

Temporal Control in Real-time Systems: Languages and Systems

Sebastian Fischmeister and Insup Lee

Department of Computer and Information Science

University of Pennsylvania

sfischme@seas.upenn.edu, lee@cis.upenn.edu

Abstract

Temporal control is an important concept for real-time systems, and several systems and languages provide means for setting temporal constraints on the program. However, the way temporal constraints are placed on the program, the granularity, and the supported kinds of temporal constraints differ from system to system. This chapter provides an introduction to temporal control in real-time systems on the basis of temporal constraints. First, it introduces the notion and nature of a temporal constraint and presents a generic model of the task life cycle. Second, it discusses six different systems and languages with temporal control by adapting the generic model to the individual system and by presenting a short example of a stopwatch task. Finally and third, a summary and discussion of the differences among these systems conclude the chapter.

1 Introduction

Real-time application cannot only rely on predictable output values, real-time applications must also have predictable timing tied to these outputs. Non-real-time applications, such as a spread-sheet program or a word processor, are insensitive to minor variances in timing but sensitive to performance. For such applications, it is acceptable that operations sometimes complete a little earlier or later as long as the overall

performance is sufficiently fast. In real-time applications, such unbounded delays are unacceptable. Failure in the application's timing predictability is as unwanted by the stakeholders as failure in the application's output predictability. So, for example, when the application returns a correct result too late it could be as bad as if it would return a wrong result. Consider an electronic braking system built into a car where we replace the standard braking system with a completely electronic one. In this system is unacceptable for the driver if there is an unpredictable, sometimes even very long, delay between the driver hitting the brake pedal and the brakes applying force to the tires. This scenario makes it essential to think about temporal constraints in real-time applications.

Temporal constraints, placed on actions of the real-time application, specify the timing behavior of the application. Such constraints are often formulated as a range of possible time values or specified as upper bounds to the occurrence time of specific events. To ensure that these temporal constraints are met, the developer may need to analyze a chain of operations and specify the latest time (i.e., deadline) when each operation must complete. In the braking example a temporal constraint on the braking action could be that the brakes have to apply force to the tires within one millisecond once the driver has hit the brake pedal. To guarantee that this temporal constraint is met, the developer will have to analyse and sum up the reaction time of the brake-pedal level detection, the transmission time of the signal to the brake controllers, and the reaction time of the disc brakes mounted on the tires. If worst case of this elapsed time is lower than the temporal constraint, the constraint is met.

Real-time computing and temporal constraints are not about performance but about timing precision. The range over which and the unit in which temporal constraints are specified is insignificant; it is the precision of the predictability that matters. Some real-time applications define temporal constraints with bounds in nanoseconds and some define constraints in seconds. For example, while the electronic braking system might require a temporal constraint of a millisecond, a patient ventilation machine, which pumps oxygen-enriched air into a patient's lungs during a surgery, has a much slower temporal constraint. The machine must ventilate the patient at a frequency of twelve to fourteen times per minute with five to six ml/kg breathing volume. Here, the temporal constraint between two pump operations is roughly five seconds and it is relatively slowly compared to the braking system.

Temporal constraints may or may not be met, however a specific class of applications, called in safety-critical real-time applications, requires the temporal constraint must to be always met. Such safety-critical real-time applications are systems whose failure or malfunction may lead to human losses. Since temporal behavior is critical, such applications have to be certified to always meet the temporal constraints before they can start operating. Two examples of safety-critical real-time applications have already been discussed: (1) if the braking system fails to meet the timing requirements, then a car crash and casualties may be the consequence, (2) if the ventilation machine fails to meet its timing requirements, then the patient might suffocate and die. Other examples are typically found in the domain of avionics, transportation, and medicine, such as air traffic control, drive-by-wire, fly-by-wire, and infusion pumps.

Temporal constraints exist in the forms of soft temporal constraints, hard temporal constraints, and firm temporal constraints. *Soft temporal constraints* are found in soft real-time systems. A soft real-time system is one in which the response time (i.e., a temporal constrain) is specified as an average value and the system does a best effort approach to meet these requirements. A soft temporal constraint specifies an average bound between two events, and a single violation of this constraint does not significantly matter, whereas many violations and high value deviations do. An example system with soft constraints is an airline reservation system. It is irrelevant if the reservation sometimes takes a little more time to book a reservation, as long as the booking operation's execution time is bounded reasonably. *Hard temporal constraints* are found in mission-critical real-time systems. A hard real-time system is a system where the response time is specified as an absolute value that is typically dictated by the environment. A hard temporal constraint specifies an absolute upper bound between two events. A single violation of such a constraint does matter as hard real-time applications are often safety-critical applications. An example of such a system is the electronic braking system mentioned above; if the braking system does not meet the constraint just once, this might have been the critical instant where meeting the constraint could have saved lives. *Firm temporal constraints* are found in firm real-time systems. Each constraint has a soft and hard part consisting of an average bound (soft) and an absolute bound (hard), respectively. An example is the mentioned patient's ventilation system. Specifying the bound between two ventilation cycles, this timing constraint has a soft constraint and a hard constraint. The soft constraint is fourteen ventilations per minute, and the hard constraint is twelve

ventilations.

Temporal constraints can be absolute or relative. *Absolute temporal constraints* are measured with respect to a global clock, usually our wall clock. An example of such an absolute temporal constraint is that the electronic Christmas tree should light up between 5pm and 2am each day from November 24th until December 27th. *Relative temporal constraints* are measures with respect to the occurrence times of specific events on the local clock. An example of such a relative temporal constraint is that the ventilation task should be activated four seconds after the last ventilation ended. Given an absolute or a relative temporal constraint, if the constraint does not hold, then a *timing violation* has happened. A timing violation occurs, for example, when the ventilation task is not rereleased within four seconds after the last ventilation ended, or if we specify that a computation should finish within one second and it does not.

A specific form of temporal control is time determinism. *Time determinism* defines that an external observer can predict the points in time, at which an application produces output (can also be called time/-value determinism). This means that the observer can treat the system as a black box and predict correct values without knowing the application's internals. The points in time specifying the output times can be given by a time range or by exactly one point in time. The smaller the range, the more precise the temporal control must be. For example, in the patient ventilation system, it is easier to predict the time range of the ventilation impulse to be $[0, 5)$ than to predict a single number such as the impulse will occur exactly every $4.7s \pm 0ns$.

2 The Model

Temporal constraints always refer to events, where an event denotes the beginning or ending of a significant activity and is considered instantaneous. Temporal constraints can be used to specify the time bound between any two events. For example, a temporal constraint can specify the delay between the 'driver-hits-the-pedal' event and the 'brakes-apply-force-to-tires' event. For real-time tasks, common events to which temporal constraints are assigned are the task's release or its completion event. A temporal constraint, bounding the time between two user visible events, is often called an *end-to-end constraint* or an *end-to-end delay*. We now describe the life-cycle model of a task (c.f., Figure 1.1), identify the most common events, and show

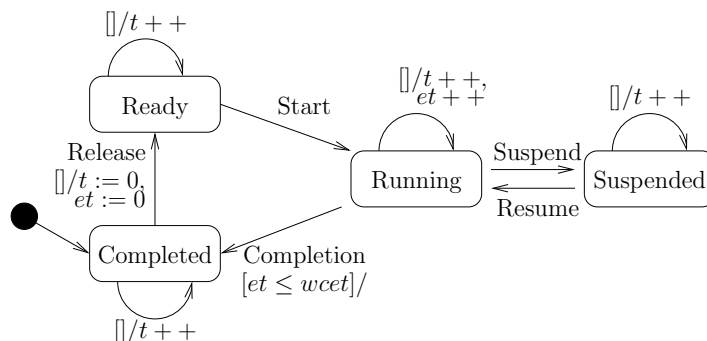


Figure 1.1: Task life cycle.

how they are used to specify temporal constraints.

Figure 1.1 provides a state diagram of a task’s life cycle in our model. We follow the traditional approach and describe the model with tasks. However, note that the model is also applicable at the level of functions, single statements, and transactions. For example, temporal scopes provide such a model at the statement level (c.f., Section 5.2). In our notation, the rounded box with a label in it denotes a state (e.g., the state *Ready*). Arrows between states define transitions. A transition can be labeled, it can have an enabling condition, and taking the transition can lead to updates of variables. The format is as follows: *label* [*condition*]/*updates*. For example, the transition between the state *Completed* and *Ready* in Figure 1.1 specifies that it is called *Release*. This transition has no enabling condition, and when it is taken, then the variable *t* is set to zero. The filled circle points to the initial state. As a shorthand notation, if a transition has an empty enabling condition (i.e., it is always enabled) and has an empty update clause (i.e., it does not update any variable) the empty construct $[]/$ is omitted. In Figure 1.1, the task starts its life cycle in the state *Completed*. When it must accomplish its action, the scheduler releases it, and the task enters the state *Ready*. Once the scheduler assigns computing resources to the task, the task enters the state *Running*. However, the scheduler can suspend the task and can change its state to *Suspended*. Then, the scheduler can resume the task and it changes back to state *Running*. Some systems also allow the developer to actively suspend and resume tasks via program instructions. Once the task has completed its activity, it stops using computing resources and changes to state *Completed*. The task resides in this state until it is released again. Note that there may be other events for the operating-system tasks; however, they are not of particular interest here.

This life-cycle model describes a number of events:

- **Release Event.** It marks the time at which the task enters the system and the scheduler adds it to its ready queue.
- **Start Event.** It marks the time at which the task starts computing for the first time since its release event.
- **Suspend Event.** It marks the time at which the scheduler removes the task's access to computing resources. The suspended task cannot continue computing.
- **Resume Event.** It marks the time at which the scheduler gives the task access to computing resources again so that the task can resume its activity.
- **Completion Event.** It marks the time at which the task finished its activity and indicates that the task no longer requires computing resources.

In addition to the events described in the model, the task and the environment can trigger application-specific events. Figure 1.1 does not show these events, because they differ for each application. For example, the task can sense an emergency shutdown button. If someone pushes this button, the event 'button-pushed' will be created in the environment. The task can sense this event and respond to it by firing an application-specific event such as 'perform-emergency-shutdown'.

To express timing constraints, the state diagram in Figure 1.1 includes the variables t and et . The variable t measures the elapsed time since the release event. The variable et measures the execution time that the task has consumed computing resources. The constant $wcet$ defines the task's worst-case execution time, i.e., the maximum amount of time that the task needs to consume computing resources to complete its activity. Note that the notation used in Figure 1.1 is rather limited in its expressiveness. We could have used other formalisms, such as timed automata, real-time process algebra, and timed Petri-net. The examples, however, would be unnecessarily complicated using them, so we use the model shown in Figure 1.1 through this paper and adapt it to the capabilities of the programming language under discussion.

Temporal constraints can be specified in different forms and notations, but essentially they define a bounded time between two events. A timing constraint can be tied to a one-time event in the lifetime of the

system or to recurring events. An instance of a one-time event is the emergency shutdown described above. A temporal constraint can define a time bound between the event ‘button-pushed’ and ‘perform-emergency-shutdown’. Since the system is off, after it has performed the shut down, we consider this as a one-time event. Recurring events in our model are caused by repetitive computational activities getting rereleased once they completed.

Real-time theory distinguishes three different recurrence types: periodic, sporadic, and aperiodic. A *periodic recurrence* produces release events at regular time intervals. It is possible to define a temporal constraint between two release events as time units. This value is also called the task’s *period*. Periodic recurrences can be specified in different notations. In scheduling theory, they are usually specified by the period P where P time units pass between two occurrences. For example, the patient ventilation pump has a period $P = 5s$. Another way to specify a temporal constraint is to define a period P' and a frequency f . For the same behavior, one can specify that the ventilation event has a period $P' = 60s$ and a frequency $f = 12$. Yet another way is to specify just the frequency in Hertz f' ; the pump has a frequency $f' = \frac{1}{5}Hz$. Note that all these notations are compatible and one notation can be converted into the other. A *sporadic recurrence* produces release events with irregular but bounded time intervals. It is possible to specify a temporal constraint between two such events as an interval with a lower limit and an upper limit. For example, using the patient ventilation pump as a firm real-time system, the temporal constraint between two release events is bounded by $[4.28s, 5s)$. This means that the elapsed time between two release events must be more than 4.28 seconds and less than five seconds. Finally, an *aperiodic recurrence* produces irregular release events and no meaningful temporal constraint can be placed on them.

3 The Example: A Stopwatch

We use the example of a software-controlled digital stopwatch to explain the different programming languages and their support for temporal control throughout this chapter. The stopwatch is appropriate, because it requires tight control of timing and needs virtually no program logic. The stopwatch has been used commonly in the literature as a guiding example to demonstrate different aspects of value and timing behavior in real-time applications (c.f., [1, 2, 3]).

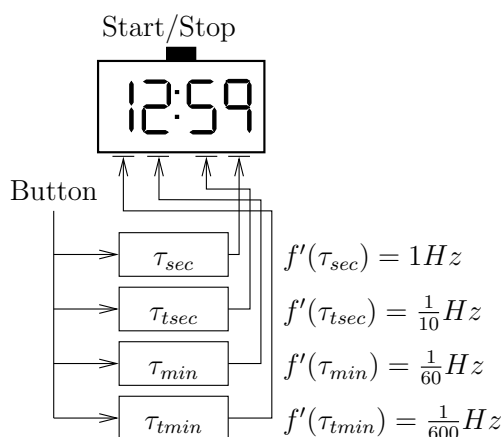


Figure 1.2: The software-controlled digital stopwatch.

The goal is to implement a software-controlled digital stopwatch with one start/stop button. The stopwatch is to behave as follows. When the user presses the start/stop button, the clock value is reset and it starts running. When the user presses the start/stop button again, the clock stops running. At any time, the stopwatch's display shows the clock's value. Additionally, the clock should exhibit predictable value update behavior; so we require the common update sequence of starting from the one column and continuing to the hour column (e.g., 12:59 to 12:50 to 12:00 to 13:00).

Figure 1.2 shows an overview of the application. It consists of four tasks and each controls an individual digit of the stopwatch's display. Task τ_{sec} controls the seconds' ones column and runs at a frequency of $1Hz$. Task τ_{tsec} controls the seconds' tens column and runs at a frequency of $\frac{1}{10}Hz$. Task τ_{min} controls the minutes' ones column and runs at a frequency of $\frac{1}{60}Hz$. And finally, Task τ_{tmin} controls the minutes' tens column and runs at a frequency of $\frac{1}{600}Hz$. Each task receives the start/stop button's status as input. The functionality of the tasks τ_{sec} and τ_{min} is to output *invocations mod 10*. The value of variable *invocations* is the number of times each task has been released. Since the one column's value range of seconds and minutes is between zero and nine, the function calculates the number of invocations modulo ten. The functionality of the tasks τ_{tsec} and τ_{tmin} is similar, these two tasks output *invocations mod 6*. Note that the intent of this design is to demonstrate the differences among the programming languages and systems reviewed in this chapter.

The difficulty in this clock's design is that it requires rigorous control of the tasks' timing and the value

behavior. Figure 1.2 shows the stopwatch 779 seconds after pushing the start/stop button. It shows the value of twelve minutes and fifty-nine seconds. So the tasks' outputs are: τ_{sec} outputs the value 9, τ_{tsec} outputs the value 5, τ_{min} outputs the value 2, and task τ_{tmin} outputs the value 1. We now highlight two potential problems, which have to be addressed when implementing this design:

- **Jitter accumulates and causes incorrect display.** If the programming language and the runtime system introduce jitter for each task invocation, then in some systems this jitter will accumulate over time and the display will eventually show incorrect values. Additionally, if each task manages its release time itself, then the stopwatch can behave incorrectly. For example, suppose that the task τ_{tsec} has updated its value from five to zero, but the task τ_{min} is late and has not been released yet. Here, the display value might change from 12:59 to 12:09.
- **Different causal ordering of events and tasks causes different behavior.** The four tasks run concurrently. Every ten minutes, all four of them are released. If the computer has only one processor, then the tasks can only complete sequentially. If the runtime system does not provide means for controlling the ordering or the outputs' update behavior, then the display might disobey the specified update behavior. For example, if the order of the completion events is τ_{sec} , τ_{min} , and τ_{tsec} , then the display values will change in this order: starting with 12:59 to 12:50 to 13:50 to 13:00. The typical order is to update from the rightmost column to the leftmost column: starting with 12:59 to 12:50 to 12:00 to 13:00.

Another problem is to implement precise semantics for the effect of pushing the button. For example, do we start measuring the time after the start/stop button has been pressed or at the time it is pressed. We will elaborate on this problem when we discuss the synchronous languages, which target exactly this problem.

4 Implicit Temporal Control

Small systems often lack explicit temporal control and thus temporal control is programmed using hand-coded delay blocks. We call such means *implicit temporal control*, because it is embedded in the program

source and does not use high-level language constructs. A widespread method is the background/foreground systems. In a *background/foreground system*, the application consists of exactly one loop without an exit condition. Within this loop, the application calls subroutines in a sequential order, which implements the application's logic. The loop's execution time essentially determines the application's temporal behavior. The loop is commonly referred to as the background. If an interrupt occurs, an interrupt service routine (ISR) preempts (suspends) the loop and services the interrupt. The ISR is commonly referred to as foreground; hence the name background/foreground system.

Figure 1.3 shows the foreground/background system's life-cycle without nested interrupts. The application typically spends time in the background part, executing the main loop. When an interrupt occurs, the system switches to the foreground and the ISR services the interrupt. Once the ISR completes, the system switches back to the background operation and resumes the main loop.

The main application domain of background/foreground systems is small embedded systems such as washers, dryers, microwave ovens, and simple radios. Compared to multi threaded systems with explicit temporal control, background/foreground systems require less system overhead and less understanding of concurrency and temporal control. However, the low system overhead has a price. The application's output is non-deterministic with respect to the timing. The points in time at which the application produces an output changes depending on the application's execution path for each run in the loop and how many and what types of interrupts occur. The application's timing is also sensitive to modifications to the loop. For example, one additional inner loop in the main loop changes the timing behavior of everything that follows after this inner loop. Such a change can alter the whole system's behavior. Also, modifying the ISR changes the timing behavior depending on how often the ISR preempts the main loop.

In this system, the stopwatch example cannot be implemented with multiple tasks, because a background/foreground system does not provide multi tasking. So, we have to resort to the means of the superloop to implement it. Listing 1 gives an idea on how the stopwatch can be implemented. The superloop starts at Line 5, and it updates one column of the watch's display after the other: First the seconds, then the tens of seconds, then minutes, and finally tens of minutes. At the end of outer-loop, we place a tight loop to wait for the next second. Line 16 assigns to i the value that results in waiting for precisely one

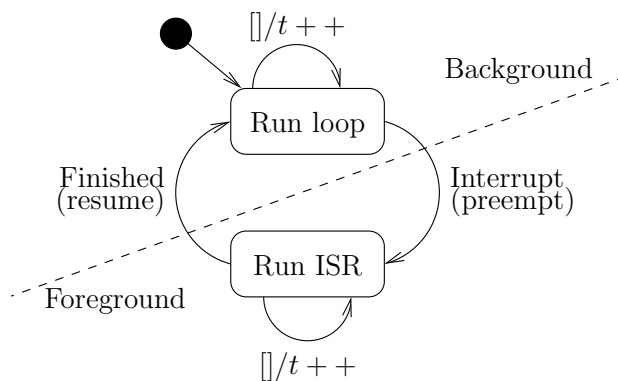


Figure 1.3: Foreground/background system's life-cycle.

second in Line 17. The faster the processor, the higher will be this number. If the value range of i is too small, then nested loops can be used. After the tight loop at Line 17 terminates, the superloop will start over again. The temporal behavior is controlled via conditional branches (c.f., Line 10 that control how often specific function blocks are executed. For example if the loop iterates once every second and the conditional branch specifies that it is executed only every sixtieth time, then the function blocks inside the branch will be executed once every minute.

```

1 void main(void) {
    unsigned short val;
    unsigned int i;

    while ( 1 ) {
6     val = get_curr_sec();
        val++;
        update_display_seconds(val);

        if (val%60 == 0) {
11         // update tens
        }
        ...

        // may have nested loops, if too short
16     i=WHILE_INSTRUCTIONS_PER_SECOND;
        while( --i );
    }
}

```

Listing 1: The task τ_{sec} with the superloop.

This example implements the desired update behavior; however, this implementation is prone to jitter and the displayed time is unequal to the measured time. The reason is that over time the displayed time diverts from the real time, because of the variance in the super loop's execution time. Even if we assume that the constant in Line 16 takes the execution time into consideration, every tenth invocation, when the tens of seconds column gets updated, the run time differs from the other nine invocations. The same happens every 60th and 3600th invocation. Over time, jitter accumulates and results in an incorrectly measured time. Note that a stop watch can be implemented with precise timing using a background/foreground system; however, it requires a more elaborate setup with interrupts and system timers. Real-time programming languages and systems provide means for this right in the language or the system.

5 Programming with Temporal Control

Different programming languages and systems allow specifying temporal constraints in different ways, as some systems support specification-based temporal constraints and others support program-based ones. *Specification-based* temporal constraints are defined by generating temporal behavior from annotated source code or some high-level specification. Specification-based approaches require additional support from the compiler, because the specification has to be translated into code additionally to the functionality code. For example, assume that the ventilation task is responsible for generating the patient ventilation impulse and it executes once every five seconds. In a specification-based approach, the developer annotates the task with something similar to “`frq 0.2`”, and the compiler translates this annotation and generates code, which executes the ventilation task with the specified frequency. *Program-based* temporal constraints are programmed in the target programming language without annotations. This approach is quite similar to the implicit temporal constraints shown before, however, in program-based temporal constraints the programming language usually provides an interface for programming temporal constraints; so, no `while(--i);` is necessary. Some programming languages provide more sophisticated support than others. The least common denominator of

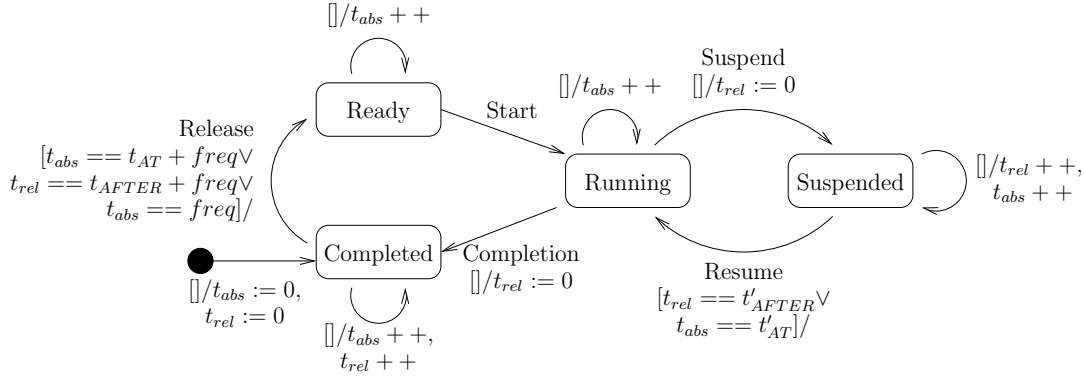


Figure 1.4: Task life cycle in PEARL without the event-triggered parts.

these interfaces is a function to read the wall-clock time and a function to delay execution for a specified amount of time. To program the ventilation task, the developer executes the task in an endless loop. At the loop's end, the task enters a statement corresponding to $delay(5s - \epsilon)$, where ϵ compensates for the execution time of the ventilation-control function. The ventilation task will then run at least once every five seconds.

In the following sections, we present a number of programming languages and systems, which provide built-in means of temporal control either on the specification or the programming level.

5.1 PEARL

The programming language PEARL, an acronym for *Process and Experiment Automation Real-time Language*, is a high-level programming language with elaborate constructs for programming temporal constraints. The language itself has been standardized as DIN standard 66253 [4]. PEARL is tailored to implement time sensitive systems, and it includes packages for input/output, multitasking, synchronization, interrupt handling, and signaling. For this chapter, we concentrate on the support for specifying temporal constraints on tasks. In this chapter, we use the revision PEARL-90 [5]. PEARL-90 is a reworked version of the original PEARL language, and in the remainder of the text, we always refer to PEARL-90 when we write PEARL.

5.1.1 The Model

PEARL provides temporal control for releasing and resuming tasks. Figure 1.4 shows the time-triggered parts of the task's life cycle. PEARL allows the developer to also specify event-triggered constraints on the life

cycle as well; however, they are unrelated to timing and thus they are not discussed in this paper. Each task starts by entering the state *Completed*. PEARL allows specifying absolute and relative temporal constraints. The variable t_{abs} realizes an absolute timer that is reset whenever the system is started. The variable t_{rel} realizes a timer for relative temporal constraints. PEARL allows placing absolute and relative constraints on the task's release. So, for the task to be released, the constraint guarding the transition from the state *Completed* to the state *Ready* must be satisfied. The variable t_{AT} represents the timing information specified with PEARL's **AT** statement, which is used to declare absolute temporal constraints such as **AT 12:00**. The variable t_{AFTER} represents the timing information specified with PEARL's **AFTER** statement, which is used in relative temporal constraints such as **AFTER 2 sec**. PEARL also supports specifying task periods shown as *freq* in Figure 1.4. The frequency determines at what time the task is released again. The task's recurrence can also be limited to a relative or absolute time span. So, for example, the developer can specify that a task runs every five seconds for the next ten minutes or between 1pm and 2pm. Once the task is released, it enters the state *Ready* and eventually the state *Running*. A task may be suspended and it enters the state *Suspended*. PEARL allows the developer to place absolute and relative temporal constraints on the transition back to the state *Running*. Note that t_{AFTER} on transition *Release* and t'_{AFTER} on transition *Resume* differ. t_{AFTER} is specified for the release event and t'_{AFTER} is specified for the resume event. The same is true for t_{AT} . For example, the developer can program that a task is to be continued at 1pm (after the lunch break), or after ten minutes measured from the task's suspension to realize a cool-down phase. Eventually, the task will complete and enter the state *Completed*.

5.1.2 The Stopwatch

Listing 2 shows part of the complete PEARL program that implements the stop watch. For details on the syntax see [4]. Specifically, the program shows the activation code for all four tasks and the complete source of one of them. The first four lines declare the temporal constraints on the individual tasks. The tasks controlling each clock column are activated depending on how often they must update their values. For example, the task updating the clock's one-second column is activated once every second. The second part of the listing declares and implements task `clock_sec`. The task uses one variable called *ctr*. In the task's

body, we first invoke a method to get the clock's current value of the one-second column. We then increment the value by one and finally write it back to the clock.

PEARL allows the developer to assign priorities to the individual tasks. In Listing 2, task *clock_tsec* has the priority 2 (i.e., the second highest one). The other tasks have different priorities according our intended order of invocation to update the clock display. So when it is 12 o'clock noon, first the task *clock_sec* will update the clock value, followed by *clock_tsec*, *clock_min*, and *clock_tmin*. This way, we implement the update behavior as specified in Section 3.

```

1 WHEN start ALL 1 sec UNTIL stop ACTIVATE clock_sec;
  WHEN start ALL 10 sec UNTIL stop ACTIVATE clock_tsec;
  WHEN start ALL 60 sec UNTIL stop ACTIVATE clock_min;
  WHEN start ALL 600 sec UNTIL stop ACTIVATE clock_tmin;

6 clock_tsec:TASK PRIO 2;
  DCL ctr INTEGER;
  BEGIN
    GET ctr FROM DISPLAY_T_ONES;
    ctr := (ctr+1)%6;
11 PUT ctr TO DISPLAY_T_ONES;
  END

```

Listing 2: The task τ_{tsec} as *clock_ones* in PEARL.

5.2 Temporal Scopes

The notion of temporal scopes has been introduced with the Distributed Programming System (DPS) [6]. A temporal scope is a language construct that can be used to specify the timing constraints of code execution and inter process communication. Such scopes allow the developer to specify timing constraints down to the statement level and register exception handlers to cope with timing violations. Other goals of DPS are that the resulting system should be apt for distributed systems, temporal scopes should be embedded into an existing language, and the system should support run-time monitoring and exception handling. Temporal scopes are a specification-based approach. The developer annotates C source code with temporal scopes and specifies its temporal behavior.

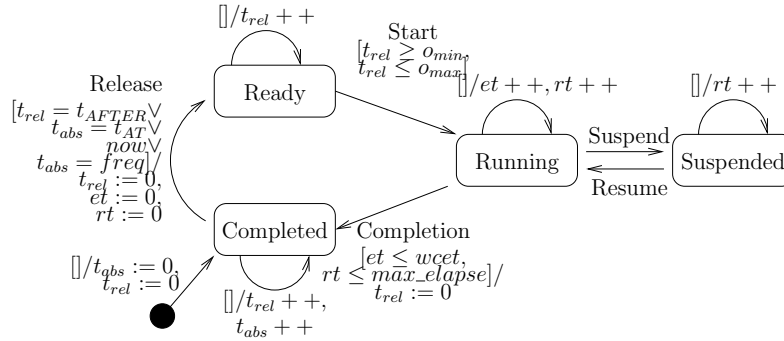


Figure 1.5: A temporal scope's life cycle.

5.2.1 The Scope's Model

Figure 1.5 shows the temporal scope's life cycle. The variables t_{abs} and t_{rel} are the same as used before. The new constraints that are not present in the basic model (c.f., Figure 1.1) are a minimum delay, a maximum delay, a maximum elapse time, and absolute and relative constraints on the release. Temporal scopes allow placing constraints on the task's release similar to PEARL. Additionally, they use the keyword `now`, which means an immediate release. The *minimum delay* (o_{min}) specifies a minimum amount of time that should pass before starting the execution of a temporal scope. So, the condition $t \geq o_{min}$ guards the transition from the state *Ready* to the state *Running* and inhibits a task from starting computing before sufficient time has passed. The *maximum delay* (o_{max}) specifies the maximum amount of time that could pass before starting the temporal scope's execution. As shown in the state diagram, the condition $t \leq o_{max}$ guards the transition from the state *Ready* to the state *Running*. This condition provides an upper bound for the temporal scope until it starts computing. The *maximum elapse time* specifies the maximum amount of time (max_elapse) that the temporal scope requires until completion once it starts computing. The state diagram in Figure 1.5 uses the variable rt to keep track of the amount of time that the task resides in the states *Running* and *Preempted*. The condition $rt \leq max_elapse$ limits the amount of time that the temporal scope can spend in the states *Running* and *Suspended*. In contrast to the execution time, the maximum elapse time considers the duration for which task is suspended. By that, the maximum elapse time combines the execution time and all additional system overhead resulting from suspensions.

5.2.2 The Stopwatch

There are three types of temporal scopes: the sequential scope, the repetitive scope, and the consecutive scope. For the stopwatch example, we will only introduce the repetitive scope. The other types are elaborately described in [6]. Listing 3 specifies the repetitive temporal scope's syntax.

```

from <start-time> to <end-time> every <period> execute <e-time> within <deadline>
do
3  <stmts>
    [<exceptions>]
end

```

Listing 3: The basic temporal scope's syntax.

Here the developer specifies the time interval <start-time> to <end-time> in which the scope will be repetitively executed at every interval specified in <period>. For each repetition, he can also specify its worst-case execution time in <e-time> and the deadline in <deadline>. For the periodic scope, the maximum elapsed time of a period is considered to be equal to its <period>. This fully specifies the temporal behavior as denoted in Figure 1.5. Then, the developer can code the scope's logic (e.g., in C as implemented in DPS) and he can specify exception handlers to handle errors caused by timing-constraint violations.

```

from 00:00 to 59:59 every 10s execute 20ms within 1s
do
    var ctr;
    ctr=get_cur_tsecs();
5  ctr=(ctr+1)%6;
    set_cur_tsecs(ctr);

    exception
        display_warning_light();
10 end

```

Listing 4: The task τ_{tsec} using temporal scopes.

The stopwatch consists of four independent tasks. Listing 4 shows the pseudo-code program for task τ_{tsec} using a repetitive temporal scopes. The scope is defined to run from zero up to one hour. During this time,

it executes the `<stmts>` once every ten seconds and within one second. One execution should take at most twenty milliseconds. In the body of the scope, it reads the current value of the second seconds column and updates it accordingly. If a timing violation occurs, then the program will show a warning light on the stop watch. Similar code can be written for the other three tasks. All four temporal scopes then run concurrently and update the stopwatch's display.

Temporal scopes concentrate on placing temporal constraints on statement blocks. The language leaves inter scope control such as synchronization and prioritization to the programming language using temporal scopes. Therefore, the developer cannot specify an order in which concurrent temporal scopes are executed, and it is impossible to implement the update behavior as specified in Section 3 solely with temporal scopes. For example, at 12 o'clock it is up to the scheduler in what order it executes the tasks and it might choose an order that does not comply with the specification. Also, the task releases are outside the scope of temporal scopes, so the application logic for when the start button or the stop button is pressed has to be specified outside the temporal scope.

5.3 The ARTS kernel

The ARTS kernel [7] allows specifying temporal constraints similar to temporal scopes. The kernel uses the temporal constraints to implement its time-fence protocol that guarantees that the temporal constraints of the active tasks are met. Such a time fence is an annotation to the C++ code containing the functionality and can be associated with functions and threads. The kernel's time-fence protocol can then detect timing-constraint violations as it compares the specified time fence with the actual run-time behavior.

5.3.1 The Model

The ARTS kernel offers a different life-cycle model for threads and functions. In the following text we explain both models by showing how it differs from the basic model in Figure 1.1. The variable t_{rel} implements a relative timer similarly to how we used it in the previous models. Figure 1.6 shows the thread's life cycle. The developer can place temporal constraints on a thread by specifying a phase, an offset, a delay, a worst-case execution time, and a period. The phase is an initial offset that delays the transition from the state *Init*

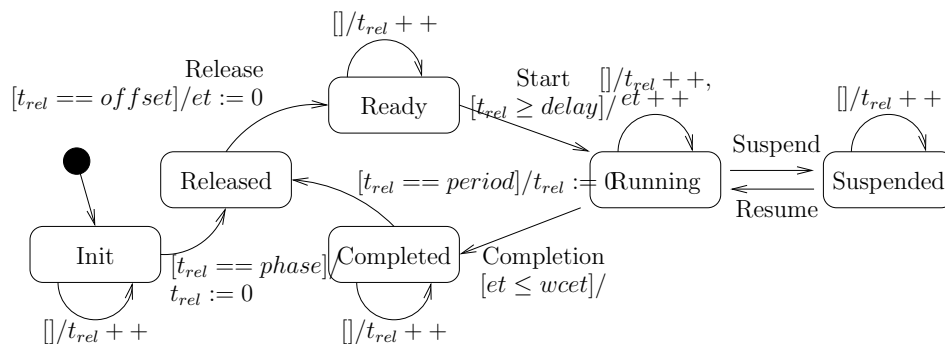


Figure 1.6: Thread life cycle in the ARTS kernel.

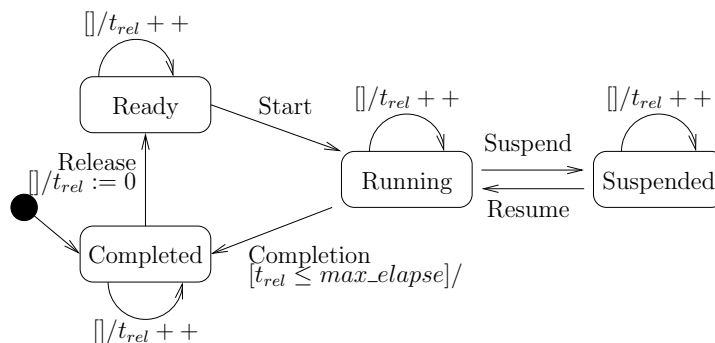


Figure 1.7: Function life cycle in the ARTS kernel.

to the state *Released*. The offset guards the transition *Release*, and it allows to specify an offset between the thread's period and its rerelease. The delay guards the transition from the state *Ready* to the state *Running*, and it specifies a minimum delay until the thread starts consuming resources, similar to temporal scopes. The worst-case execution time specifies an upper bound on how much time can pass until the thread enters the state *Completed*. If the worst-case execution time is not met then a timing violation will occur. Eventually, the thread completes and waits in the state *Completed* until its period is due and it can get rereleased.

Figure 1.7 shows the life-cycle model for functions in threads. The developer can only place a temporal constraint about the maximum elapse time on functions. The maximum elapse time guards the transition from the state *Running* to the state *Completed* and the function can only enter the state *Completed*, if no more than *max_elapse* time units have passed since it entered the running state. Note the semantic difference in the maximum elapse time between temporal scopes and the ARTS kernel. In temporal scopes, the maximum elapse time starts when the scope enters the state *Running*, whereas in the ARTS kernel, clock starts ticking when the function enters the state *Ready*. The reason for this is that the ARTS kernel prohibits specifying minimum and maximum delays for functions.

5.3.2 The Stopwatch

The ARTS kernel provides specification-based support for temporal control. The developer annotates the C++ source code and writes values for the worst-case execution time, the period, the phase, and the delay next to threads and the maximum elapse time next to functions. The ARTS/C++ compiler introduces the token `///#`, which marks the beginning of the ARTS related temporal specification. The constraint is specified as: `///# <priority>, <stack.size>, <wcet>, <period>, <phase>, <delay>`.

Listing 5 implements the task τ_{tsec} of the stopwatch in the ARTS kernel. The task is implemented as a thread. The symbol “_” represents omitted annotation parameters, which are not relevant for the temporal control. As specified in the listing, it has a worst-case execution time of ten milliseconds, a period of ten seconds, a phase of zero, and a delay of zero seconds. In the complete stopwatch example, all four threads are implemented like this but with different periods.

To implement the specified update behavior on the clock’s display, we use priorities explained in Section 5.1. So the thread implementing the seconds’ one column has the highest priority and the thread implementing the minutes’ ten column has the lowest one.

```

Thread Minutes::RT_Thread( ) // # 2, -, 10ms, 10s, 0, 0s
{
    //thread body
    int tens_seconds = get_cur_tens_seconds();
5   tens_seconds = (tens_seconds + 1) % 6;
    set_cur_seconds(tens_seconds);

    ThreadExit( ); //reincarnate this thread
}

```

Listing 5: Task τ_{tsec} using the ARTS kernel.

5.4 Real-Time Specification for Java

Java became a popular programming language during the last decade. To support its use in real-time systems, a committee was formed to add real-time capabilities to Java. The result is the real-time specification for Java (RTSJ). It adds facilities for real-time systems as well as temporal control at the thread level. The goals of RTSJ are to fit real-time programming into the typical Java programming experience: it should be backwards compatible, have no syntactic extensions, and fulfill the “write once, run everywhere” mantra. Consequently, RTSJ offers program-based temporal control, in which the developer must specify temporal behavior as program instructions. And, RTSJ offers means for temporal control only at the thread level.

5.4.1 The Model

RTSJ allows the developer to specify periodic, aperiodic, and sporadic tasks. In the following text, we discuss only RTSJ’s support for temporal control of periodic tasks (c.f., Figure 1.8). The developer can specify an initial offset o_{init} , the task’s period, its deadline relative to the period, and its worst-case execution time. The offset is a relative time and guards the transition from the state *init* to the first release. After the task has been released once, the offset is irrelevant. The task’s period guards the transition from the state

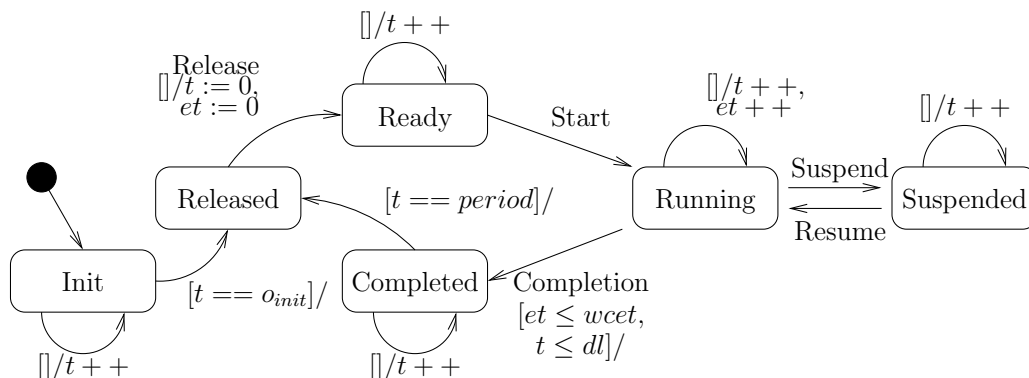


Figure 1.8: Task life cycle in RTSJ.

Completed to the state *Released*. The task will be rereleased once the time value equals its period. The developer can also specify a deadline dl relative to the task's period. This condition, $t \leq dl$, guards the transition from the state *Running* to the state *Completed* and requires the task to finish computation prior to that time. Finally, RTSJ also allows specifying the worst-case execution time. In RTSJ this value is called *cost*. It also guards the transition from the state *Running* to the state *Completed* and limits the time in which the task can reside in the state *Running*.

5.4.2 The Stopwatch

RTSJ is an object-oriented language and provides a program-based approach to specifying temporal constraints. Its programming interface for temporal control is encapsulated in an object named `ReleaseParameters`. Temporal-control parameters for periodic tasks are grouped in the subclass `PeriodicParameters`. When instantiating a new thread, the developer must pass release parameters to the thread's constructor. In the following, we implement the stopwatch's task τ_{tmin} .

```

1 public class TSec extends RealTimeThread {
    ...

    public void run() {
        while (true) {
6         int val = getCurrentTSecValue();
            val=(val+1)%6;
            setCurrentTSecValue(val);
            waitForNextPeriod();
        }
    }
  
```

```

    }
11 }

    TMin createInstance() {
        ...
        PeriodicParameters pp = new PeriodicParameters(offset,
16     new RelativeTime(10.0*SECONDS), // the period
        new RelativeTime(5.0), // the cost
        new RelativeTime(10.0*SECONDS), // the deadline
        null, null);

21     return new TSec(priority, pp);
    }
}

```

Listing 6: Task τ_{tsec} in RTSJ.

Listing 6 implements task τ_{tsec} in a class called `TSec`. Lines 5 to 9 specify the task’s functionality where the counter gets incremented by one and as a minute has at most sixty seconds, its output value is calculated modulo six. The method call `waitForNextPeriod()` in Line 9 declares that this task has completed and it returns control to the scheduler. Since temporal control in RTSJ is program-based, we also must add this code in the source code. Lines 15 to 19 specify the task’s temporal behavior using the class `PeriodicParameters`. The variable `offset` specifies the model’s offset o_{init} . The next parameters specify the task’s period, its worst-case execution time, and its deadline. In our example, we set the deadline equal to the task’s period. The remaining two parameters are for error handling and are irrelevant here. Finally, we instantiate a new thread and return it as the result of the method `createInstance()`. This thread can then be put into the state *Init* by calling the thread’s `start()` method.

5.5 Esterel

The Esterel language [2] is an example for a synchronous languages [1]. A synchronous language bases its model of time on the synchrony hypothesis, which assumes the underlying machine is infinitely fast and program outputs become available at the same time instant as inputs are supplied. This hypothesis

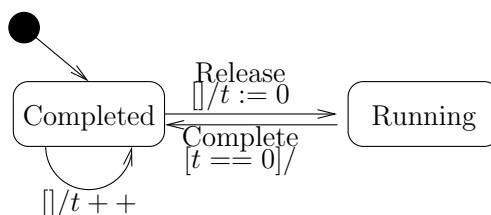


Figure 1.9: Esterel's life-cycle model.

essentially reduces actions to events, and it allows Esterel to define precise temporal constraints. One notable different to the other systems present before is that one of Esterel's goals is to support verification of temporal properties. Its language semantics are mathematically formalized and, leveraging this, the developer can precisely predict the temporal behavior of applications written in Esterel.

5.5.1 The Model

Esterel's life-cycle model is trivial, because every reaction is reduced to an instant. As every task completes instantaneously, the task's life cycle includes only the states *Completed* and *Running*. When a task switches from the state *Completed* to state *Running*, it instantaneously (i.e., the measured time t equals zero) completes and switches back to the state *Completed*.

The synchrony hypothesis shifts the focus of controlling temporal behavior from inside tasks to in-between tasks. Another consequence of the synchrony hypothesis for temporal control is that in synchronous languages it is unnecessary to control the temporal behavior inside the task. Computation of tasks is reduced to an event and multiple tasks cannot interfere with each other, because all of them complete instantaneously; so temporal control inside the task becomes unnecessary. The synchrony hypothesis allows the developer to concentrate on the temporal behavior in-between tasks; he concentrates on what point in time the release events of individual tasks will occur. The language provides several constructs to specify temporal constraints between events. The smallest time unit is one instant, and the Esterel language uses the signal `tick` to separate two instants. Since the computation is reduced to one instant, the developer can specify precisely whether, for example, a followup task starts (and completes) in the same instant or in the next instant. This type of temporal control is more precise in terms of semantics than the other approaches.

5.5.2 The Stopwatch

The Esterel language provides constructs to control the emission of events (called *signals*) and checks for their presence. For the stopwatch example, we show two possible implementations of task τ_{tsec} and discuss the temporal behavior of each. We also explain the used language constructs as we explain the example. The example uses only a subset of the language constructs available. For a full list see [2].

```

await immediate START;
2
abort
  every immediate S do
    count := count + 1;
    emit TIME(count mod 6);
7  end
  when STOP

sustain TIME(count);

```

Listing 7: Task τ_{tsec} in Esterel.

Listing 7 implements the task τ_{tsec} . The listing shows only the functionality part of the task and lacks the module and other declarations encapsulating it. We assume that the signal **S** becomes present every 10 seconds, the valued signal **TIME** is connected to the display, and the signal **STOP** and **START** occur when the stopwatch's start/stop buttons are pressed. The language construct **await** does nothing and terminates when the specified signal is present. The keyword **immediate** further refines the construct and specifies that it also checks the current instant. So, Line 1 waits until the start button *is* pressed. Note the emphasis on the expression 'is pressed', because the program continues within the same time instant. So the **tick** is not over yet. The **abort ... when** presents a preemption mechanism. Without the preemption condition, the statement runs to completion. If the condition ever holds, the inner statement block terminates immediately within the same time instant. This inner statement block, from Line 4 to Line 7, increments the second's ten column every ten seconds. Similar to other programs already shown, we have to compute the display value modulo six to show values between and including zero and five for the ten digit. At the instant, in which the stop button *is* pressed, the program stop incrementing the counter and just continually displays

the last value.

As stated before, Esterel allows precise temporal control down to the level of individual time instants. In the stopwatch example this means: does the stopwatch measure the time between pressing the start and the stop button, or does it measure the time from the moment the start button is pressed until the stop button has been pressed? It is similar to the ways that one can specify ranges with the excluding brackets $(,)$ and the including brackets $[,]$. Listing 8 also implements the stopwatch, however, this time with different temporal behavior. The construct `weak abort` allows the body to run before it terminates. In the second implementation in Listing 8, the program increments the counter at the time instant the stop button *has been* pressed, and then continually displays the value; so it adds one more time instant compared to the previous listing.

```

await immediate START;

weak abort
  every immediate S do
5     count := count + 1;
      emit TIME(count mod 60);
  end
  when STOP
  pause
10 sustain TIME(count);

```

Listing 8: Task τ_{tsec} version two in Esterel.

Esterel does not support specifying priorities explicitly in the programming language, but the execution ordering among tasks can be specified with the parallel operator. Using the operator \parallel , the developer can specify concurrent behavior. In a construct such as $[A \parallel B \parallel C]$, A has priority over B and C , and B has priority over C . This permits specifying priorities on a local level, and this can be used to implement the desired update behavior as in the other systems. So, the required update behavior of the clock's display can be implemented.

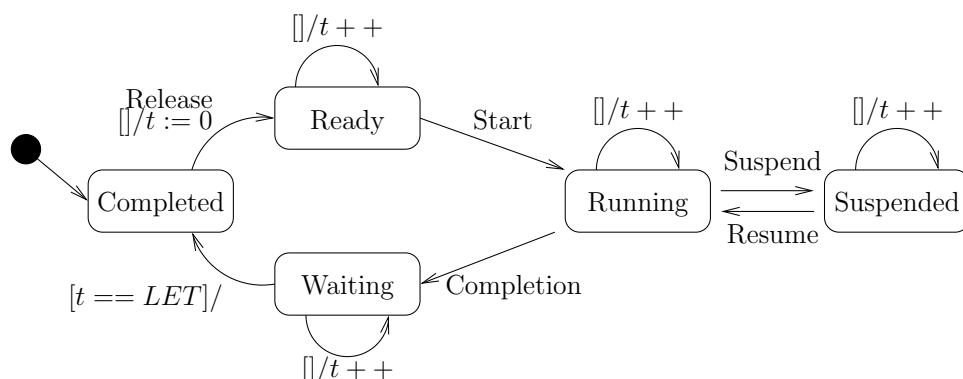


Figure 1.10: Task life cycle in Giotto.

5.6 Giotto

Giotto [8] provides a programming abstraction for specifying temporal constraints. It introduces a new layer, which controls the timing and guarantees value/time determinism between runs as introduced in Section 1. This layer is implemented via a virtual machine, called an embedded machine. Giotto initially aimed strictly at single-node, multi-mode applications with periodic tasks, whose deadline is equal to the period. Its programming abstraction allows coding applications and guaranteeing their temporal constraints without inspecting the source code. Extensions [9, 10] to Giotto introduce support for distributed systems, background tasks, and division of tasks into sub-tasks.

5.6.1 The Model

Giotto realizes a time-triggered approach for computation, which releases tasks in a time-triggered periodic manner. Each task executes the following stages: (1) read inputs (e.g., sensor data), (2) call task-specific functionality, and (3) write outputs (e.g., update actuators). The time between (1) and (3) is called the logical execution time (LET) of a task. The task is released at the beginning and terminated (i.e., completed) at the end of the LET. The release and the completion event are time-triggered. Within these two events, the task-specific functionality is executed according to a scheduling scheme. The start of the LET specifies the point in time when the input values are read. The end of the LET specifies the point in time when the output values are written (even if the task has completed before that time and the output is not available until the end of the LET). A task needs to terminate before the end of its LET, so the worst-case execution

time must be smaller than the LET.

Figure 1.10 shows the task's life cycle in Giotto. The model includes a new state *Waiting*, which is not present in the default model (c.f., Figure 1.1). The only temporal constraint that the developer can specify is the LET. The condition $t == LET$ guards the transition from the state *Running* to the state *Completed*. The state resides in the state *Waiting* until the LET has expired and it switches to state *Completed*. Note that the task cannot reside in the state *Completed*. It is instantly released again, since in Giotto all tasks by construction have a deadline equal to their period.

5.6.2 The Stopwatch

Giotto uses a specification language to describe temporal constraints of tasks. In this language, the developer specifies only the timing, but not the functionality. Other programming languages, such as C, can be used to program the functionality. For a specification of the Giotto language see ???. In the following, we specify the timing of the stopwatch's tasks in Giotto's specification language.

```

start Started {

mode Started() period 3600 {
    actfreq 3600 do act_sec(a_sec_driver);
5   taskfreq 3600 do comp_sec(sec_driver);

    actfreq 60 do act_tsec(a_tsec_driver);
    taskfreq 60 do comp_tsec(tsec_driver);

10  actfreq 10 do act_min(a_min_driver);
    taskfreq 10 do comp_min(min_driver);

    actfreq 1 do act_tmin(a_tmin_driver);
    taskfreq 1 do comp_tmin(tmin_driver);
15 }
}

```

Listing 9: The stop watch in Giotto.

Listing 9 declares all four tasks for the four columns on the stop watch. Each task consists of a driver that is responsible for reading and writing its values, and an actuator for updating the clock's display. The listing also declares only one mode; the full example would need three modes: *Stopped*, *Started*, and *Reset*. Guards would then specify mode switches, so for example, the system starts in the mode *reset* and when the user presses the start button, it switches into the mode *Started*. Listing 9 also excludes the declaration of drivers, tasks, and sensors.

Listing 9 declares the start mode to be *Started* in which the stop watch is running. This mode *Started* has a period of 3600 seconds. Within this period, all tasks execute with a specific frequency. For example, the task `comp_tsec` executes with the frequency 60, so it executes 60 times within 3600 seconds which amounts to once every ten seconds. The actuators also execute with a specific frequency when updating the clock's display. For example, the actuator `act_min` has a frequency that results in the actuator's execution once every minute. The functionality code for both the task and the actuator may be implemented in an arbitrary programming language.

Giotto's value update behavior differs from the other systems. The actuators are executed at a specific frequency and are independent of the task. Given a program as specified in Listing 9, a compiler generates code that ensures the timely execution of these actuators. For example, at 12 o'clock, all four actuators will be executed concurrently. Consequently, the clock's display will change independently from the task's run-time behavior, because it is irrelevant when, for instance, task `task_min` produces its new value as long as it produces this new value before its actuator is activated. However, Giotto lacks mechanisms for specifying priorities for actuator updates, so it is up to the code generator to put the actuator execution into a specific order. Consequently, the updates on the clock's display will always be the same in all runs, however, the developer cannot specify this behavior in Giotto.

6 Comparison and Conclusions

Temporal control is an essential part of real-time systems. It means putting constraints on the program with respect to timing. Here we summarize the examined systems and compare them. Note that this comparison looks only at the direct support in the language and system, because virtually any temporal constraint can

| Name | Granularity | Task model | Type | Constr. | Err. handling |
|--------------|-------------------------|---------------------------|--------------------|-----------|---------------|
| PEARL-90 | Task | per,sus | Spec. | Abs.&rel. | No |
| Temp. scopes | Statement | off, per, dl ^a | Spec. ^b | Abs.&rel. | Exceptions |
| ARTS kernel | Task, fun. ^c | ph, off, per ^d | Spec. ^b | Rel. | No |
| RTSJ | Task | off, per, dl | Prgm. | Abs.&rel. | Exceptions |
| Esterel | Statement | Reactive | Prgm. | - | No |
| Giotto | Task | per | Spec. | Rel. | No |

^aAlso timing control for one-time execution (e.g., statement blocks) with offset.

^bAlthough it is specification-based, it intertwines code and timing specification.

^cArts provides different temporal control elements for tasks and functions.

^dAlso offers deadlines for function calls.

Table 1.1: Comparison sheet

be programmed implicitly.

Esterel and other synchronous languages [1] provide the most elaborate control. The reduction of computation to time instances allows the developer to specify precise temporal constraints such as: does the stopwatch stop at the moment you press the stop button or right after that moment has passed. The PEARL language and temporal scopes provide the most elaborate support for specifying temporal constraints for tasks and statements. PEARL concentrates on temporal constraints for task releases and when they are resumed; however, it leaves out control for deadlines and task completion. Temporal scopes provide means for controlling task releases as well; they concentrate on constraints during the task's runtime such as minimum/maximum delay and maximum elapse time. Temporal scopes also provide the most fine-grained support for temporal constraints, since they permit placing constraints on individual statements. Most other systems, except ARTS, only operate at the level of tasks. ARTS permits placing constraints on tasks and individual functions. Esterel also permits coding constraints on individual statements, however, it is more an implicit control compared to temporal scopes. Also, all systems with tasks can theoretically wrap individual statements into a single task and thus also support constraints on individual statements.

For each of these systems, we presented a state machine showing the life-cycle of tasks, functions, and statement blocks. Table 1.1 provides an overview of the state machine's properties of the systems and languages that we discussed in this chapter. The first column lists the name. The second column lists the

code granularity, the system provides the developer to place temporal constraints. The third column lists the constraints that are supported in the system and language. The legends are as follows: *per* means supporting periodic rerelease, *dl* means supporting deadlines different from the task period, *off* means supporting task offsets from its release, *sus* means supporting suspension constraints, and *ph* means phasing.

The fourth column lists the type of support for temporal constraints the system provides. Two systems – RTSJ and Esterel – require the developer to implement the temporal constraints using the programming language. RTSJ provides an object-oriented programming interface with classes like `ReleaseParameters` and `RelativeTime`. In Esterel, all these concepts have to be implemented on the basis of *ticks*, which separates time instances from each other. The remaining systems offer specification-based support. PEARL and temporal scopes use a very readable statement to declare temporal constraints for tasks and program blocks. In the ARTS kernel this declaration is more cryptic, because it is merely a sequence of comma-separated numbers and each position denotes a particular parameter in the temporal constraint. For example, the fourth parameter in the list specifies the task period. Giotto uses a different approach than the other systems with specification-based support. In the previously mentioned systems, the temporal constraints and the program logic are intertwined and stored together in one file. In PEARL a clear separation can be made, but a temporal constraint can also be placed somewhere in the code. In Giotto, temporal constraints are specified independently from the program sources and are completely separated from the program logic.

The fifth column lists what kinds of temporal constraints the system supports. Only PEARL and temporal scopes support absolute temporal constraints throughout the life cycle. PEARL provides the `AT` instruction and temporal scopes provide repetitive temporal scopes that can also use absolute time. RTSJ supports also supports absolute constraints, however, throughout the whole life cycle. All other systems only support relative temporal constraints, although absolute constraints can be implemented implicitly.

Finally, the last column lists the system's support for error handling. Temporal scopes and RTSJ support timing error handling. Both systems use the concept of exception handling, and in case the run-time system detects a timing violation, it raises an exception and executes the programmed exception handler. ARTS also supports error handling; however, it is limited to timing violations in functions, not in tasks.

The language constructs and systems overviewed in this chapter are a representative subset of the many

different systems and languages available. Most real-time kernels provide a programmable interface for temporal constraints and they use similar concepts as presented in this work. A large number of systems support the basic temporal constraints such as period and deadline. For example, SHaRk [11], Asterix kernel [12], the HARTIX kernel [13], nano-RK [14], timed multitasking [15], and virtually all commercial real-time systems. Some of them have interesting extensions of the presented concepts. For example, the Spring kernel [16] allows specifying a function to compute the worst-case execution time depending on the current task inputs. It also introduces incremental tasks that produce an immediate result and use the remaining time to refine that answer, similar to imprecise computation [17]. In such systems, temporal constraints can be specified for both parts: the mandatory part for the immediate answer and the part that refines the result. Another system is TMO [18], an acronym for *time-triggered, message-triggered objects*, that also supports temporal constraints on tasks and functions similar to the presented systems; however, its main focus is on the object-oriented programming paradigm.

As a concluding remark, we note that as timeliness of computation is a requirement in real-time systems, temporal constraints are a fundamental concept for such systems and research on how to specify and where to place these constraints has been extensive. The investigated sample of systems and languages shows their different approaches and their different levels of support for placing temporal constraints; however, none has it all. Depending on the needs of the actual application, one or the other system may be better suited. Future research on this topic should consider the currently established work and consolidate the different approaches into one unified framework for placing temporal constraints using a specification-based approach with comprehensive error handling.

References

- [1] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer, 1997.
- [2] G. Berry. *The Esterel v5 Language Primer*. Centre de Mathematiques Appliquees, Ecole des Mines and INRIA, 2004 Route des Lucioles, 06565 Sophia-Antipolis, version 5.21 release 2.0 edition, April 1999.
- [3] S. Fischmeister and K. Winkler. Non-blocking deterministic replacement of functionality, timing, and

- data-flow for hard real-time systems at runtime. In *Proc. of the Euromicro Conference on Real-Time Systems (ECRTS'05)*, 2005.
- [4] *DIN 66253, Teil 1: Informationsverarbeitung, Programmiersprache PEARL, Basic PEARL Entwurf*. Beuth Verlag Normenausschu Informationsverarbeitung im DIN Deutsches Institut für Normung e.V, Berlin, Köln, June 1978.
- [5] GI-Fachgruppe Echtzeitsysteme und PEARL. *PEARL 90 - Language Report*, 2.2 edition, 1998.
- [6] I. Lee and V. Gehlot. Language constructs for distributed real-time programming. In *Proc. of the IEEE Real-time Systems Symposium (RTSS)*, 1985.
- [7] H. Tokuda and C. W. Mercer. Arts: a distributed real-time kernel. *SIGOPS Oper. Syst. Rev.*, 23(3):29–53, 1989.
- [8] T. A. Henzinger, C. M. Kirsch, and B. Horowitz. Giotto: A time-triggered language for embedded programming. In T. A. Henzinger and C. M. Kirsch, editors, *Proc. of the 1st International Workshop on Embedded Software (EMSOFT)*, number 2211 in LNCS. Springer, October 2001.
- [9] G. Menkhaus, S. Fischmeister, M. Holzmann, and C. Farcas. Towards Efficient Use of Shared Communication Media in the Timed Model. In *Proc. of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2005.
- [10] S. Fischmeister and G. Menkhaus. Task Sequencing for Optimizing the Computation Cycle in a Timed Computation Model. In *23rd International Digital Avionics Systems Conference (DASC'04)*. IEEE Press, 2004.
- [11] P. Gai, L. Abeni, M. Giorgi, and G. Buttazzo. A New Kernel Approach for Modular Real-Time Systems Development. In *Proceedings of the 13th IEEE Euromicro Conference on Real-Time Systems (ECRTS'01)*, June 2001.
- [12] H. Thane, A. Pettersson, and D. Sundmark. The Asterix Realtime Kernel. In *Proc. of the 13th Euromicro Conference on Real-Time Systems (ECRTS'01)*, June 2001.

- [13] C. Angelov and J. Berthing. A jitter-free kernel for hard real-time systems. In *Proc. of the 1st International Conference on Embedded Software and System (ICCESS'04)*, Hangzhou, China, December 2004.
- [14] A. Eswaran, A. Rowe, and R. Rajkumar. nano-RK: An Energy-Aware Resource-Centric Operating System for Sensor Networks. In *Proc. of the IEEE Real-Time Systems Symposium (RTSS'05)*, December 2005.
- [15] J. Liu and E. Lee. Timed multitasking for real-time embedded software. *IEEE Control Systems*, 22(6):65–75, 2003.
- [16] J.A. Stankovic and K. Ramaritham. The Spring Kernel: A New Paradigm for Hard Real-Time Operating Systems. *IEEE Software*, 8(3):62–72, 1991.
- [17] Swaminathan Natarajan. *Imprecise and Approximate Computation*. Kluwer Academic Publishers, Norwell, MA, USA, 1995.
- [18] K.H. Kim. Object structures for real-time systems and simulators. *IEEE Computer*, 30(8):62–70, 1997.

Index

- aperiodic, 7
- background/foreground system, 10
- completion event, *see* event, completion
- constraint
 - end-to-end, 4
- delay
 - end-to-end, 4
 - maximum, 16
 - minimum, 16
- determinism
 - time, 4
- elapse time
 - maximum, 16
- end-to-end
 - constraint, 4
 - delay, 4
- event
 - completion, 6
 - release, 6
 - resume, 6
 - start, 6
 - suspend, 6
- Giotto, 27
- interrupt service routine, 10
- ISR, *see* interrupt service routine
- maximum delay, 16
- maximum elapse time, 16
- minimum delay, 16
- PEAL, 13
- PEARL-90, *see* PEARL
- period
 - task, 7
- periodic, 7
- release event, *see* event, release
- resume event, *see* event, resume
- sporadic, 7
- start event, *see* event, start
- suspend event, *see* event, suspend
- system
 - background/foreground, 10
- task period, 7
- temporal control
 - implicit, 9
- temporal constraints
 - program-based, 12
 - specification-based, 12
- temporal constraint
 - absolute, 4

firm, 3

hard, 3

relative, 4

soft, 3

time determinism, 4

timing violation, 4