Communicating Process Architectures 2003
Jan F. Broenink and Gerald H. Hilderink (Eds.)
IOS Press, 2003

Sampling and Timing: A Task for the Environmental Process[†]

Gerald H. HILDERINK and Jan F. BROENINK
Twente Embedded Systems Initiative,
Drebbel Institute for Mechatronics and Control Engineering,
Faculty of EE-Math-CS, University of Twente,
P.O.Box 217, 7500 AE, Enschede, the Netherlands
g.h.hilderink@utwente.nl

Abstract. Sampling and timing is considered a responsibility of the environment of controller software. In this paper we will illustrate a concept whereby an environmental process and multi-way events play an important role in applying timing for untimed CSP software architectures. We use this timing concept for building our control applications based on CSP concepts and with our CSP for C++ (CTC++) library. We present a concept of sampling of control applications that is orthogonal to the application. This implies global timing on the basis of timed events. We also support traditional local timing on the based of timed processes,

1. Introduction

At Control Engineering at the University of Twente, we use concepts from the theory of *Communicating Sequential Processes* (CSP) [1; 2] to design real-time control software architectures and to reason about concurrency and their reactive/real-time behaviours [3]. The CSP concepts contribute in managing complexities and provide guidelines for a clean and prescribed software development approach. The CSP concepts provide an excellent separation of concerns at a high level of abstraction in terms of processes and their interrelationships. The result is that nothing will be designed or implemented in an ad-hoc manner and students easily learn these abstract concepts of designing and implementing real-time software.

The real-time control software is the software part of a mechatronic system that aims at improving the dynamics of the system or to automate it. The dynamic behaviour of the plant, i.e. the 'machine'-part of the mechatronic system, imposes timing requirements on the software. The control loop from obtaining sensor data via processing to sending actuator signals must be completely processed within every sampling period *Ts*. This sample period *Ts* is determined by the dynamic behaviour of the plant and lies in our domain typically between 0.1 to 10 ms. Furthermore, all sensors must be sampled at the same moment causing as little jitter as possible (sampling). The actuators too must set their new value at the same moment (actuation). In practice it is often required that sampling is done before actuation in order to avoid disturbance of the measurements by the actuation. The distance between sampling and actuation is fixed within the sampling period. See Figure 1. The inputs can be done in parallel and the outputs can be done in parallel, or a

[†] This research is supported by PROGRESS, the embedded system research program of the Dutch organization for Scientific Research, NWO, the Dutch Ministry of Economic Affairs and the Technology Foundation STW.

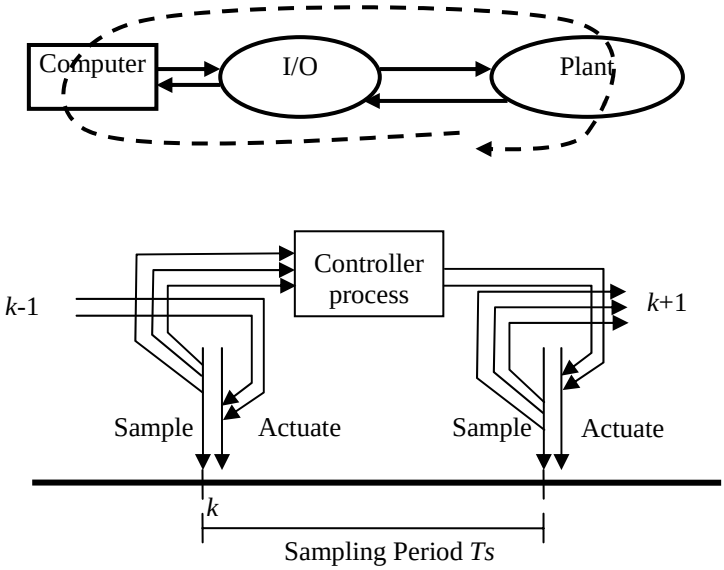


Figure 1 Real-time control loop (above) and timing of one sample–process–actuate–loop where the k^{th} sample is shown (below)

fixed sequence can sometimes be more practical when converters are under control of the software.

Real-time implies on the one hand that the response is expected before a deadline or precisely on a specified moment (neither before nor after). Its timeliness implies correctness. On the other hand real-time systems are intrinsically concurrent because they are coupled to the real world. Discrete control systems require a constant and precise sampling rate of their inputs and outputs to the real world they control. With precise we mean a predefined moment in time within a narrow range of variation that is acceptable to ensure a reliable and stable control system. Furthermore, the processes between these inputs and outputs should guarantee that they perform their tasks and deliver their outputs for actuation before their deadlines expire. Each control loop deals with a parallel aspect of the system and therefore control loops are naturally parallel to each other.

Sensors and actuators require signal conversions between the continuous-time and real world representation and discrete-time and digital representation. Common converters are Analog to Digital Converters (ADCs), Digital to Analog Converters (DACs), and counters. These converters require accurate timing with as less jitter as possible between the input sampling and output actuation, and with a fixed distance between the successive sampling events.

A shared clock between the converters is the fastest and therefore the most accurate solution, but not every interface card has timers providing a shared external timer among all peripheral devices. Often configurations require the system timer; the internal timer on the computer board where the software (e.g. the interrupt service routine) has control over the converters. The correspondence between these different configurations of external and internal timers is that timing is part of the environment of the application and orthogonal to the application. This separation of concerns can be used to our advantage when we make an untimed design timed through encapsulation within its environment orthogonal to the design.

We use a graphical CSP notation by Hilderink [4] to illustrate the examples in the form of CSP diagrams.

In this paper we propose a conceptual approach to incorporate timing for accurate sampling on external CSP channels in software. This approach differs from the solution found in the programming language occam [5] and the CSP for Java libraries JCSP [6] and

CTJ [7]. This proposal may become a basic concept behind the timing framework of CT developed by Hilderink [7]. CT stands for *Communicating Threads* which provides concurrent programming where multithreading in execution is encapsulated in CSP-like abstractions of processes and channels. Currently, this timing framework has been successfully implemented in CT for C++ (CTC++).

Section 2 discusses the problem with timed processes as a result of local timing. The conceptual background to this proposal from which our solution is derived is discussed in Section 3. The solution in the form of environmental services is presented in Section 4. Section 5 applies these services in two examples.

2. Sampling problems with timed processes

In occam and on the transputer, time was provided by a TIMER object [5]. The TIMER object is a kind of channel on which processes can read the time or on which processes get blocked until the specified time has expired. The same syntax is used to specify a timeout-guard in the alternative construct. A similar object, called CSTimer, is defined in JCSP [6]. CTJ implements a Java-like static sleep method that will block a process for a specified time and CTJ provided separate timeout-guards for timeouts on alting. These timing concepts in occam, JCSP and CTJ are encapsulated in processes, called *timed processes*. The problem is that these processes cannot guarantee precise sampling on channel communication. In particular, external channels are concerned with data conversion between the digital computer and the analog world at specified moments in time. The timed processes do not provide an adequate solution to sampling and timing issues on channels.

In this example, we illustrate a few problems with timed processes concerning sampling. Consider two independent controller processes, LController and HController, as depicted in Figure 2. The channels c_1, c_2, and d_1 are connected to sensors and channels c_3 and d_2 are connected to actuators. In this example, input-channels c_1, c_2, and d_1 communicate with an AD converter each and output-channels c_3 and d_2 communicate with a DA converter each. Of course, processes see channels and *not* the converters. In this sense they are hardware independent. They only depend on the integrity of the values. The HController process repeatedly reads values from channel d_1 and after computation the result is sent to channel d_2. The LController process repeatedly reads values from channel c_1 and c_2 and after computation the result is send over channel c_3.

At the open-ends of the arrows one can imagine processes in hardware to which processes in software communicate. These processes in hardware are not rendered in the diagram, because they are out of the context of the control software.

The sensor data on channels c_1, c_2 should be sampled at the same time. Channel c_3 should be actuated shortly after c_1 and c_2 are sampled, in order to not disturb the measurements at c_1 and c_2. This is similar for the channels d_1 and d_2 of HController but this can be done at the same or a different sampling interval than for LController.

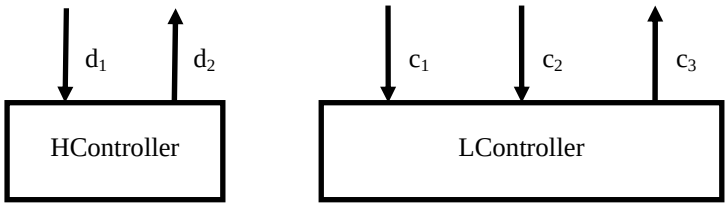
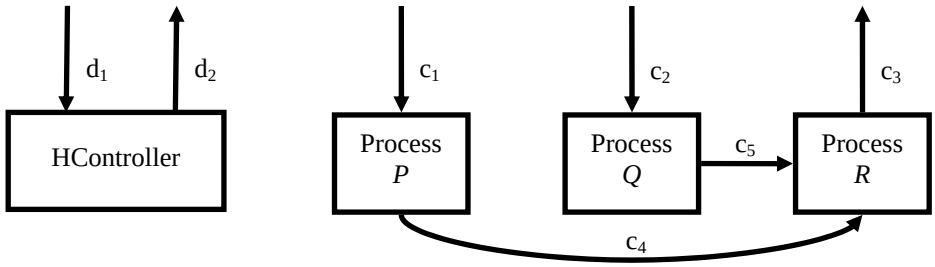


Figure 2 Example: control application consisting of a higher-priority controller process and a lower-priority controller process



(a) Communication relationships

(b) Compositional relationships

Figure 3 Lower-priority controller process divided into sub-processes

In this example we will assume that HController operates at a higher sampling frequency than LController. This requires that HController executes at a higher priority then LController. The problems that rise in this situation are common problems with timed processes. We will illustrate this by the following example in which LController consists of three sub-processes each carrying out a computational task, see Figure 3a. We will omit their functional descriptions.

Process *P* receives information from channel c_1, process *Q* receives information from channel c_2, and process *R* determines the output and sends the results to channel c_3. The processes *P* and *Q* send the results of their calculations to process *R* via the internal channels c_4 and c_5. All processes are part of the same control loop and therefore the channels c_1 and c_2 should be sampled and c_3 should also be actuated at the same sampling frequency. As mentioned in Section 1, the order of reading and writing should be fixed and reads and writes should be very close to each other: first input on c_1 then input on c_2 then output on c_3. Any varying delay between the inputs and outputs can cause significant jitter that may bring the system into an unstable state. Figure 3b shows that all three processes are meant to execute in parallel, $P \parallel Q \parallel R$. This example suffers from the following problem when we apply the timed processes for sampling.

Sampling cannot be based solely on reading and writing on channels. Pre-emption and arbitrary scheduling of processes can cause a varying delay between reading and writing on channels. Thus, sampling on reading and actuation on writing on channels is not an adequate solution. The resulting variation in sampling intervals or jitter can make plant unstable and unpredictable. It is important that jitter is negligibly small which makes any delay approximately constant compared to the sample time Ts. Thus, scheduling of the processes does not guarantee a fixed order of inputs and outputs. Any order is possible. Serializing these processes does not really solve the problem. This is because the computations that are performed by the processes cause a distance (delay) between the inputs and outputs. To overcome this problem we need to make sure that sampling and actuation is done fast and in direct sequence. Disassembling the processes and moving the inputs and outputs to another sequential process is not an option because we want to specify three parallel tasks for some reason that is not addressed here. Serializing requires breaking the internal loops of the processes into a common outer loop. The problem increases when multiple frequencies are involved. This approach has tremendous effect on architectural

changes and prevents reusing processes.

A common problem with the sleep/after method of the timer object is that after its wakeup *the process can be delayed by a higher priority process*. This is obvious the case when two controllers at two different sampling frequencies are performed. In order to fulfil the real-time requirements we apply the rate-monotonic priority ordering; the process with the higher frequency gets higher priority than a process with a lower frequency. Atomic coherency between conversions during sampling and actuation in a control loop is required, which a (timed) process cannot provide. This particular order of sampling and actuation should be orthogonal to the software architecture otherwise this would have a tremendous impact on the design.

❑

In order to achieve sampling, special precautions in hardware or in software have to be taken. In hardware, a common external timer on which peripheral devices are triggered and an atomic sequence of inputs and outputs is guaranteed. In software, this implies some sort of ceiling priority for each sequence of pairs of inputs and outputs to guarantee adequate sampling and actuation; the processor interrupt handling can be used for this.

After conversion the devices usually generate an external interrupt that reflects a timed event on which values are read from peripherals into memory or values are copied from memory to the device. Unfortunately, not all devices are supplied with a common timer whereby devices are triggered at the same time. More often devices are triggered by a timed interrupt service routine of low-level instructions using the internal timer of the computer. Often, this is a flexible solution whereby the software stays in control over the devices.

In this paper we bring this concept of internal and external timing to a higher level of abstraction conform the CSP concepts on which we build our applications. We will apply timed events instead of timed processes, which proposal conceptually resembles with timing in hardware *and* timing in software. With the latter we mean that we can also create timed processes based on the same proposal. If this proposal is carefully applied then the software architecture becomes independent of external timing such as sampling and actuation.

3. Timing concept

3.1 Timing in CSP

Timing in CSP primarily concerns events, because *an event is an occurrence in time and space*. An event happens at a certain moment in time (e.g. periodic or sporadic) and somewhere in the system. We can observe the real-time behaviour of processes by measuring the time between events in which the processes engage. This is a simplified and a more pragmatic approach than determining the exact execution time of processes. Traces of events help determining the possible patterns of events in time. This takes into account the compositional relationships between processes. The worst case time can help in determining that processes can guarantee their deadlines. Timing analysis is not discussed in this paper.

The CSP concepts we use are based on untimed CSP [1; 2] which theory does not deal with intrinsic timing concerns. Timed CSP [8] does deal with timing, but it does not offer a pragmatic approach to be useful in software. The problem is that the timeout operator in timed CSP is far too complex and inefficient to build in software. Furthermore, the timeout operator requires semantically changes to original CSP operators.

We will address a different timing concept for untimed CSP designs, which does not need any semantically changes to the original CSP operators (i.e. ;, ||, and □) and to

extended CSP operators (i.e. $\overleftarrow{\|}$, $\overleftarrow{\Box}$, and $\overrightarrow{\Delta}$). In this approach, exceptions are involved but these exceptions are orthogonal to the original semantics—the original semantics of these CSP constructs remain intact. The exact exception handling is not discussed in this paper.

CSP defines that processes communicate with each other through shared channel objects. Processes that are willing to communicate on a channel will be blocked until the other side is willing to communicate. When a reader process and a writer process engage in communication then data is passed from writer to reader. At that moment both processes will continue in parallel. This moment is called *the communication event*. For sampling it is important that those communication events occur at periodic moments in time. This involves external channels that connect the controller processes in software to processes in hardware such as signal converters to which sensors and actuators are connected.

Considering the notion of events, we believe that the time stamp of occurrence is a property of an event (see previous definition of event) and not a property of a channel. However, timing can be performed by channel communication when some sort of process at a peer-end of a channel is willing to communicate at a precise moment in time.

3.2 Environmental process and timing

In CSP, an event requires two or more processes to engage in—an event is multi-way. However, a communication event with channels is considered to be strictly two-way—one producer process and one consumer process must be willing to communicate in order to let the communication event happen. Another form of communication between multiple processes is a barrier object as described in the theory of BSP [9]. When all participating processes synchronize (block) on a barrier object then data exchange between the processes takes place and successively the barrier releases the processes to continue in parallel. The rendezvous is the communication event. With a barrier object a communication event is two-way or multi-way which value is equal to the number of participating processing ($\#\geq 2$).

In CSP textbooks, the environment is often mentioned as an external or *environmental process* that also must be willing to accept events. We will assume that this includes communication events. In this approach we take the environmental process into account which also participates in communication events. An environmental process is basically another CSP process that synchronizes on internal events. In this case a communication event can be considered to be $(n+1)$-way, where $n=2$ for channel synchronization and $n\geq 2$ for barrier synchronization. This is exactly what this approach is about.

The environment can influence the behaviour of concurrent software in several ways. This concerns the acceptance of events. In an ideal environment all events are accepted in the application. This idealness is usually the starting-point of a design. In reality, things can go wrong whereby the environment can become partially disabled. In this case the system may not engage in certain events. For example, consider a plant with embedded systems and field-busses whereby a communication cable is accidentally cut or plugged-out. When that happens no communication events will happen over a disconnected channel. The reverse is also possible when a disconnect channel becomes connected. These disabilities and abilities can rise in time. The non-idealness of the environment is considered during the refinement of the design.

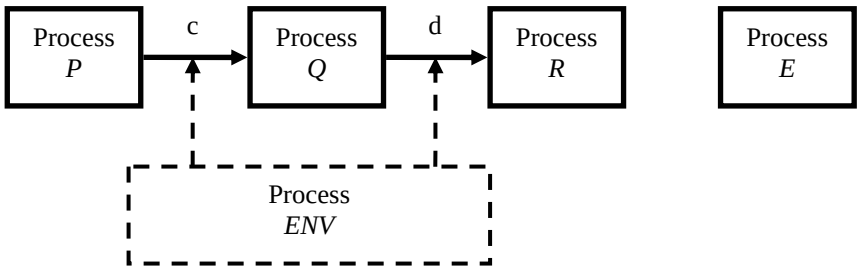
More generally, when a producer process and a consumer process are willing to communicate then the environment should also accept the event in order to let the event happen. It is possible that the environment cannot accept the event.

The environment is time-variant (depending on time) and memory-based (non-anticipating) and can contain multiple timelines. The timelines can be divided in *continuous time*, *discrete time*, or a hybrid of both. For example, a processor performs its tasks in

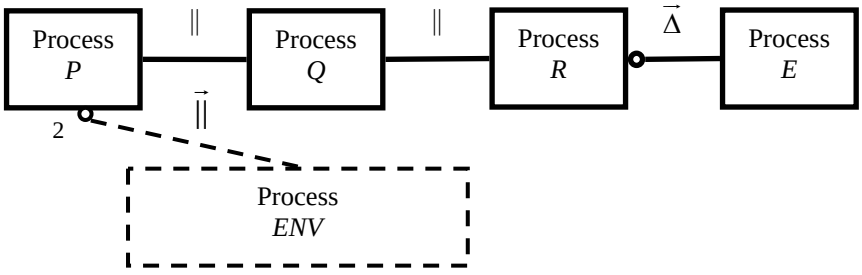
discrete time. The entire system, consisting of the computer (discrete time) connected to the real world (continuous time), is called a hybrid system. Therefore, we consider time a property of the environment being not pre-emptable. Time and the environment are orthogonal to the application. Of course, the sampling period could be a constant parameter used by the control laws that are performed by the control application, but this is simply and solely a value derived from the dynamics of the system.

In our proposal, the environmental process can perform services for the application processes that are running in the environment. These processes can call upon these services through call-channels. In this paper we introduce such an environmental process with services that provides us means for sampling. The environmental process is invisible to the application and is not part of any compositional construct in the application. This environmental process can be very complex and we do not always have to know what this environmental process exactly is. We only need to know in what way the environment can influence the application. This may or may not be in control of the application. In a distributed system, every computer system has a separate environmental process.

Figure 4 illustrates the presence of an environmental process in a CSP diagram. Consider the three processes *P*, *Q*, and *R* communicating over channels *c* and *d* in Figure 4a. For completeness, Figure 4b specifies that *P*, *Q*, and *R* are in parallel and that the objects *c* and *d* are rendezvous channels on which the processes synchronize on communication. For internal channels we assume that *P* and *Q* engage in the communication event when they both are willing to communicate over channel *c* whereby *ENV* always accepts that communication event. Also we assume that no exceptions occur. This is similar for the processes *Q* and *R* with channel *d*. This assumption is not general, because we forget the facts that events can only occur in an environment that accepts them, i.e. the environmental process *ENV* must also be willing to engage in these events. The environmental process *ENV* is the highest-priority process that is never pre-empted by the application processes. Usually, process *ENV* is hidden from the design because it is not explicit part of the design;



(a) Communication relationships with environmental process ENV



(b) Compositional relationship with environmental process ENV

Figure 4 Example with visualized environmental process

therefore the process and relationships are dashed. It is clear to see that *ENV* is connected to each channel in the design. Process *ENV* does not send any data. On error, process *ENV* can throw an exception to pair *P*-*Q* or pair *Q*-*R* with an error message that specifies the reason why the communication event cannot take place. This exception can be handled by the exception handling construct in the application, see process *E* and the $\vec{\Delta}$-relationship.

4. Environmental services

The following methods are services that can be carried out by the environmental process. These methods are called by the control application and they are served when the environmental process is willing to accept the call. We developed an `Environment` class that provides a global static call-channel which service can be invoked by any process at any time. All methods, except for the `time()` method, act upon a communication event and require a channel or a barrier. This includes timed-guards in the alternative construct. The call-channel approach makes these methods automatically thread safe.

We apply these methods systematically by accommodating these to the network building process that creates the network of processes and channels. Thus, the sampling frequencies are set at the top-level of the application.

- **Accept communication event `at` specified time**

```
Environment::at(channel, time);
Environment::at(channel, time, Time interval_time);
```

The producer and consumer processes need to be willing to communicate before the environmental process is willing to accept the communication event on the specified channel or barrier at the specified time (in microseconds) or period. If this is not the case and the processes engage in the event after the specified time expires then any blocked process will be released and an exception is thrown at the producer and at the consumer. Hence, the real-time requirement has not been met.

❑

```
Environment::at(barrier, time);                    // single-shot
Environment::at(barrier, time, interval_time);     // periodical
```

The environmental process will participate in the barrier synchronization and will commit to the synchronization at the specified time. If one participant does not sync before the environmental process then exceptions will be thrown to all processes and all processes will be released. Hence, the real-time requirement has not been met.

❑

- **Accept communication event `after` specified time**

```
Environment::after(channel, time);                 // single-shot
Environment::after(channel, time, interval_time);  // periodical
```

The communication between the producer and consumer processes will be delayed until specified time. Any communication after the specified time will be accepted and they both immediately continue. No exceptions are thrown. If an interval time is specified then the next waiting time will be incremented with the interval time.

❑

```
Environment::after(barrier, time);                    // single-shot
Environment::after(barrier, time, interval_time);     // periodical
```

The environmental process will participate in the barrier synchronization and will commit to the synchronization at specified time. No exceptions are thrown. If an interval time is specified then the next waiting time will be incremented with the interval time.

```
Environment::after(guard, time);                      // single-shot
Environment::after(guard, time, interval_time);       // periodical
```

If the alternative construct is waiting and the alting process at the other end is willing to communicate before the specified time then the guard will become ready at the specified time. Otherwise the guard will be ready when the alting process at the other end is willing to communicate. This guard is called a *timed-guard*. No exceptions are thrown. A timed *skip-guard* is used for specifying a *timeout-guard*. The skip-guard will be ready at the specified time and no exceptions are thrown at timeout. As with channels and barriers the guard can be periodically timed. The specified interval time increments time each period. Since guards are local to a process, implies that `after(guard,time,..)` is used locally and no other process can alter time on a local guard.

The timed-guard can be used with `at(channel,time,..)` and `after(channel,time,..)`.

The method `after(guard,time1,..)` is independent of the time as specified with `at(channel,time2,..)` or `after(channel,time2,..)`. The method `after(guard,time1,..)` on `channel` could very well endanger the deadline as specified by `at(channel,time2,..)` which results in an exception when `time1` > `time2`. Although this is a natural behaviour, this combination is not very applicable. Therefore, `after(guard,time1,..)` and `at(channel,time2,..)` should be used with care to avoid timeout exceptions

We do not support guards for barriers, because we are uncertain about multiple alting on a barrier; this introduces a conflict that is similar to two-way alting on a channel (i.e. input-guarding and output-guarding on one channel) at the same time as described by Jones [10].

❑

The (periodic) timing stops when `accept(channel)` or `refuse(channel)` are used.

- **`Accept` communication**

```
Environment::accept(channel);
Environment::accept(barrier);
```

The environmental process will accept any communication event on the specified channel or barrier. This will cancel any timing as specified with `at(..)` or `after(..)`, or any refusal that was specified with `refuse(..)`. Also after a refusal of the channel one can perform accept to cancel the refusal. If the channel or barrier cannot be accepted then an exception it thrown. The `accept(..)` method is synchronized. The method will block when it was called before `at(..)`, `after(..)`, or `refuse(..)`. This prevents any race hazards between the methods. A guard is not influenced by the environmental process.

The `accept(..)` methods can be used to undo any timing on channels or to let the environment accept an event that it refused. Furthermore, if the environment does not want to accept the event then the application has no control over it and a `UnacceptableException` results.

❑

- **`Refuse` communication and (optionally) throw exception**

```
Environment::refuse(channel, exception_message);
Environment::refuse(barrier, exception_message);
```

This method will let the environmental process refuse the acceptance of the communication event on the specified channel or barrier. If an exception message is specified then it will let the channel or barrier throw the exception message to the participating processes. The exception message is passed to the producer and consumer processes. If no exception message is specified then the channel or barrier will block processes until the environment is willing to accept the events.

The `refuse(..)` method can be used to command the environment that an artificial refusal should be carried out. This method can be used for two main reasons:

1. The application can be tested on possible failures on communication. This way the robustness of the application can be tested.
2. In case the application deadlocks or livelocks then there is no way to terminate the program or a particular part of the program. If an deadlock or livelock is detected then with `refuse(..)` one can throw exceptions to channels or barriers that will release synchronization and exception handling must gracefully terminate the program.

A `UnacceptableException` results if the environmental process cannot refuse events on the specified channel or barrier.

❑

- **Get the actual `time`**

```
Environment.time() : Time
```

Returns the absolute time read by the environmental timer.

❑

- **Let a process `sleep` or `sleepUntil` a specified time**

Besides the `Environment` class CTJ supports the `Thread` class that is similar as the Java Thread class in order to manipulate the thread of control in processes. Since no channel or barrier is directly involved we cannot use the environmental process. Thus we need the help of the `Thread` class instead of the `Environment` class. The services by the Thread class are local to the process and thread-oriented rather than communication event oriented.

```
Thread.sleep(relative_time)
```

Let the thread of control in a process sleep for the specified relative time.

❑

```
Thread.sleepUntil(absolute_time)
```

Let the thread of control in a process sleep until the specified absolute time.

❑

These methods may internally use the `Environment` methods to perform temporarily blocking of the process on some hidden timed event. For sampling and steering we do not use these methods.

5. Example

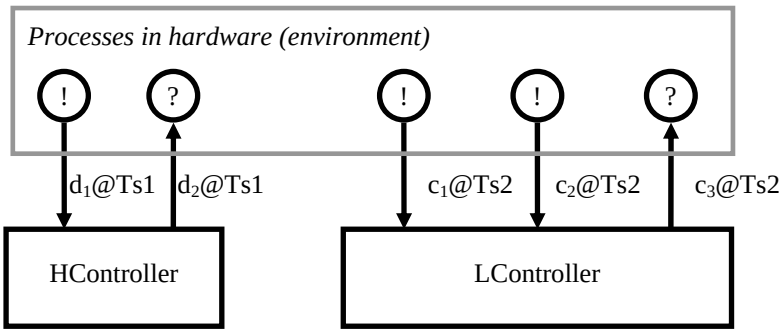
In this example we will illustrate the use and mechanism for sampling. Consider the two controller processes HController and LController depicted in Figure 2. The external channels d_1 and d_2 are timed on sample interval T_{s1} and external channels c_1, c_2 and c_3 on sample interval T_{s2}. In the following code we create the external channels and assign them to the environmental process with specified start time and sampling interval. The sampling rate for HController is 1 kHz and the sampling rate for LController is 0.1 kHz. The start time is specified such that sampling starts when everything is constructed otherwise deadlines may be passed on start-up.

```
//--- create external channels
Channel<int> d1 = new ADC(0);
Channel<int> d2 = new DAC(0);
Channel<int> c1 = new ADC(1);
Channel<int> c2 = new IncCounter(0);
Channel<int> c3 = new DAC(1);

//--- set up sampling timing and register channels to environment
long Ts1 =  1000; // in usec
long Ts2 = 10000; // in usec

long starttime = Environment::time() + 100000;

// firstly the inputs
Environment::at(d1, starttime, Ts1);
Environment::at(c1, starttime, Ts2);
Environment::at(c2, starttime, Ts2);

// secondly the outputs
Environment::at(d2, starttime, Ts1);
Environment::at(c3, starttime, Ts2);

//--- create processes and compositional relationships
...
```

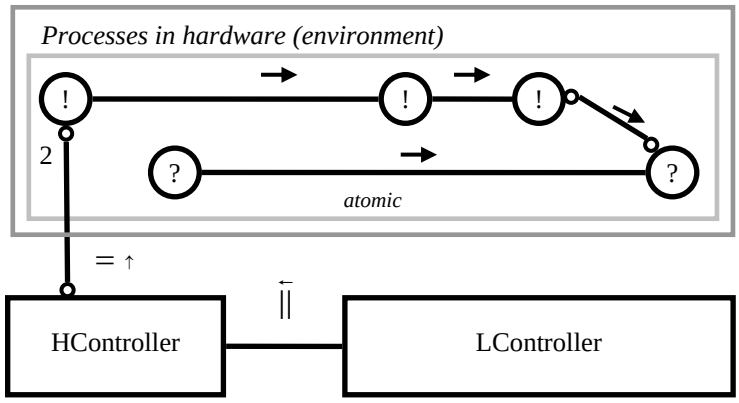
Although processes can read and write on these channels in parallel, the conversions will be in some sequence. This is because the environment uses the interrupt service routine that demands an atomic sequence.

Every registration with the same start time and sampling interval belong to the same atomic group and its order of execution is determined by the sequence of registration. Therefore, the sequence of registration can be important. The sequence of inputs and outputs will be sorted by its time stamp and when the time stamps are equal then the sequence is determined by the sequence of registration. Due to this constraint, the programmer can minimize conversion latencies be choosing a most optimal order of registration. We will not discuss the implementation of how this is achieved in this paper.

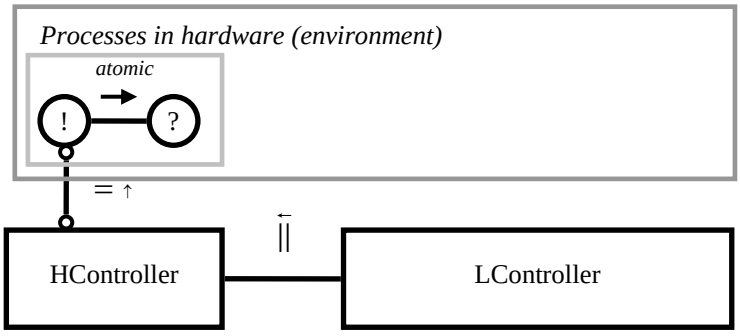
We will illustrate this mechanism in Figure 5 using CSP diagram notations. Figure 5a shows the communication relationships of both controllers with their input-output counterparts in hardware (the input/output bubbles in the grey rectangle). In Figure 5b-d, the compositional relationships between these the hardware inputs/outputs are rendered for



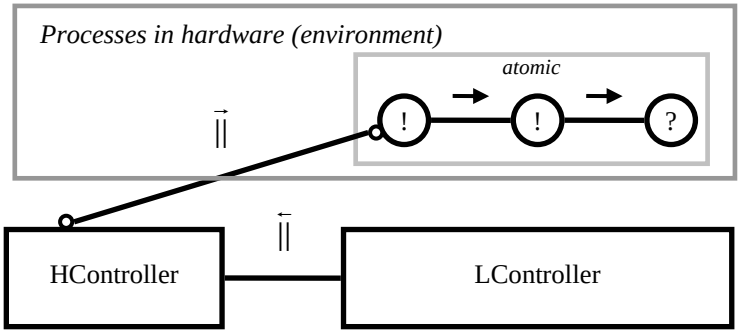
(a) Communication relationships



(b) Compositional relationships on $t = t_{s1} = t_{s2}$



(c) Compositional relationships on $t = t_{s1}$ and $t \neq t_{s2}$



(d) Compositional relationships on $t = t_{s2}$ and $t \neq t_{s1}$

Figure 5 Atomic sequence of inputs and outputs by the environmental process

different scenarios. This is the solution for using interrupt handling on the internal timer.

Process LController has a lower sampling frequency ($1/T_{s2}$) than the sampling frequency

($1/T_{s1}$) of process HController, with $T_{s1} < T_{s2}$. Thus, we specify that LController gets a lower priority than HController.

The conversions of the devices are done atomically. That is, they cannot be interrupted by the application. This is depicted by the atomic rectangles in grey. Processor interrupt mechanisms are sequential and mostly priority or pre-emption based. Therefore, the sequential compositions and the prioritized parallel composition with the hardware input/output processes are enforced by this environment.

This mechanism adapt to three scenarios, where:

1. $t = t_{s1} = t_{s2}$ $\Leftrightarrow t_{s1} = t_{s2}$
2. $t = t_{s1}$ and $t \neq t_{s2}$ $\Rightarrow t_{s1} \neq t_{s2}$
3. $t = t_{s2}$ and $t \neq t_{s1}$ $\Rightarrow t_{s1} \neq t_{s2}$

with variable t be the actual time. The sampling times are defined as:

$$t_{s1} \in \{starttime + k_1 * T_{s1} | \text{ for } k_1 = 0,1,2,3,\ldots\}$$

$$t_{s2} \in \{starttime + k_2 * T_{s2} | \text{ for } k_2 = 0,1,2,3,\ldots\}$$

Scenario 1: Figure 5b shows two equal time stamps. Sampling and actuation of all channels will be done in a predefined sequence. Firstly, all sampling processes must be performed and secondly all actuation processes must be performed. This introduces some jitter between sampling and a varying distance between sampling and actuation. Usually this jitter is so small that it does not endanger the stability or the accuracy of the controllers.

If it is really required that this variety is not allowed then all conversions should be done at the same time at a fixed sequence. This can also be specified with a sequence of `at(…,starttime,Ts)` registrations. In this case scenario 2 and 3 do not exist. The problems can be that this solution consumes more energy and the quality and costs of all components depend on the highest sampling frequency in the system. This may require more expensive converters. **Scenario 2**: Figure 5c shows the situation when only time stamp t_{s1} is reached. **Scenario 3**: Figure 5d shows the situation when only time stamp t_{s2} is reached. However, in the practical case presented in this paper, Scenario 3 will never occur when frequencies are multiplicities like in our JIWY case study of 1 kHz and 0.1 kHz. It is included here for completeness.

We have applied the environmental process concept successfully for a mechatronic system, called JIWY [11]. The control software is programmed in C++ with CTC++. A small variation of jitter between sampling and actuation is allowed, and therefore both scenario 1 and 2 apply for JIWY.

6. Conclusions

In this paper we described a concept of timing for untimed CSP architectures which involves an environmental process that participates in communication events. An environmental process is basically another CSP process that synchronizes on internal events. This approach allows adequate timing on external channel communication that provides sampling for control systems; something that was not possible with a timer object or a sleep method. Proposing multi-way events contributed in an orthogonal concept that does not require changes to an untimed CSP architecture. This concept may increase the extendibility, portability, and maintainability of CSP based software designs.

References

[1] C.A.R. Hoare. Communicating Sequential Processes. Prentice-Hall. London, UK, 1985.
[2] A.W. Roscoe. The Theory and Practice of Concurrency. C. A. R. Hoare and R. Bird *Series in Computer Sciences*, Prentice-Hall. 1998. 0-13-674409-5.
[3] G.H. Hilderink, A.W.P. Bakkers and J.F. Broenink. A Distributed Real-Time Java System Based on CSP. In *The Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC-2000)*, IEEE Computer Society. Newport Beach, California**:** pp. 400-407, 2000.
[4] G.H. Hilderink. A Graphical Modelling Language for Specifying Concurrency based on CSP. In *IEE Proceedings Software*, IEE. **150:** 108-120, 2002. ISSN 1462-5970.
[5] INMOS. occam 2 Reference Manual. C. A. R. Hoare *International Series in Computer Science*, Prentice Hall. 1988. ISBN 0-13-629312-3.
[6] P.H. Welch and P.D. Austin. The JCSP home page, http://www.cs.ukc.ac.uk/projects/ofa/jcsp, 1999.
[7] G.H. Hilderink. Communicating Threads for Java (CTJ) home page, http://www.rt.el.utwente.nl/javapp, 2002.
[8] S. Schneider. Concurrent and Real-time Systems. S. U. U. D. Barron and B. U. U. P. Wegner *Worldwide series in computer science*, John Wiley & Sons. Chichester, UK, 2000. 0-471-62373-3.
[9] U.o. Oxford. Better Ways to Program Parallel Processors, http://oldwww.comlab.ox.ac.ul/oucl/ocpara/methods.html, (UK).
[10] G. Jones. On Guards. In E. T. Muntean *7th Occam User Group & International Workshop on Parallel Programming of Transputer based Machines*, LGI-IMAG. Grenoble, 1987.
[11] D.S. Jovanovic, G.H. Hilderink and J.F. Broenink. A Communicating Threads (CT) Case Study: JIWY. In J. S. Pascoe, P. H. Welch, R. J. Loader and V. S. Sunderam *Communicating Process Architecture 2002*, IOS Press. University of Reading, UK. **60:** pp. 311-320, 2002. ISSN 1383-7575.