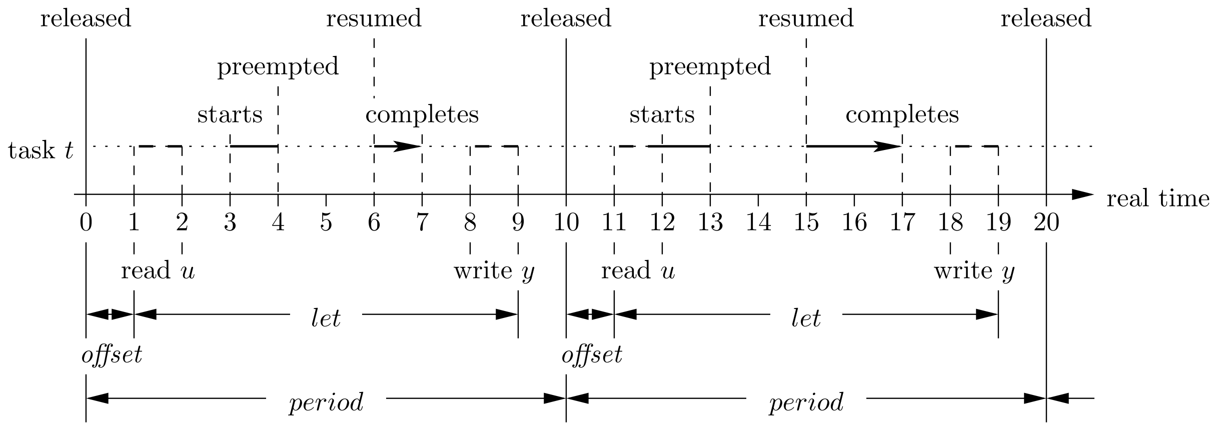


Figure 1.13: An execution of Program 1.5 in real time

language for the implementation of the tasks. For example, Giotto programs only determine when tasks are released, read input, and write output but not what and how the tasks compute. The existing LET languages are all compiled into so-called *E code*, which is executed by a virtual machine called the *Embedded Machine* [16]. E code is low-level timing code that determines when tasks are released, read input, and write output. The Embedded Machine still requires, e.g., an EDF scheduler to execute the tasks. Note that the execution of LET tasks may also be scheduled at compile time using so-called *schedule-carrying code (SCC)* [19], which is E code extended by instructions that explicitly dispatch tasks to execute. With SCC, no runtime scheduler is needed to execute LET programs.

LET programming has several advantages that are rooted in the compositionality of the LET model. Real-time programs written in the LET model are predictable, portable, and composable in terms of their I/O behavior. For example, Program 1.5 will show the same I/O behavior modulo I/O load on any platforms and in any contexts as long as the program executes in a time-safe fashion. Thus LET programming can be seen as a generalization of the early PET programming style to handle concurrent tasks while maintaining temporal accuracy and precision of the individual tasks' I/O. Note that LET programs may also implement conditional timing behavior while still being input-determined such as, e.g., changing the frequencies of tasks or even replacing tasks by others at runtime [11]. An important advantage of Giotto is that time safety of Giotto programs can be checked in polynomial time with respect to the size of the programs although there might be exponentially many conditional timing behaviors [18]. Checking time safety of programs written in more general LET languages such as HTL may only be approximated with less than an exponential effort [11]. The LET semantics is abstract in the sense that platform details such as actual execution times and scheduling strategies are hidden from the programmer. Similar to ZET programs, modeling tools such as Simulink can be used to develop and simulate LET programs. LET programming therefore also enables model-based design in the sense that models may directly be translated into semantically equivalent executable programs [21]. Finally, LET programs written in Giotto [20], HTL [11], and TDL [9] have been shown to run on distributed systems with support for clock synchronization.

LET programming also has several disadvantages. A LET task may not take advantage of available processor time in order to compute and write output faster than its logical execution time. Unused processor time may only be used by other tasks. With the LET philosophy, getting a faster processor or better scheduler, or optimizing the implementations of tasks does not result in faster I/O behavior but in more processor availability to do other work. While this can be seen as a strength for certain applications including control, it can also be seen as a weakness for other applications. Nevertheless, the essence of LET programming is to program explicitly how long computation takes or, equivalently, when I/O is done. So far, with the exception of xGiotto, only real-time clock ticks have been used to determine the LET of tasks although other types of events may be used. LET programming is the most recent style of real-time programming and still immature compared to the more established concepts. LET programming languages and development tools are mostly prototypical, and have not been used in industrial applications.

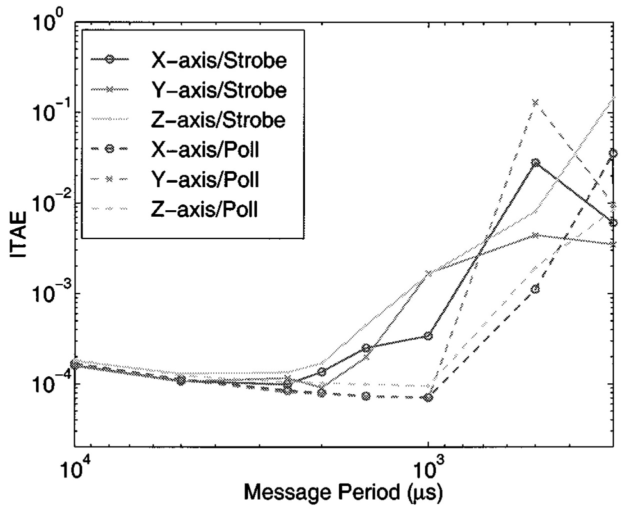


Figure 1.14: Control performance over Ethernet [10]

7 Networked Real-time Systems

This section provides an overview of the most recent work on control software engineering for distributed architectures using deterministic and non-deterministic communication channels. Recall the two- and three-block control systems illustrated by Figures 1.3 and 1.4, respectively. It is usually assumed the inter-component data flows represented by the arrows between the blocks are instantaneous. Here, we discuss the real-time programming of systems in which these dataflows take time. Within the control community this class of systems has come to be called *Networked Control Systems (NCS)*.

Well-established NCS examples include systems of many controllers connected over a field bus in a process plant, or controllers communicating over a CAN[2] bus in a car. The networking revolution is also driving an efflorescence of new control systems for new domains like tele-surgery, smart cars, unmanned air vehicles, or autonomous underwater vehicles, distributed over the Internet or wireless LANs. This surge in NCS has led to new techniques for networked real-time programming.

We sub-divide networked real-time systems into two categories distinguished by the determinism of the network inter-connecting components. We call the categories deterministically networked real-time systems (DNRTS) and non-deterministically networked real-time systems (NNRTS). The DNRTS category abstracts systems in which inter-component communications over the network are highly reliable. The loss of messages is rare enough and jitter in the inter-message time interval is small enough to be treated as exceptions or faults that are specially handled. Control systems inter-connected over TTA [25], or lightly-loaded CAN[3] or Ethernets are usually in this category. The NNRTS category abstracts systems where loss or jitter in inter-controller communications is so frequent that it must be viewed as part of normal operation. In this category are systems in which inter-controller communications flow over TCP connections on the Internet, as UDP messages over wireless LANs like Wi-Fi[4] or Bluetooth[5], and well-utilized Ethernet or CAN.

Figures 1.14, 1.15, and 1.16 illustrate the impact of network losses on real-time control over Ethernet, CAN, and a token passing network. The figures are reproduced from [10]. The vertical axis of the figures shows the performance of a three-axis machine tool under real-time control. Performance gets worse up the axis. There are three controllers that aim to position the machine tool as desired on the X, Y, and Z-axes. The controllers send their commands to and receive their measurements from the machine tool over a shared network. In all other respects, they are decoupled from each other. The horizontal axis shows the sampling rate. The sampling rate increases to the right. Figure 1.14 shows performance when the machine is controlled

[2]CAN standard ISO 11898-1
[3]CAN Standard ISO 11898-1
[4]http://standards.ieee.org/getieee802/802.11.html
[5]http://www.bluetooth.com

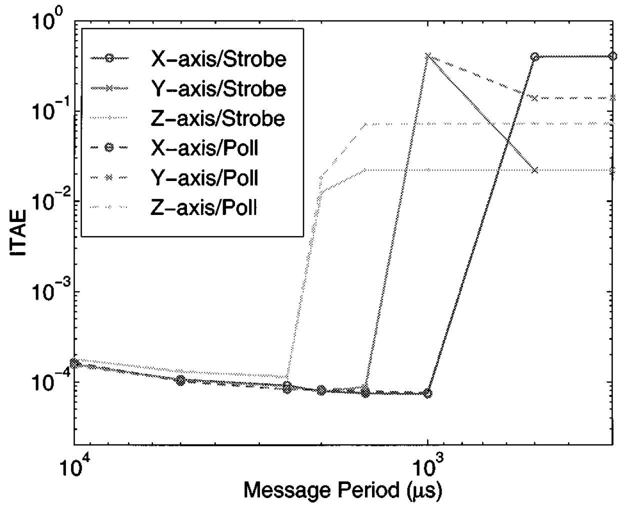


Figure 1.15: Control performance over CAN [10]

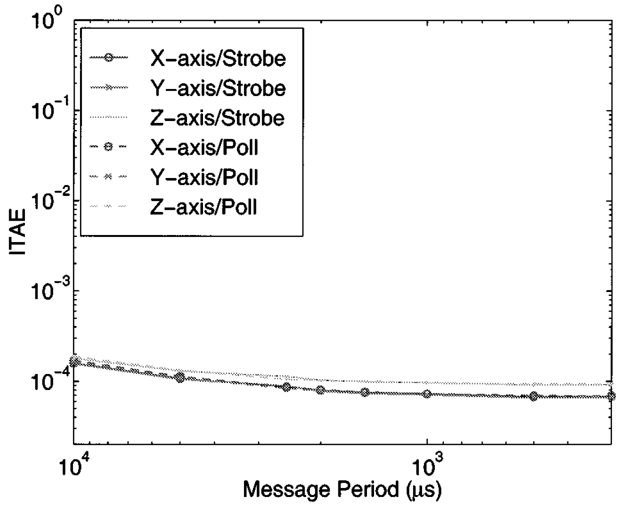


Figure 1.16: Control performance over Wired Token Network [10]

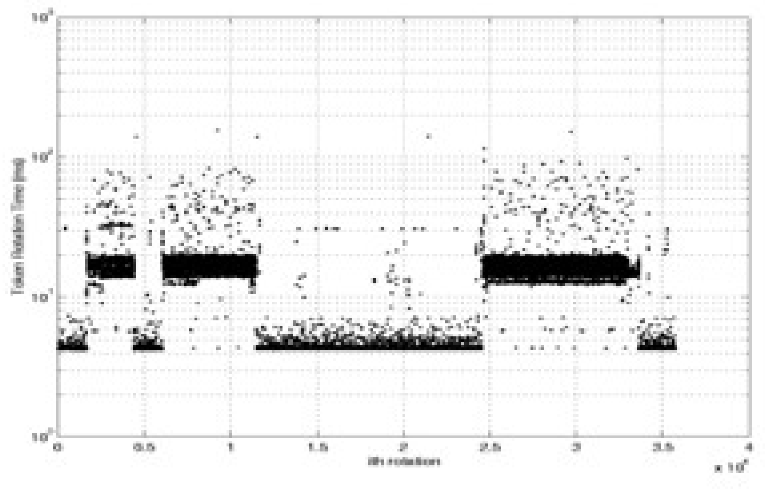


Figure 1.17: Channel access jitter

over an Ethernet. Initially, the Ethernet is under-utilized. As the sampling rate is increased performance improves. When there is too much traffic on the Ethernet, messages are lost, and performance deteriorates. Figure 1.15 shows performance when the controllers communicate over a CAN network. Like Ethernet, CAN is a contention access network. Any node with a message to send may contend for the bus if it detects the bus is idle. Therefore, one observes a performance pattern similar to Figure 1.14. However, unlike Ethernet, each CAN message can have a priority level. When multiple messages contend for the bus at the same time, the higher priority wins. Here, the Z-axis controller has the highest priority and the X-axis controller has the lowest priority. The figure shows the performance of the low priority control system deteriorates first, followed by the medium and high priority control systems. Thus Ethernet and CAN at low utilization can be viewed as DNRTS. At high utilization we view them as NNRTS.

Figure 1.16 shows performance when the controllers share the channel using a token-passing protocol. The network has a token that passes in a fixed order through the controllers with each vehicle receiving the token at fixed periods. On receiving the token it sends its data while the others are silent. This ensures each controller sends and receives all its data as long as the total channel data rate is not exceeded. The performance will remain good until the channel data rate is exceeded at which point it will cease to function. Token-passing protocols represent a way to realize DNRTS for real-time control over wired networks. They are an alternative to the time-division multiple-access (TDMA) approach followed by TTA. They also improve the determinism of delay over wireless networks though not as greatly.

Figure 1.17 is an example of jitter in channel access delays associated with a wireless token passing protocol [8]. The protocol is configured to give controllers access at 50 Hz, i.e., every 20 msec. The control system is activated three times as represented by the three bursts near 20 msec. Most access delays are clustered around 20 msec. However, as one can see, there is considerable jitter. The data is in UDP broadcast packets. The broadcasts are either received almost immediately or not at all as illustrated by the Erasure/Loss line in Figure 1.20.

More recently, the TTA architecture has been fused with CAN to create TTCAN[6] for wired communication. This is more suitable as a networking environment for DNRTS. Other networking environments such as the token bus (IEEE 802.4) or polling protocols can deliver deterministic access delays to network nodes. These work well when all components at a node send messages with the same period. However, when network nodes have control components computing at multiple rates, TTA provides the best support.

[6]TTCAN standard ISO 11898-4

$$\begin{aligned} u &\rightarrow 2,4,.,.,.,2,4,.,2,... \\ y &\rightarrow 4,8,4,.,.,8,4,.,.,... \\ z &\rightarrow 1,.,2,1,2,1,.,.,.,... \\ w &\rightarrow 2,4,.,2,.,4,.,.,.,... \end{aligned}$$

Figure 1.18: Logical behavior of the three-block control system

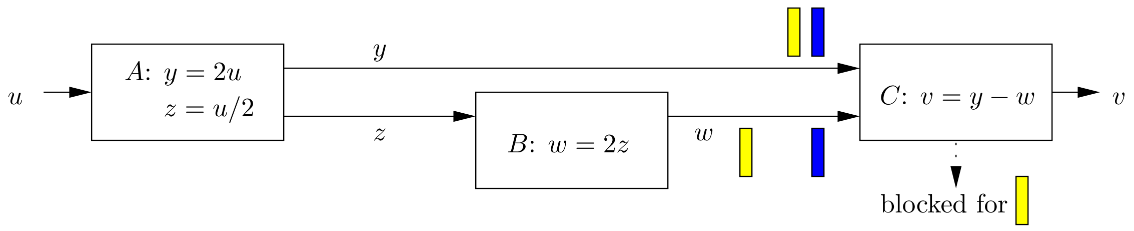


Figure 1.19: A compilation scheme to preserve the logical semantics over FIFO channels

7.1 DNRTS

Research over the last decade indicates engineers should be able to use high-level tools like Simulink or Lustre to program DNRTS. We explain this abstractly in terms of the computing semantics of digital control in Section 2. For each variable in a program and a sequence of values for each of its input variables, the semantics gives us the sequence of values taken by the variable and the timing of each value (see, for example, Figure 1.5). We will refer to the set of value sequences as the logical aspect of the semantics and the timing as the real-time aspect of the semantics. Preserving the logical aspect in a distributed environment is now well understood [38, 1]. It turns out to be possible under pretty weak assumptions on the network. Preserving the real-time aspect is not as well understood and appears challenging. We will discuss this in the context of the Time-Triggered Architecture (TTA) and comment briefly on other networking environments. Amongst current networking environments, this one may be the best suited for DNRTS.

Suppose we have a three-block control system as shown in Figure 1.4. Assume each block is on a computing node and the input-output connections between the blocks, denoted by the arrows, need to be network channels. This sort of control schematic is easily specified in Simulink without the network channels. In Simulink, each block will also have a period at which it reads and writes its outputs. The literature gives insight into the following questions:

- What sort of channels can enable us to preserve the logical aspect of the semantics?
- What sort of channels can enable us to preserve the real-time aspect of the semantics?

It turns out the logical aspect can be supported by many channels. For example, any reliable First in First Out (FIFO) channel works. In other words, a TCP channel can do the job.

Figure 1.18 illustrates the logical aspect of the semantics in Figure 1.5. We care only about the sequence of values for each variable but not about the vector of values represented by the columns in Figure 1.5. An execution that preserves the logical aspect of the semantics will be one that preserves the sequence of values for each variable. It may not preserve any ordering of values across the variables, i.e, the vector.

If the components are connected by reliable FIFO channels, there is a conceptually simple compilation scheme able to preserve the logical semantics. This is illustrated by Figure 1.19 in the context of the three-block example. Basically, each component must be compiled to block until it receives the right number of inputs on each channel. Since all blocks in Figure 1.19 are assumed to have the same rate, this means block C should block until it receives the first inputs on both the y and w ports, compute the first value of v as soon as it does so, and then block until it receives another pair of inputs. If the second value on port w is delayed relative to the second value on port C, then C must block for w.

Locally, each program is required to behave exactly like the usual synchronous program illustrated by Program 1.4. Components on different network nodes can be driven entirely by the arrival of messages on the input-output channels. Thus there is no global clock or scheduler, rendering the architecture Globally Asynchronous and Locally Synchronous, abbreviated GALS by Benveniste et.al. [1]. If the synchronous subsystem on each network node is compiled to accept its inputs in any order, this kind of compilation is also modular. Any change to the program on a network node, will require re-compilation of that node alone. It may also be noted that the same scheme has been shown to work for programs with feedback loops provided each loop has a delay or is strictly causal. In other words, each loop should include at least one component whose output at time k depends only on inputs up to time $k-1$.

Synchronous programming languages like Esterel subsume asynchronous programming by permitting signals to be either present or absent at each execution of the program. If one seeks to distribute such programs without controlling execution with respect to one master clock, the potential for signals to be absent poses a conundrum. The absence of a signal may be meaningless without the master clock. While all such programs cannot be distributed in a GALS architecture, a sub-class identified as the isochronous programs can be distributed without a master clock. An isochronous program pair is one where in any state pair from the two programs, unification of the two states over the present common variables implies unification over the entire set of common variables [1]. In other words, the values of signals supplied by the environment is sufficient to determine the absence of the other signals thus rendering any checking of signal presence with respect to any master clock unnecessary.

We now turn to our second question, i.e., preservation of the real-time aspect. Meeting periodic execution deadlines in DNRTS requires two kinds of schedulability. Computation must be schedulable at each computing node and communication must be schedulable between nodes. Not only must the inter-controller channels be reliable and FIFO, they must also provide access to computing nodes at scheduled times. Alternatively, the communication channels between computing nodes must be under-utilized enough to support a zero communication time abstraction, i.e., a node communicates its output to the others as soon as it is ready. A high bandwidth wired Ethernet may support such an abstraction. In this case, the compilation problem may be the same as that discussed in Sections 5 and 6.

The joint scheduling of computation and communication has been investigated over TTA [7]. TTA is well suited to the computing abstractions of digital control. It supports distributed implementations built upon a synchronous bus delivering to computing nodes a global fault-tolerant clock. Each computing node connects to the bus using a network card running the Time-Triggered Protocol (TTP). The bus is required to have a communication schedule determined *a priori*. This is called the *Message Description List* (MEDL). It specifies which network node will transmit which message at which time on a common global timeline. The timeline is divided into cycles. The cycle is the unit of periodic operation, i.e., the composition of each cycle is the same. The MEDL specifies the composition of the cycle. Within a cycle there are rounds of identical length. Each round is composed of slots. These can have different lengths. Each slot is given to a network node. Within the slot, the node sends a frame composed of one or more messages. Thus the objective of communication scheduling when distributing over TTA is specification of the MEDL.

The approach in [7] compiles both Simulink and Lustre over TTA. Simulink programs are handled by translating them to Lustre. The control engineer is assumed to specify the allocation of components or blocks to nodes. One or more components at each node may have input-output connections to components on other computing nodes. The compilation process starts with components annotated with deadlines, worst-case execution time assumptions, and the real-time equivalents of the logical clocks driving the component. The deadlines are relative. For example, if a component has an output y and an input u, a deadline would specify the maximum time that may elapse between the availability of u and the availability of y. The compilation process can be thought of as a two-step process with the first step handling the logical aspect and the second the real-time aspect.

The first step of compilation, called the *Analyzer* [7], builds the syntax tree and global partial order across all the components in all the nodes in the system. This partial order accounts for all the dataflow dependencies between components and represents constraints that will preserve the logical aspect of the semantics. The results of this stage of compilation with the annotations previously mentioned and the partial order constraints can be passed to the next compiler called the *Scheduler* because it generates the computing schedule at each node and the MEDL for the bus. The scheduling problem is a Mixed Integer Linear Program (MILP) and computationally difficult. To reduce complexity, the developers allow the fine-grain partial order over all components to be lumped into coarser grain super-components that include some of the original components as sub-components. The Scheduler then works with the super-components,

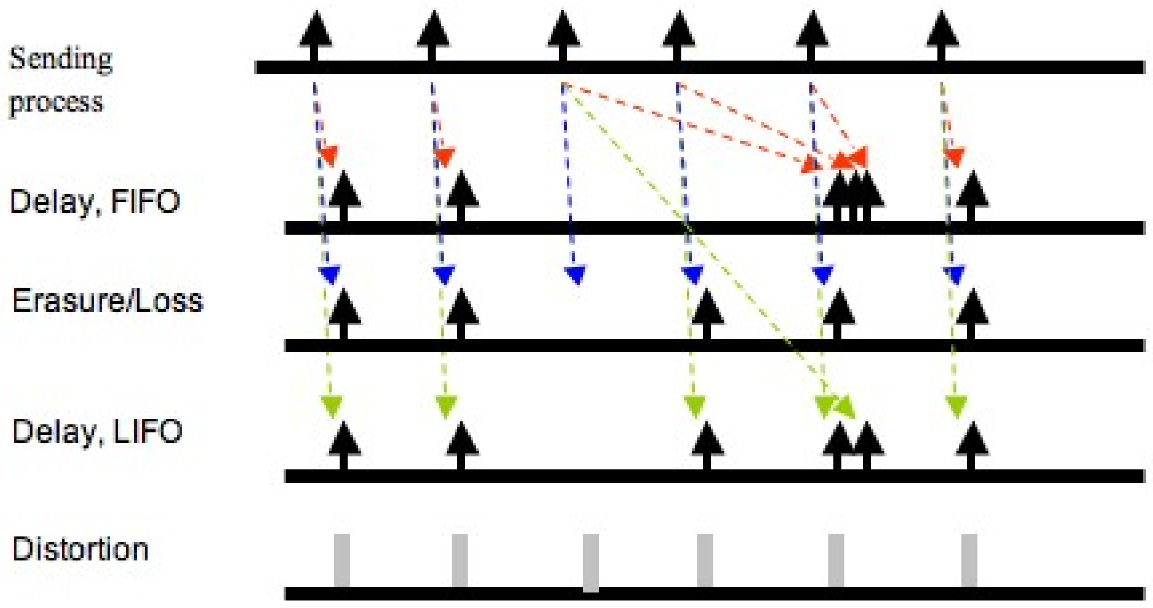


Figure 1.20: Message reception processes in NNRTS

moving progressively smaller super-components, if the coarser representations are not schedulable.

7.2 NNRTS

In this subsection, we discuss networked real-time systems in which inter-component dataflows experience frequent jitter, or loss. We emphasize jitter because the problems arise because of the variability of delay. DNRTS also have delay but the delay is predictable. When controllers communicate over the Internet or local wireless channels, there are no techniques to eliminate both losses and jitter. Essentially, in such networks, one must choose between loss and jitter. One cannot eliminate both. Protocols preventing loss do so by re-transmission which creates variable delay or jitter.

The situation is illustrated by Figure 1.20, which shows the different kinds of NNRTS channels encountered in practice. A controller may output a periodic message stream. However, the corresponding message process observed at the input port of the receiving controller may be asynchronous. If the messages arrive through a TCP channel they may experience different delays though all messages will be received in the right order. The TCP channel is FIFO. Alternatively if the messages arrive through a UDP channel in a one-hop wireless LAN, some may be lost while those that arrive do so with delays small enough to be abstracted away. This combination of loss and negligible delay has received wide attention in the information theory literature, where it is referred to as the erasure channel. The fourth line in the figure illustrates the Last In First Out (LIFO) channel with delays. When messages report sensor measurements or the actuator commands of other controllers, the most recent values are generally more useful than earlier ones. Therefore, it makes sense to have the network give priority to delivery of the information generated most recently instead of enforcing a first in first out order. Finally, if communication were analog, one might assume a distorted version of the transmitted signal is always received without loss or delay. While this formulation is of considerable theoretical interest [28, 35], we will not discuss it further because control with analog communication is so uncommon in current engineering practice.

When inter-component dataflow is jittery even the logical aspect of the semantics assumed by digital control cannot be preserved in the manner used for DNRTS. Suppose we were controlling one truck to follow another at some safe distance as illustrated in Figure 1.21. The control commands of the following truck would be functions of quantities like the position, velocity, or acceleration of the truck in front (see [34]), information we assume flows over a network. If we were to compile and distribute the program as prescribed for DNRTS, the module controlling the following vehicle would block until every message from the leading

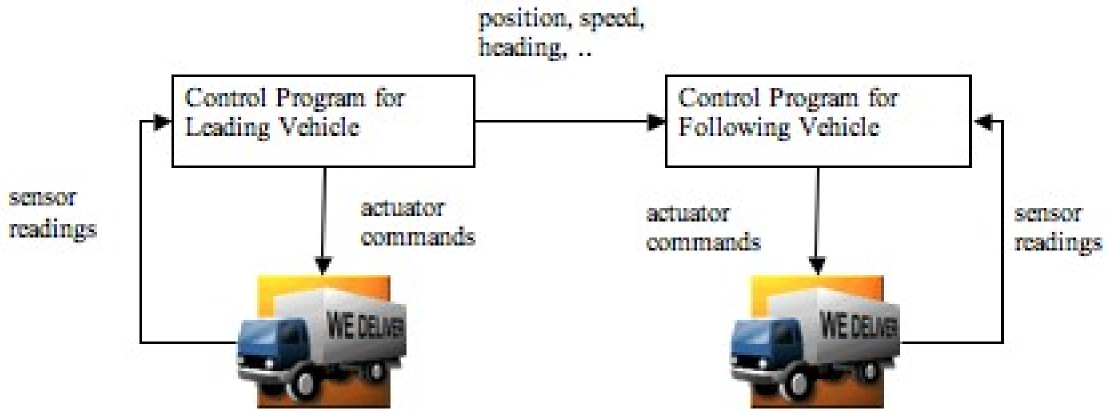


Figure 1.21: Two-vehicle control system schematic

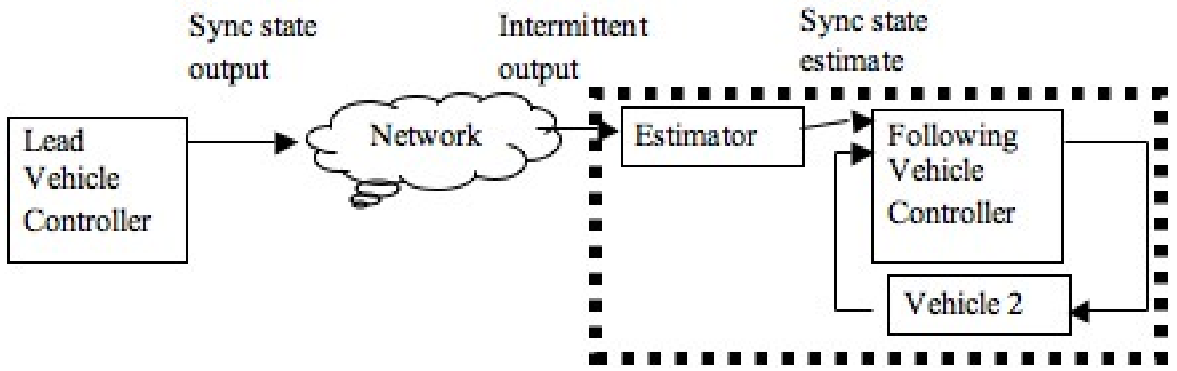


Figure 1.22: Controlling in the face of network losses

vehicle module is received. Since the blocking time would be as jittery as the access delay in Figure 1.17, the following vehicle would issue actuator commands at irregular intervals. Blocking real-time controllers for unpredictable durations is usually unacceptable.

Having recognized the impact of the loss or delay on control performance, control theory has moved away from the perfect inter-controller channel abstraction discussed in Section 2. It has introduced the physicality of inter-controller links into its semantics and produced design techniques to account for the losses or delays. As yet, it remains unclear how real-time programming might produce tools to facilitate embedded programming in these new semantics. Engineers work at a low level with C, UDP or TCP sockets, and OS system calls, rather than at the high level envisaged in the Simulink to Lustre to TTA chain described for DNRTS.

A new semantics for NNRTS is illustrated by Figure 1.22. Developments in control theory show performance is considerably enhanced by abandoning the simplified inter-controller connection abstraction in Figure 1.21 for the more sophisticated scheme in Figure 1.22 when there is a non-deterministic inter-component dataflow. The three-vehicular system examples we have discussed all use this scheme. One puts a component at the receiver end that receives an intermittent asynchronous message stream from the UDP socket at its input but produces a periodic synchronous signal at its output. At the most basic level, this component is as an asynchronous-to-synchronous converter. It enables the entire system downstream, for example, the warning or control systems, to be designed in the classical way, e.g., as a digital control system computing, in this case, at 50 Hz.

In Figure 1.22, we refer to this converter component as an *Estimator*. This is because of control theory showing it is wise to give it semantics richer than that of just an asynchronous-to-synchronous converter.

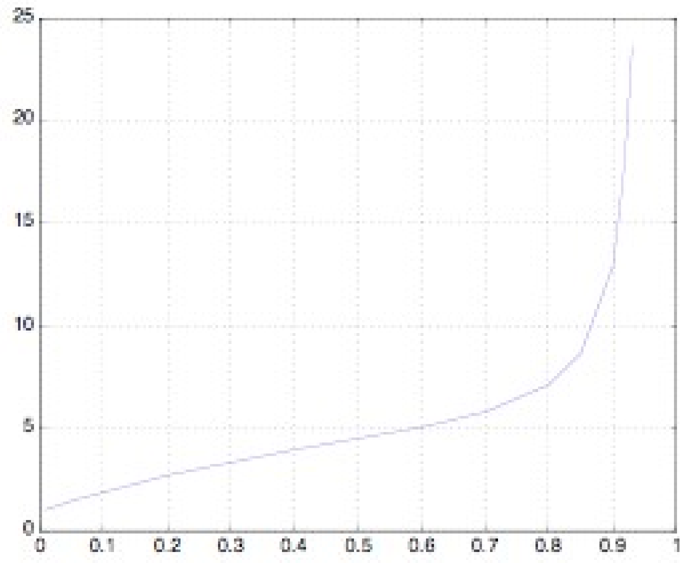


Figure 1.23: Performance obtained by using an estimator

For example, the component output may compute or approximate, at each time, the expected value of the message that might be received over the channel at the time conditioned on all past received values. In many cases, this is a type of Kalman Filter [22, 30, 33], aptly named by Sinopoli et.al. as a "Kalman Filter with Intermittent Observations". Since messages in these systems report on the state of dynamical systems, and the state is correlated, this is a viable approach. The purpose of this converter is to help the receiver deal with the channel distorting the output of the sender.

Figure 1.23 [30] is a quantification of the performance robustness obtained by using the scheme in Figure 1.22. It is computed for one car following another. The vehicle model is a linear approximation of the demonstration vehicles. The objective of control is to maintain constant spacing between the cars. The spacing deviates from the desired constant value because of acceleration or deceleration by the leading vehicle. The following vehicle takes some time to speed up or slow down which leads to spacing errors. The vertical axis in the figure plots:

$$\frac{spacing\ error}{leading\ vehicle\ acceleration}$$

One would like this ratio to be small. The horizontal axis plots the loss probability in the inter-controller communication channel. It is assumed the loss process is Bernoulli, i.e, each packet is lost with probability p or received with probability $1 - p$. The figure shows that as p gets larger the gain gets larger, i.e., the performance deteriorates. The control system computes an estimate of what might have been received whenever a message is lost and uses it to compute the actuator command. The value of doing this appears in the shape of the blue line in the figure. The performance remains good for a long time and breaks down only at high loss rates.

Control is developing effective methods for NNRTS by breaking away from its established computing abstractions as presented in Section 2. The new abstractions are still debated and will take time to settle. Nevertheless, as the class of control systems distributed over the Internet or wireless LANs is growing, the real-time programming community should accelerate its effort to produce a tool chain for the clean and correct real-time programming of NNRTS.

References

[1] A. Benvenieste A., B. Caillaud, and P. Le Guernic. Compositionality in dataflow synchronous languages: specification and distributed code generation. *Information and Computation*, 163(1):125–71, 2000.

[2] A. Benveniste, P. Caspi, P. Guernic, H. Marchand, J. Talpin, and S. Tripakis. A protocol for loosely time-triggered architectures. In *Proc. International Workshop on Embedded Software (EMSOFT)*, volume 2211 of *LNCS*. Springer, 2001.

[3] G. Berry. *The Esterel Language Primer, version v5 91.* http://www.esterel-technologies.com/technology/scientific-papers/, 1991.

[4] 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.

[5] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages.* Addison Wesley, 2001.

[6] G. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications.* Springer, 2005.

[7] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From Simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications. In *Proc. ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 153–162. ACM Press, 2003.

[8] M. Ergen, D. Lee, , R. Sengupta, and P. Varaiya. Wireless token ring protocol. *IEEE transactions on Vehicular Technology*, 53(6):1863–1881, November 2004.

[9] E. Farcas, C. Farcas, W. Pree, and J. Templ. Transparent distribution of real-time components based on logical execution time. In *Proc. ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 31–39. ACM Press, 2005.

[10] L. Feng-Li, J. Moyne, and D. Tilbury. Network design consideration for distributed control systems. *IEEE Transactions on Control Systems Technology*, 10(2):297–307, March 2002.

[11] A. Ghosal, T.A. Henzinger, D. Iercan, C.M. Kirsch, and A.L. Sangiovanni-Vincentelli. A hierarchical coordination language for interacting real-time tasks. In *Proc. ACM International Conference on Embedded Software (EMSOFT)*. ACM Press, 2006.

[12] A. Ghosal, T.A. Henzinger, C.M. Kirsch, and M.A.A. Sanvido. Event-driven programming with logical execution times. In *Proc. International Workshop on Hybrid Systems: Computation and Control (HSCC)*, volume 2993 of *LNCS*, pages 357–371. Springer, 2004.

[13] N. Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer, 1993.

[14] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9), 1991.

[15] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE*, 91(7), July 2003.

[16] 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.

[17] T.A. Henzinger, C.M. Kirsch, and B. Horowitz. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, January 2003.

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

[19] T.A. Henzinger, C.M. Kirsch, and S. Matic. Schedule-carrying code. In *Proc. International Conference on Embedded Software (EMSOFT)*, volume 2855 of *LNCS*, pages 241–256. Springer, 2003.

[20] T.A. Henzinger, C.M. Kirsch, and S. Matic. Composable code generation for distributed Giotto. In *Proc. ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. ACM Press, 2005.

[21] T.A. Henzinger, C.M. Kirsch, M.A.A. Sanvido, and W. Pree. From control models to real-time code using Giotto. *IEEE Control Systems Magazine*, February 2003.

[22] R. Kalman. A new approach to liner filtering and prediction problems. *Transactions of the ASME, Journal of Basic Engineering*, 82D:34–45, March 1960.

[23] K. H. Kim. A non-blocking buffer mechanism for real-time event message communication. *Real-Time Systems*, 32(3):197–211, 2006.

[24] C.M. Kirsch. Principles of real-time programming. In *Proc. International Workshop on Embedded Software (EMSOFT)*, volume 2491 of *LNCS*, pages 61–75. Springer, 2002.

[25] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer, 1997.

[26] H. Kopetz and J. Reisinger. NBW: A non-blocking write protocol for task communication in real-time systems. In *Proc. IEEE Real-Time Systems Symposium (RTSS)*, 1993.

[27] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1), 1973.

[28] A. Sahai. *Anytime Information Theory*. PhD thesis, Massachusetts Institute of Technology, 2001.

[29] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi. Defining and translating a ”safe” subset of Simulink/Stateflow into Lustre. In *Proc. ACM International Conference on Embedded Software (EMSOFT)*, pages 259–268. ACM Press, 2004.

[30] P. Seiler and R. Sengupta. An h-infinity approach to networked control. *IEEE Transactions on Automatic Control*, 50(3):356–364, 2005.

[31] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[32] I. Shin and I. Lee. Compositional real-time scheduling framework. In *Proc. IEEE Real-Time Systems Symposium (RTSS)*, pages 57–67, 2004.

[33] B. Sinopoli, L. Schenato, M. Franceschetti, K. Poolla, M.I. Jordan, and S.S. Sastry. Kalman filtering with intermittent observations. *IEEE Transactions on Automatic Control*, 49(9):1453–1464, September 2004.

[34] D. Swaroop. *String Stability of Interconnected Systems: An Application to Platooning in Automated Highway Systems*. PhD thesis, University of California at Berkeley, 1994.

[35] S. Tatikonda. *Control Under Communication Constraints*. PhD thesis, Massachusetts Institute of Technology, 2000.

[36] J. Templ. TDL specification and report. Technical Report T004, Department of Computer Sciences, University of Salzburg, November 2004.

[37] S. Tripakis, C. Sofronis, N. Scaife, and P. Caspi. Semantics-preserving and memory-efficient implementation of inter-task communication on static-priority or edf schedulers. In *Proc. ACM International Conference on Embedded Software (EMSOFT)*, pages 353–360. ACM Press, 2005.

[38] M. Zennaro and R. Sengupta. Distributing synchronous programs using bounded queues. In *Proc. ACM International Conference on Embedded Software (EMSOFT)*. ACM Press, 2005.